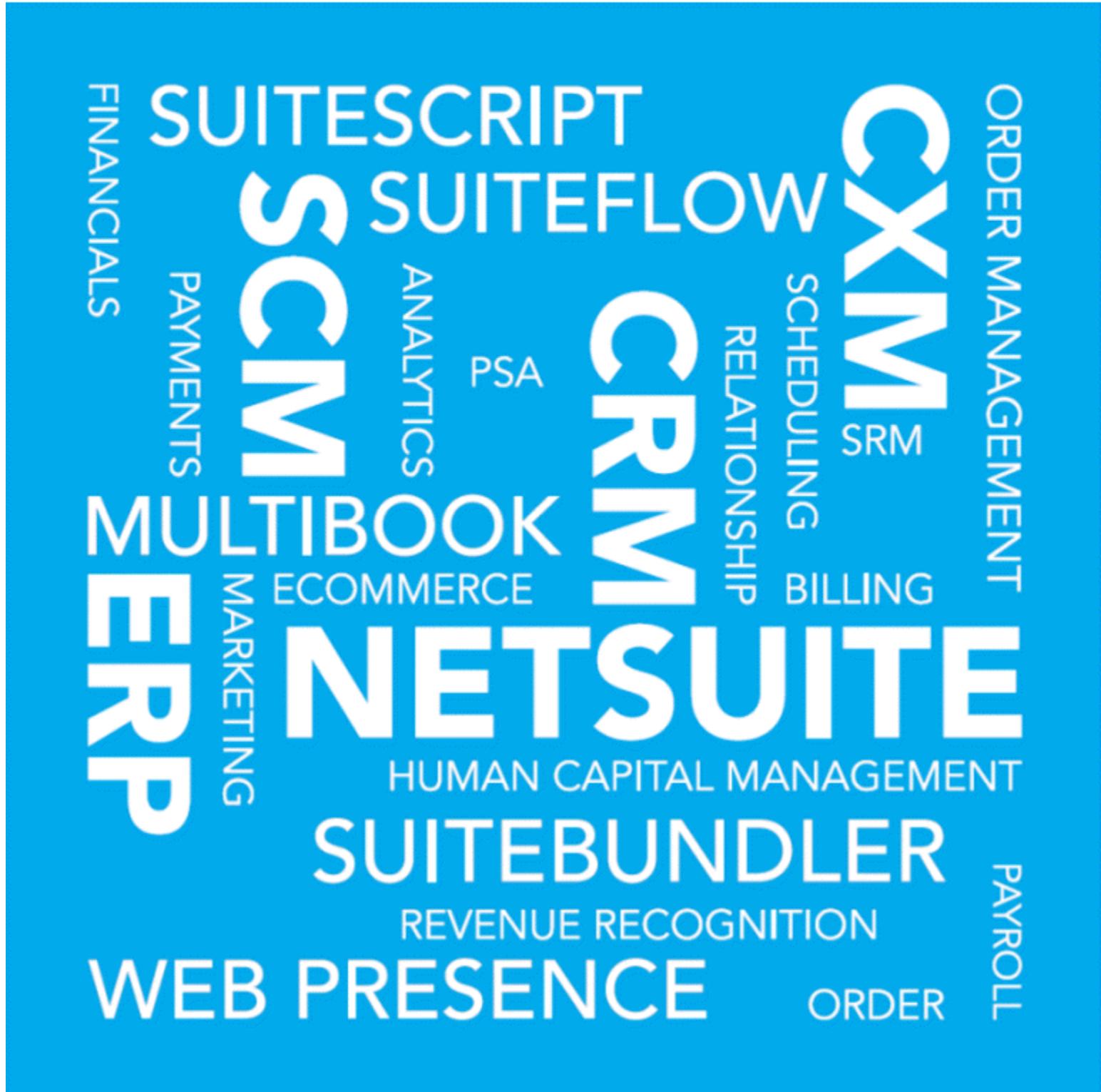


SuiteCommerce Site Development



Copyright © 2005, 2019, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

If this document is in public or private pre-General Availability status:

This documentation is in pre-General Availability status and is intended for demonstration and preliminary use only. It may not be specific to the hardware on which you are using the software. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to this documentation and will not be responsible for any loss, costs, or damages incurred due to the use of this documentation.

If this document is in private pre-General Availability status:

The information contained in this document is for informational sharing purposes only and should be considered in your capacity as a customer advisory board member or pursuant to your pre-General Availability trial agreement only. It is not a commitment to deliver any material, code, or functionality, and

should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle Master Agreement, Oracle License and Services Agreement, Oracle PartnerNetwork Agreement, Oracle distribution agreement, or other license agreement which has been executed by you and Oracle and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Sample Code

Oracle may provide sample code in SuiteAnswers, the Help Center, User Guides, or elsewhere through help links. All such sample code is provided "as is" and "as available", for use only with an authorized NetSuite Service account, and is made available as a SuiteCloud Technology subject to the SuiteCloud Terms of Service at www.netsuite.com/tos.

Oracle may modify or remove sample code at any time without notice.

No Excessive Use of the Service

As the Service is a multi-tenant service offering on shared databases, Customer may not use the Service in excess of limits or thresholds that Oracle considers commercially reasonable for the Service. If Oracle reasonably concludes that a Customer's use is excessive and/or will cause immediate or ongoing performance issues for one or more of Oracle's other customers, Oracle may slow down or throttle Customer's excess use until such time that Customer's use stays within reasonable limits. If Customer's particular usage pattern requires a higher limit or threshold, then the Customer should procure a subscription to the Service that accommodates a higher limit and/or threshold that more effectively aligns with the Customer's actual usage pattern.

Beta Features

Oracle may make available to Customer certain features that are labeled "beta" that are not yet generally available. To use such features, Customer acknowledges and agrees that such beta features are subject to the terms and conditions accepted by Customer upon activation of the feature, or in the absence of such terms, subject to the limitations for the feature described in the User Guide and as follows: The beta feature is a prototype or beta version only and is not error or bug free and Customer agrees that it will use the beta feature carefully and will not use it in any way which might result in any loss, corruption or unauthorized access of or to its or any third party's property or information. Customer must promptly report to Oracle any defects, errors or other problems in beta features to support@netsuite.com or other designated contact for the specific beta feature. Oracle cannot guarantee the continued availability of such beta features and may substantially modify or cease providing such beta features without entitling Customer to any refund, credit, or other compensation. Oracle makes no representations or warranties regarding functionality or use of beta features and Oracle shall have no liability for any lost data, incomplete data, re-run time, inaccurate input, work delay, lost profits or adverse effect on the performance of the Service resulting from the use of beta features. Oracle's standard service levels, warranties and related commitments regarding the Service shall not apply to beta features and they may not be fully supported by Oracle's customer support. These limitations and exclusions shall apply until the date that Oracle at its sole option makes a beta feature generally available to its customers and partners as part of the Service without a "beta" label.

Send Us Your Feedback

We'd like to hear your feedback on this document.

Answering the following questions will help us improve our help content:

- Did you find the information you needed? If not, what was missing?
- Did you find any errors?
- Is the information clear?
- Are the examples correct?
- Do you need more examples?
- What did you like most about this document?

Click [here](#) to send us your comments. If possible, please provide a page number or section title to identify the content you're describing.

To report software issues, contact NetSuite Customer Support.

Table of Contents

Developer Tools	10
Overview	10
Developer Environment	11
Install Node.js	11
Install Gulp.js	12
Set Up Theme Developer Tools	13
Set Up Extension Developer Tools	14
Theme Developer Tools	16
Fetch Active Theme Files	17
Test a Theme on a Local Server	19
Deploy a Theme to NetSuite	21
Extension Developer Tools	24
Create Extension Files	24
Create a Baseline Extension	24
Create Additional Modules for an Extension	26
Create Custom Content Types for an Extension	27
Fetch Active Theme and Extension Files	28
Test an Extension on a Local Server	30
Deploy an Extension to NetSuite	31
Developer Tools Reference	34
Developer Tools Roles and Permissions	34
Confirm That You Have the SCDeployer Role	34
Prepare the SCDeployer Role	34
Set Up the SCDeployer Role Manually	35
Gulp Command Reference for Themes and Extensions	37
Theme Development Files and Folders	41
Extension Development Files and Folders	43
Mixed Domains in a Local Server	46
Secure HTTP (HTTPS) with the Local Server	47
Deploy to a NetSuite Sandbox	50
Deploy to a Custom SSP Application	51
Troubleshooting the Developer Tools	54
Site Configuration	58
Overview	58
Configure Properties	59
Configuration Properties Reference	62
Advanced Tab	62
Backend Subtab	63
Cache Subtab	65
Custom Fields Subtab	66
Favicon Path Subtab	66
Filter Site Subtab	66
Image Resize Subtab	67
Pagination Subtab	67
Search Results Subtab	70
Checkout Tab	70
Credit Card Subtab	74
Forms Subtab	75
Payment Methods Subtab	76
Integrations Tab	77
AddThis Subtab	77
Bronto Subtab	79
Categories Subtab	79

Facebook Subtab	83
Google AdWords Subtab	86
GooglePlus Subtab	87
Google Tag Manager Subtab	90
Google Universal Analytics Subtab	91
Pinterest Subtab	92
Site Management Tools Subtab	95
Twitter Subtab	97
Layout Tab	100
Bottom Banner Images Subtab	100
Carousel Images Subtab	101
Color Palettes Subtab	101
Cookies Warning Banner Subtab	102
Footer Subtab	103
Header Subtab	103
Images Subtab	104
Light Colors Subtab	104
Navigation Subtab	104
Legacy Tab	105
Newsletter Subtab	105
Footer Subtab	106
Multi-Domain Tab	107
Hosts Subtab	107
Translations Subtab	109
My Account Tab	109
Addresses Subtab	110
Cases Subtab	110
List Header Subtab	111
Overview Subtab	111
Quotes Subtab	112
Return Authorization Subtab	115
SCIS Integration Subtab	116
Transaction List Columns Subtab	117
Search Tab	121
Result Display Options Subtab	121
Result Sorting Subtab	122
Search Results Subtab	123
Search Results per Page Subtab	124
Type Ahead Subtab	124
Shopping Tab	125
Item Options Subtab	125
Newsletter Subtab	125
Quick Order Subtab	126
Reviews Subtab	126
Wishlist Subtab	129
Shopping Catalog Tab	131
Facets Subtab	134
Facets Delimiters Subtab	136
Facets SEO Subtab	138
Item Options Subtab	139
Multi-Image Option Subtab	142
Product Details Information Subtab	142
Recently Viewed Items Subtab	143
Store Locator Tab	144
Store Locator Subtab	144

Store Locator Google Maps Subtab	147
Configuration File Types	150
JSON Configuration Files	150
Create JSON Configuration Files	151
JavaScript Configuration Files	168
Site Management Tools Configuration	170
Upgrade from SMT Version 2 to SMT Version 3	170
SMT Templates and Areas	172
Site Management Tools Areas	172
Templates and Areas for SCA	176
SMT Custom Preview Screen Sizes	182
Working with SMT Landing Pages in a Sandbox Account	183
Changing SMT to Use a Different Hosting Root	184
Configuring Escape to Log In	185
Internationalization of SMT Administration	186
CMS Records for SMT	186
Custom Record for CMS Content	187
CMS Contents Record	188
CMS Page Record	190
CMS Page Type Record	191
Custom Content Type	193
Custom Record for Custom Content Type	194
CMS Content Type Record	196
Customization	198
Overview	198
Themes	200
Themes Overview	200
Develop Your Theme	201
Customize Pre-Existing Theme Files	202
Add a New File to a Theme	203
Override Active Extension Files	205
Set Up Your Theme for Customization in SMT	207
Customize Site Search Elements	216
Best Practices for Creating Themes	218
SuiteCommerce Base Theme	218
General Best Practices for Themes	219
HTML Best Practices	219
Sass Best Practices	221
Asset Best Practices	223
Design Architecture	227
Theme Manifest	234
Extensions	239
Develop Your Extension	239
Create Page Types	241
Add a CMS Page Type Record	243
Create Layout Thumbnails	244
Extension Manifest	244
Troubleshoot Activation Errors	250
Commerce Custom Fields	251
Create Custom Fields Using an Extension	252
Create Custom Fields by Customizing Templates	256
Theme and Extension SuiteApps	264
Bundle Themes and Extensions as SuiteApps	264
Update Themes and Extensions	267
Core SCA Source Code	269

Core SuiteCommerce Advanced Developer Tools	269
Set Up Your Development Environment	270
SCA on a Local Server	274
Deploy to NetSuite	275
Gulp Command Reference for SCA Developer Tools	277
The Build Process	278
Customize and Extend Core SuiteCommerce Advanced Modules	283
Best Practices for Customizing SuiteCommerce Advanced	284
Customization Examples	288
Architecture	320
Overview	320
Core Framework Technologies	321
Model View Controller (MVC) and Backbone.js	321
Asynchronous Module Definitions (AMD) and RequireJS	322
Logic-less Templates and Handlebars.js	322
Templates and the Template Context	323
SuiteCommerce Modules	327
The Modules Directory	327
Dependencies	328
Application Modules	328
Module Architecture	329
Entry Point	330
Routers	330
Views	330
Models, Collections, and Services	334
Product Details Page Architecture	352
Patches	357
Overview	357
Patches	358
Standard Promotion with Inline Discount and Rate as Percentage Not Updating the Amount in Checkout	363
Incorrect Value for Shipping Estimate Occurs in Shopping Cart	366
Page With URL Fragments Redirects Too Many Times	369
See Complete List of Stores Link on Store Locator Page Does Not Show Store List	371
Quantity Pricing Displayed in Web Store Even When "Require Login for Pricing" is Enabled	374
Edited Shipping Address on the Review Your Order Page is Not Showing Changes	377
Custom Page Title in SMT Does Not Display Correctly	379
Currencies Change to the Default in the Shopping Application	382
Order Confirmation Page Not Displayed When Using Single Page Checkout and External Payment	385
Incorrect Discounted Amounts on Checkout Summary	387
DeployDistribution Folder Does Not Include Local Files	390
HTML List Styles Do Not Display	392
HTML List Styles Do Not Display (Kilimanjaro)	392
HTML List Styles Do Not Display (Elbrus and Vinson)	395
HTML List Style Does Not Display (Theme)	398
Item Search Displays Incorrect Results	401
Category Product Lists Return Page Not Found	403
Cannot Test an Extension on a Local Server	407
Secure Shopping for Site Builder Implementations	410
Secure Shopping for Site Builder Extensions (Vinson)	411
Secure Shopping for Site Builder (Pre-Denali)	418
Cannot Scroll Through Long Menu Lists Using iOS	422
Elbrus Release and Earlier – Menu List Scrolling Patch (iOS)	423
Kilimanjaro – Menu List Scrolling Patch (iOS)	425

Pages Not Indexed Using Google's Mobile-First Indexing	429
Disabling Display of SuiteCommerce Gift Wrap & Message Extension Transaction Line Fields	431
Users Not Redirected to External Payment System	436
Incorrect Redirect URL for External Payments	439
Invoices Do Not Include the Request for a Return Button	443
npm Error on Implementations	447
Content Appears Incorrectly in a Merchandising Zone	448
Reference My Account Generates Error on Load	452
Error Loading Shopping Page Due to Uncaught TypeError	453
Users Redirected to Checkout Application Instead of Shopping Application	456
Add to Cart Button Does Not Work If Quality Field Selected	458
URLs with Redundant Facets Generated	460
Content Flickers or Disappears When Browsing the Product Listing Page	465
Enabling Google AdWords Causes Error on Login	467
URL for Commerce Categories Contains Incorrect Delimiters	470
Order Summary for Item-Based Promotions	474
CSS Error Hides First div Element on Product Details Page	477
Invoice Terms Not Included In Order Details	479
Users Required to Re-enter Credit Card Payment Method Details on Payment Page	480
Selected Invoice Not Displayed When Making an Invoice Payment	484
Log In to See Prices Message Appears When Users are Logged In	487
Item Record HTML Meta Data Not Appearing on Page Meta Data	490
Delivery Options Not Appearing After Editing the Cart and Re-entering a Shipping Address	492
Order Confirmation and Thank You Page is Blank	496
Matrix Item Options Not Displaying With Google Tag Manager Enabled	500
Delivery Methods Not Appearing in One Page Checkout	504
Mastercard 2-Series BIN Regex Patch	508
Denali – Mastercard Regex Patch	508
Mont Blanc – Mastercard Regex Patch	512
Vinson – Mastercard Regex Patch	515
Auto-Apply Promotions for Elbrus	518
Modifications to Existing Promotions Code	518
Custom PromocodeNotifications Module	532
Change Email Address Patch	536
Denali — Change Email Address Patch	537
Mont Blanc — Change Email Address Patch	555
Vinson — Change Email Address Patch	572
Elbrus — Change Email Address Patch	589
Duplicate Product Lists in Internet Explorer 11	607
Save for Later Item not Moved to Cart	608
Running Gulp Commands Results in a Syntax Error	610
Missing Promo Code on Return Request	612
Enhanced Page Content Disappears when Resizing the Browser	615
Invoices Page Displays Incorrect Date Sort (pre-Denali)	620
PayPal Payments Cause Error at Checkout	622
Canonical Tags Populated With Relative Paths	625
Shopping Cart Not Scrolling (Mobile)	628
Error When Adding Items to Categories in Site Management Tools	631
Item Search API Response Data not Cached	636
Secure Shopping Domains (Elbrus, Vinson, Mont Blanc, and Denali)	638
Secure Shopping Domain (pre-Denali)	639
Migration Tasks	640
PayPal Address not Retained in Customer Record	655
Login Email Address Appears in the Password Reset URL	656
How to Apply .patch Files	660

Upgrade SuiteCommerce to SuiteCommerce Advanced	664
Update SuiteCommerce Advanced	666
Commerce Releases and Versioning	666
Migrate to the Latest Release of SuiteCommerce Advanced	667
Migrate from Aconcagua and Later	668
Migrate From Kilimanjaro and Earlier	670

Overview

 **Applies to:** SuiteCommerce Web Stores | Aconcagua

Before customizing a SuiteCommerce site, you must set up the developer tools. These tools let you:

- Create your own themes and extensions.
- Compile your themes and extensions locally for testing.
- Deploy themes and extensions to your NetSuite account. You can also deploy directly to your production or sandbox accounts in NetSuite.
- Create and edit code locally in your preferred text editor or IDE.
- Retrieve pre-existing theme source code as a basis for building a custom theme.
- Build baseline extensions to access the Extensibility API.
- Expose theme Sass variables for the Site Management Tools Theme Customizer.

Follow the instructions in these topics to set up and use the developer tools:

- [Developer Environment](#) – This topic explains how to create your developer environment. The term **developer environment** refers to both the tools used to develop your code and the physical location where you store it all.
- [Theme Developer Tools](#) – These topics explain how to use the theme developer tools to start building your own themes, test them locally, and deploy them to a NetSuite account.
- [Extension Developer Tools](#) – These topics explain how to use the extension developer tools to start building your own extensions, test them locally, and deploy them to a NetSuite account.
- [Developer Tools Reference](#) – These topics provide a handy reference of each Gulp.js command, what your theme and extension workspaces look like, plus important information on testing locally and deploying to NetSuite.

Developer Environment

ⓘ Applies to: SuiteCommerce Web Stores

 [View a Related Video](#)

SuiteCommerce provides different tools that let you create, test, and deploy code as either themes or extensions. The developer tools you need depend on your SuiteCommerce implementation. The lists below explain what tools you need by implementation. After determining the correct tools you need, use the provided checklist to ensure a complete setup.

SuiteCommerce

- [Install Node.js](#)
- [Install Gulp.js](#)
- [Set Up Theme Developer Tools](#)
- [Set Up Extension Developer Tools](#)

SuiteCommerce Advanced (Aconcagua release and later)

- [Install Node.js](#)
- [Install Gulp.js](#)
- [Set Up Theme Developer Tools](#)
- [Set Up Extension Developer Tools](#)
- [Core SuiteCommerce Advanced Developer Tools](#)
- [Set Up SCA 2019.1 for Theme and Extension Developer Tools](#) (required for SCA 2019.1)



Important: If you are implementing the Aconcagua Release of SCA or later, the best practice is to use themes and extensions to customize your site. If you require access to objects not available using the Extensibility API, use the SCA developer tools and customization practices outlined in [Core SCA Source Code](#).

SuiteCommerce Advanced (Kilimanjaro release and earlier)

- [Install Node.js](#)
- [Install Gulp.js](#)
- [Core SuiteCommerce Advanced Developer Tools](#)



Need help? Download the [Developer Tools Checklist](#) for a handy quick reference.

Install Node.js

Node.js is the platform required for all Gulp.js tasks required to develop any SuiteCommerce site. Node.js is available at:

<https://nodejs.org/en/download/>

Install the version of Node.js that is supported by your implementation according to the table below. Use the most current minor release (designated by x in the table below) for the version of Node.js you are using.

SuiteCommerce Implementation	Supported Node.js Versions
SuiteCommerce	10.15.x
SuiteCommerce Advanced — 2019.1	10.15.x
SuiteCommerce Advanced — 2018.2	8.11.4
SuiteCommerce Advanced — Aconcagua	8.9.x LTS
SuiteCommerce Advanced — Kilimanjaro	6.11.x
SuiteCommerce Advanced — Elbrus	4.x.x LTS and Later
SuiteCommerce Advanced — Vinson	4.x.x LTS
SuiteCommerce Advanced — Mont Blanc	4.4.x and 0.12.x
SuiteCommerce Advanced — Denali	0.12.x

When installing Node.js, ensure that you install all Node.js components. This includes the Node Package Manager (NPM), which is used to install Gulp.js and other files required by the developer tools and the SuiteCommerce build process.

After running the installer, you should see Node.js in the list of available programs on your system. The `npm` command should also be added to the path on your system.

Note: Ensure that you use the correct installer for your platform. You may have to restart your machine for the Node.js commands to be recognized in the path variable on your system.

To verify that Node.js is installed correctly:

1. Depending on your platform open a command line or terminal window.
2. Enter the following command:
`node -v`

If everything is installed correctly, this command outputs the currently installed version of Node.js. There should be no errors or warnings.

After successfully installing Node.js, you are ready to [Install Gulp.js](#).

Install Gulp.js

After successfully installing Node.js, the next step in setting up your developer environment is to install Gulp.js. Gulp.js is a third-party, open-source tool that automates many of the tasks related to creating web-based applications. This is required for all SuiteCommerce implementations.

To Install Gulp.js:

1. Depending on your platform, open a command line or terminal window.

2. Enter the following command:

```
npm install --global gulp
```



Note: Install Gulp.js using the `--global` flag as shown above. On Linux and MacOS, you must run this command using sudo. This is required to ensure that Gulp.js is installed correctly.

3. Verify that Gulp.js was installed correctly by entering the following command:

```
gulp -v
```

If installed correctly, this command outputs the currently installed version of Gulp.js. Ensure that this version is 3.9.1 or higher. There should be no errors or warnings.

Permissions

To use Gulp.js to deploy source files to NetSuite, you must use a **System Administrator** role with the following permissions set to **Full**:

- Documents and Files
- Website (External) publisher
- Web Services

After successfully installing Gulp.js, you are ready to set up one or more of the following:

- Set Up Theme Developer Tools
- Set Up Extension Developer Tools
- Core SuiteCommerce Advanced Developer Tools

Set Up Theme Developer Tools

Applies to: SuiteCommerce Web Stores | Aconcagua

Before you can create a theme, you must download the theme developer tools and extract them to create a top-level development workspace. This is where you maintain a theme's HTML, Sass, and asset files. You use the developer tools to run Gulp.js commands to fetch files from the server, test your changes locally, and deploy themes to NetSuite.



Important: You can only develop one theme per top-level theme workspace. To develop two or more themes simultaneously, you must set up multiple instances of the theme developer tools, one for each theme.



Note: These tools are required for all SuiteCommerce sites and any SCA sites implementing the Aconcagua release or later. You cannot customize your SCA site's Sass or HTML template files without these tools.

To download and extract theme developer tools:

1. Login to your NetSuite account.
2. In NetSuite, go to Documents > Files > File Cabinet.

3. Navigate to SuiteBundles/Bundle 284094/.
 4. Download the .zip file you find there:
ThemeDevelopmentTools-19.1.x.zip (where x equals the latest minor release).
 5. Extract the .zip file to a location in your local environment. This becomes your root development directory for custom themes.
- The .zip file extracts into a directory named **ThemeDevelopmentTools-19.1.x** by default (where x equals the latest minor release). However, you can rename this directory to suit your needs.



Note: **SuiteCommerce Advanced Developers:** The 2019.1 release of the Extension Management Bundle includes the theme and extension developer tools for 2019.1 and 2018.2. If you are implementing the 2018.2 release of SCA, you must use the 2018.2 developer tools when creating themes and extensions due to non-backward compatibility.



Important: Do not move, delete, or rename any files or folders within the top-level development directory.

6. Open a command line or terminal window.
7. Access your root development directory created previously.
8. Enter the following command to install additional Node.js packages into this directory:

```
npm install
```



Note: This command installs the dependencies required to manage your custom themes. These files are stored in the node_modules subdirectory of the root development directory. This command may take several minutes to complete.

You are now ready to begin theme development. See [Theme Developer Tools](#) for information on fetching an active theme, testing on a local server, and deploying to a NetSuite account. For information on developing a theme, see [Themes](#).

If you also intend to build extensions, you must [Set Up Extension Developer Tools](#).

Set Up Extension Developer Tools

Applies to: SuiteCommerce Web Stores | Aconcagua

Before you can create an extension, you must download the extension developer tools and extract them to create a top-level development directory. This is where you maintain all of your extension's JavaScript, SuiteScript, Configuration, HTML, Sass, and assets. You use the tools to run Gulp.js commands to build baseline files for your extension, test your changes locally, and deploy extensions to NetSuite.



Note: These tools are required for all SuiteCommerce site and any SCA sites implementing the Aconcagua release or later. You build extensions to interact with the Extensibility API to extend the application. See the help topic [Extensibility Component Classes](#) for an explanation of what components are currently accessible using the Extensibility API.

To download and extract the extension developer tools:

1. Login to your NetSuite account.
2. In NetSuite, go to Documents > Files > File Cabinet.

3. Navigate to SuiteBundles/Bundle 284094/.
 4. Download the .zip file you find there:
ExtensionDevelopmentTools-19.1.x.zip (where x equals the latest minor release).
 5. Extract the .zip file to a location in your local environment. This becomes your root development directory for your custom extensions.
- The .zip file extracts into a directory named **ExtensionDevelopmentTools-19.1.x** by default (where x equals the latest minor release). However, you can rename this directory to suit your needs.



Note: **SuiteCommerce Advanced Developers:** The 2019.1 release of the Extension Management Bundle includes the theme and extension developer tools for 2019.1 and 2018.2. If you are implementing the 2018.2 release of SCA, you must use the 2018.2 developer tools when creating themes and extensions due to non-backward compatibility.



Important: Do not move, delete, or rename any files or folders within the top-level development directory.

6. Open a command line or terminal window.
7. Access your root development directory created previously.
8. Enter the following command to install additional Node.js packages into this directory:
`npm install`



Note: This command installs the dependencies required to manage your custom extensions. These files are stored in the node_modules subdirectory of the root development directory. This command may take several minutes to complete.

You are now ready to begin extension development. See [Extension Developer Tools](#) for information on building a baseline extension, testing on a local server, and deploying to a NetSuite account. For information on developing an extension, see [Extensions](#).

If you also intend to create themes, you must [Set Up Theme Developer Tools](#).

Theme Developer Tools

 **Applies to:** SuiteCommerce Web Stores

The theme developer tools are required for theme developers. This applies to all SuiteCommerce and any SuiteCommerce Advanced implementations using Aconcagua release and later. After you have successfully installed the theme developer tools you can do the following:

- [Fetch Active Theme Files](#)
- [Test a Theme on a Local Server](#)
- [Deploy a Theme to NetSuite](#)



Important: These procedures assume that you have successfully set up your developer environment to use the theme developer tools. See [Developer Environment](#) for details.

Before You Begin

Be aware of the following important information:

- The `gulp theme:deploy` command checks to see if the theme you have customized is a published theme or a previously deployed, custom theme.
 - When you deploy customizations to a published theme, the theme development tools require that you create a new, custom theme. You cannot overwrite content protected by copyright.
 - When you deploy customizations to a pre-existing custom theme, the theme development tools give you the option to create a new theme (using the existing theme as a baseline) or update the existing theme with a new revision.
- When you create a new custom theme, the developer tools rename the local directory where you created your customizations. This name matches the theme name you specify when you deploy your code.

Example: When you download the files for the **SuiteCommerce Base Theme**, the developer tools create the `SuiteCommerceBaseTheme` directory in your theme workspace. You make your customizations, test, and deploy to NetSuite. The developer tools prompt you to create a new theme, which you name **MyTheme1**. When the deployment is complete, your local theme directory is renamed to `MyTheme1`.

- If you deploy a theme with extension overrides and later activate new extensions for the same domain. Your deployed customizations do not apply to the new extensions. You must update your theme to include these customizations.
- During activation, if you decide not activate an extension for which you created an override within the active theme, that override does not take effect at runtime and has no impact on your domain.
- After deploying a theme or extension to NetSuite, you must activate the theme using the Manage Extensions wizard to apply your changes to a domain. Even if your theme is already active for a domain, you must reactivate your theme for your changes to compile.

Theme parameters, including your NetSuite email, account, role, and domain information, are stored in the `.nsdeploy` file when you fetch a theme. To reset your login and deployment information, delete the `.nsdeploy` file. This file is located here:

```
<LOCAL_SOURCEFILES_ROOT>/gulp/config/.nsdeploy
```

Likewise, theme information, including the theme name, fantasy name, version, and description, are stored in the theme's `Manifest.json` file. The manifest file is located here:

```
<LOCAL_SOURCEFILES_ROOT>/Workspace/<THEME_DIRECTORY>/manifest.json
```

Fetch Active Theme Files

To create a theme, you first download files for the currently active theme to use as the baseline for your own custom theme. You can download files of any published theme or any previously deployed custom theme.

When you run the `gulp theme:fetch` command, the theme developer tools download all editable theme-related files for the active theme and place them in your theme workspace. If you have any active extensions when you run the `gulp theme:fetch` command, the Sass, HTML, and assets of those extension download to your local environment as well. These are provided should you need to make changes to the extensions to match your new theme.



Warning: When you fetch a theme, the developer tools download the active theme's development files to your theme directory. This action overwrites any theme files currently stored in your local workspace. This can result in loss of development files if you are not careful. To ensure that you do not lose any work under development, do not run the `gulp theme:fetch` command until you have deployed your work to NetSuite. To develop two or more themes simultaneously, you must extract the theme developer tools into separate workspaces, one for each theme.



Important: You must at least have a theme activated for a domain before continuing with this procedure. If this is your first theme, activate the SuiteCommerce Base Theme to use as your baseline. This theme includes many best practices for creating a theme. See the help topic [Install Your SuiteCommerce Application](#) for a list of required SuiteApps and corresponding Bundle IDs.

To fetch the active theme and extension files:

1. Open a command line or terminal.
2. Access the top-level theme development directory you created when you downloaded the developer tools.
3. Enter the following command:
`gulp theme:fetch`
4. When prompted, enter the following information:
 - NetSuite Email – Enter the email address associated with your NetSuite account.
 - NetSuite Password – Enter your account password.



Note: The developer tools do not support emails or passwords containing special characters such as + and %.

- Choose Your Target Account and Role – Select the NetSuite account where your SuiteCommerce application is installed. You must specify the correct role with required permissions for connecting to NetSuite.



Important: You must log in using the SCDeployer role to access your NetSuite account. Failure to use this role may result in an error. See [Developer Tools Roles and Permissions](#) for instructions on setting up this role.

- Choose Your Website – Select your website that includes the domain you are customizing.
- Choose Your Domain – Select your domain with the active themes and extensions you want to download.

Your next step is to develop your theme. See [Themes](#) for details.

What Happens When You Fetch an Active Theme?

When you run the `gulp theme:fetch` command, the theme developer tools:

- Create the Workspace directory in your theme development directory. If the Workspace directory already exists, the developer tools clear its contents before downloading. This action overwrites any files currently stored in your local workspace with the files being fetched.
- Download all theme-related HTML and Sass files for the active theme and store them in your workspace by module within a directory specific to the theme. For example, Workspace/<THEME_DIRECTORY>/Modules. The theme directory is intuitively named to match the name of the active theme.
- Download all theme-related assets to Workspace/<THEME_DIRECTORY>/assets.
- Download all theme-related skin preset files to Workspace/<THEME_DIRECTORY>/Skins.
- Validate that all files declared in the theme's manifest.json have been downloaded correctly. If any problems occurred, the developer tools list all missing files and direct you to execute the `gulp theme:fetch` command again.

If the chosen domain includes any active extensions when you fetch the active theme, the developer tools:

- Download all extension-related HTML and Sass files for the active extensions and store them in your workspace by module within directories specific to each active extension. For example, Workspace/Extras/Extensions/<EXTENSION_DIRECTORY>/Modules. The extension directories are intuitively named to match the name of each active extension.
- Download all extension-related assets to/Workspace/Extras/Extensions/<EXTENSION_DIRECTORY>/assets.
- If you are downloading a theme that includes previously deployed overrides, the developer tools download these into/Workspace/<THEME_DIRECTORY>/Overrides.

When you fetch the active theme, you are basically downloading the HTML, Sass, and assets files for the active theme to use as a baseline for your own theme development. Because a published theme is protected by copyright, you cannot overwrite any files for that theme. You can, however, use an existing theme as a baseline for your own, which you later deploy as a unique theme. The SuiteCommerce Base Theme SuiteApp provides a good starting point for building your own theme.



Note: Although you cannot overwrite a published theme, you can make any development changes to your own themes.

When you fetch a theme for a domain, the developer tools download the HTML, Sass, and asset files for the active theme plus any HTML, Sass, and asset files of any extensions active at the time you run the `gulp theme:fetch` command. The local server needs these files to compile your theme when

testing locally. You also have the opportunity to override the HTML and Sass for the active extensions with custom changes or add new assets. Any overrides you declare affect the domain once activated. Be aware that this only affects your domain. You are not overwriting any published extension's code.

Example:

Your domain has an active theme, **SuiteCommerceBaseTheme**, and an active extension, **PublishedExtension1**. Your Vendor name is **ACME**. You run the `gulp theme:fetch` command and specify your domain.

In this example, your Workspace directory structure should look similar to following:

```
Workspace/
  Extras/
    Extensions/
      ACME/
        PublishedExtension1/
          assets/
          Modules/
          manifest.json
          application_manifest.json
      SuiteCommerceBaseTheme/
        assets/
        Modules/
        Overrides/
        Skins/
        manifest.json
```

When the gulp processes are complete your theme developer environment should look similar to the following:

File Types	Workspace Subdirectory
Theme-related HTML and Sass files	<code>../SuiteCommerceBaseTheme/Modules/</code>
Theme-related asset files	<code>../SuiteCommerceBaseTheme/assets/</code>
Theme-related skin presets	<code>../SuiteCommerceBaseTheme/Skins/</code>
Extension-related HTML and Sass source files (sorted by module)	<code>../Extras/Extensions/ACME/PublishedExtension1/Modules/</code>
Extension-related asset files	<code>../Extras/Extensions/ACME/PublishedExtension1/assets/</code>
Pre-existing extension overrides	<code>../SuiteCommerceBaseTheme/Overrides/</code>

Test a Theme on a Local Server

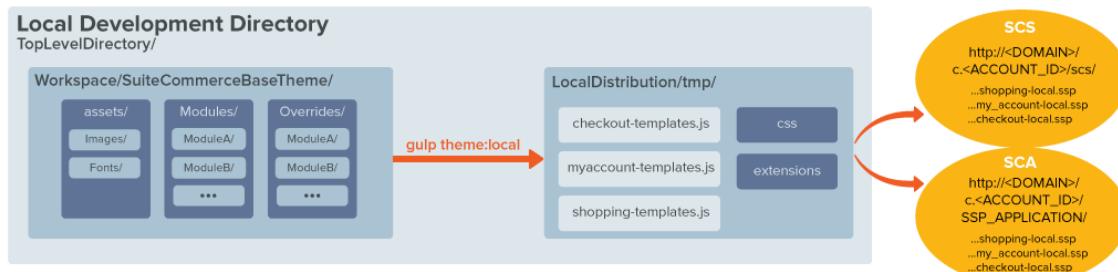
You can test your theme customizations on a local server before deploying to NetSuite. To set up your files for a local server, access the top-level directory in a command terminal and run the following command:

```
gulp theme:local
```

This command compiles all source files, including your theme and any extensions customizations, into combined files within a LocalDistribution/tmp directory. This directory contains the Sass and template files used by the local version of the application, so you can test your site before deploying anything to NetSuite.

When the server starts, Gulp.js initializes watch tasks that listen for changes to files in the Templates and Sass directories. When you save changes to a file, gulp automatically recompiles the source files and updates the LocalDistribution directory. Gulp also outputs a message to the console when an update occurs.

Important: The local server is primarily used to test frontend changes. You must deploy and activate your theme to view any changes you made to skins or to Sass variables exposed to SMT.



To run your Theme customizations on a local server:

1. In your local developer environment, open a command line or terminal and access the top-level development directory.
2. Run the following command:

```
gulp theme:local
```



Warning: Potential data loss. Besides compiling and deploying your theme to a local server, the `gulp theme:local` command updates the manifest.json file for the theme. This action overwrites any manual changes you made to this file. To preserve any manual changes to your manifest.json file, run the `gulp theme:local --preserve-manifest` command instead. See [Theme Manifest](#) for details on this file.

3. Navigate to the local version of the application using one of the following URLs:
 - `http://<DOMAIN_NAME>/c.<ACCOUNT_ID>/<SSP_APPLICATION>/shopping-local.ssp`
 - `http://<DOMAIN_NAME>/c.<ACCOUNT_ID>/<SSP_APPLICATION>/my_account-local.ssp`
 - `http://<DOMAIN_NAME>/c.<ACCOUNT_ID>/<SSP_APPLICATION>/checkout-local.ssp`
 - `DOMAIN_NAME` – replace this value with the domain name configured for your NetSuite website record.
 - `ACCOUNT_ID` – replace this value with your NetSuite account ID.
 - `SSP_APPLICATION` – replace this value with the URL root that you are accessing. For SuiteCommerce sites, this part of the URL should read **scs**.

The local server starts automatically. With the local server running, you can make changes to your local files and see the results immediately. Gulp.js automatically recompiles the application when you save any changes to theme-related HTML, Sass, or assets. Simply refresh your browser to see the changes.

You can also deploy your theme to NetSuite. This is required if you want to activate the theme on a domain. This is also required to test any Sass variables exposed to the Site Management Tools Theme Customizer. See [Deploy a Theme to NetSuite](#).



Important: If you add a new file or make changes to any overrides after launching the local server, you must run the `gulp theme:local` command again to include the new changes. Gulp.js does not automatically compile new files or process overrides.

Deploy a Theme to NetSuite

When you initially deploy a theme to NetSuite, you configure your environment to deploy to a specific account and SSP Application. For SuiteCommerce sites, this is straightforward (as a managed bundle, SuiteCommerce relies on the SuiteCommerce SSP Application only). If deploying to a SuiteCommerce Advanced (SCA) account, you must also configure the correct SSP Application, related to the current release you are implementing.

To deploy a theme, use the following command:

```
gulp theme:deploy
```

This command validates your customizations, copies them into a local DeployDistribution directory, and updates the manifest files with any necessary overrides. Gulp.js then uploads these files to your NetSuite account, making them available for activation.

When you deploy your theme to NetSuite, the developer tools upload your development files to a location in your NetSuite File Cabinet. The developer tools compile your theme files locally, but only for testing. Deploying a theme does not apply your theme to a domain. Deployment simply makes your theme available for activation in the Manage Extensions Wizard. See the help topic [Manage Themes and Extensions](#) for details.



Note: You cannot overwrite any themes protected by copyright. This includes any themes published by Oracle or Oracle-certified partners. You can, however, use these themes as a baseline for your own customizations.

For more Gulp.js commands used with themes and extensions, see [Gulp Command Reference for Themes and Extensions](#).

To deploy your theme to NetSuite:

1. In your local developer environment, open a command line or terminal and access the top-level development directory.
2. Run the following command:

```
gulp theme:deploy
```



Warning: Potential data loss. Besides compiling and deploying your theme, the `gulp theme:deploy` command updates the `manifest.json` file for the theme. This action overwrites any manual changes you made to this file. To preserve any manual changes to your `manifest.json` file, run the `gulp theme:deploy --preserve-manifest` command instead. See [Theme Manifest](#) for details on this file.

3. When prompted, enter the following information.

Unless otherwise noted, create all entries using only alphanumeric characters without spaces.



Note: These prompts appear the first time you deploy a new theme to NetSuite. Subsequent deployments of the same theme only prompt for your NetSuite password. To reset the login information and fill in these prompts again, use the `gulp theme:deploy --to` command.

- NetSuite Email – Enter the email address associated with your NetSuite account.
- NetSuite Password – Enter your account password.



Note: The developer tools do not support emails or passwords containing special characters such as + and %.

- Vendor – Enter the name of the vendor building this theme.
- Name – Enter a name for your custom theme.
- Version – Enter a version for your theme.



Important: To take advantage of theme update requirements, use semantic versioning (SemVer) notation. For more information, see <https://semver.org/>.

- Description – Enter a brief description for your theme. This will appear in NetSuite when you select themes to activate.
- Supported Products – Select the product or products (SuiteCommerce Online or SCIS) to which you are deploying. An asterisk (*) identifies a selected product.

SuiteCommerce Online applies to SuiteCommerce and SuiteCommerce Advanced.

- Use the spacebar key to select or deselect an option.
- Use the UP and DOWN arrow keys to scroll through the options.
- Toggle the A key to select or deselect all options.

Your customizations do not apply to your site until you activate the theme for a specific domain using the Manage Extensions wizard in NetSuite. The deploy process includes a notation reminding you of the domain and theme name to set during this process. See the help topic [Activate Themes and Extensions](#) for instructions on activating your new theme for the domain or domains of your choice.

What Happens When You Deploy a Theme?

The deployment process is specific to uploading theme files and extension overrides (if applicable) to a location in your NetSuite File Cabinet. This process only compiles your source files for testing before deployment. The deployment process does not deploy compiled files or associate any files with a site or domain. To do that, you must activate your theme using the Manage Extensions Wizard.

When you download a theme or extension's source files using the `gulp theme:fetch` command, you receive all HTML, Sass, overrides, and assets (images, fonts, etc.) of the active theme. This command places these files in your `Workspace/<THEME_DIRECTORY>` folder. This is where you build your custom themes and extension overrides.

During the deployment process:

- The developer tools copy all of your custom theme development files (modules, assets, and overrides) into the `DeployDistribution` folder in your top-level development directory. If this directory does not exist, the developer tools create it.
- The developer tools validate your customizations.
- The developer tools update the `manifest.json` file to include any overrides.

- The developer tools upload development files to your NetSuite file cabinet as bundled files (maintaining the same organizational structure).
- NetSuite creates an Extension record for the custom theme. Like a published theme or extension, these files have not compiled into usable files, but they are available for activation.



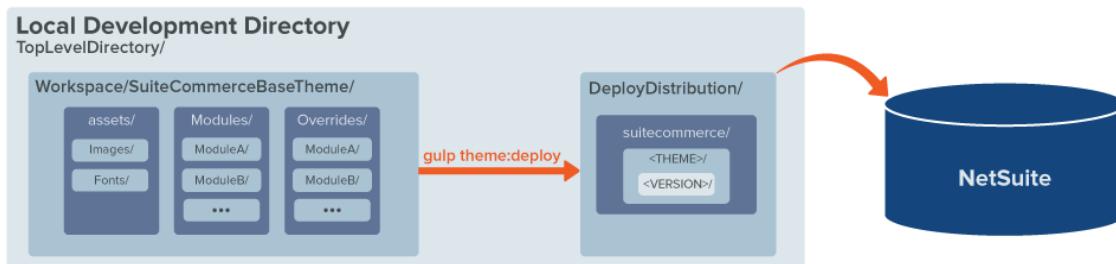
Note: Your extension overrides always deploy as part of a custom theme. If you activate a custom theme but fail to activate any extension for which the theme includes overrides, the application will ignore the overrides and your site will function normally.

Example

You download the source files for the active theme (**SuiteCommerce Base Theme**) and the active extension (**PublishedExtension1**). You follow best practices and procedures to customize your theme files and override your extension files. After testing on a local server, you decide to deploy your customizations.

You run the `gulp theme:deploy` command, and the theme development tools update the manifest file with any overrides, validate your customizations, and copy the contents of your theme directory to your DeployDistribution directory. The development tools then upload your development files to a location in your NetSuite File Cabinet. These files are not compiled into any usable runtime files, but they are now available for activation on a site and domain.

When you run the `gulp theme:deploy` command, the development tools prompt you for a new theme name, which you call **MyCustomTheme1**. Finally, the development tools rename your Workspace/SuiteCommerceBaseTheme directory to Workspace/MyCustomTheme1 and update the manifest and configuration file with the new theme name. This prepares the local environment to function with the new name.



Extension Developer Tools

 **Applies to:** SuiteCommerce Web Stores

The extension developer tools are required for all extension developers. This applies to all SuiteCommerce and any SuiteCommerce Advanced implementations using Aconcagua release and later.

After you have successfully installed the extension developer tools, you are ready to develop, test, and deploy extensions. Extension development requires using the Extensibility API.



Important: These procedures assume that you have successfully set up your developer environment to use the extension developer tools. See [Developer Environment](#) for details.

The Extension developer tools let you do the following:

- [Create Extension Files](#)
- [Fetch Active Theme and Extension Files](#)
- [Test an Extension on a Local Server](#)
- [Deploy an Extension to NetSuite](#)

Before You Begin

Be aware of the following important information:

- The `gulp extension:fetch` command only fetches the following:
 - HTML and Sass files associated with the currently active theme for the specified domain.
 - Extensions development files for custom extensions currently active for the specified domain. You cannot download files for published extensions.
- Login parameters, including your NetSuite email, account, role, and domain information, are stored in the `.nsdeploy` file when you create an extension. To reset your login and deployment information, delete the `.nsdeploy` file. This file is located here:
`<LOCAL_SOURCEFILES_ROOT>/gulp/config/.nsdeploy`
- Extension information, including the extension name, fantasy name, version, and description, are stored in the extension's `Manifest.json` file. The manifest file is located here:
`<LOCAL_SOURCEFILES_ROOT>/Workspace/<EXTENSION_DIRECTORY>/manifest.json`

Create Extension Files

The following section explains how to use the extension developer tools to create initial files to use as a baseline for extension development. This is a critical first step when building a new extension, as it provides you with a starting point in your extension workspace.

Create a Baseline Extension

A **Baseline Extension** is a set of files, installed locally in your extension development directory. You use these files as a basis to build any extension for your SuiteCommerce site.

When you run the `gulp extension:create` command, the developer tools create this baseline extension in a `Workspace` directory within your top-level extension development directory. The

baseline extension starts out with one module that you design, based on your needs. It can include any combination of the following file types within one new module:

- Templates
- Sass
- Configuration
- JavaScript
- SuiteScript

To create a baseline extension:

1. Open a command line or terminal.
2. Access the top-level extension development directory you created when you installed the extension developer tools.
3. Enter the following command:
`gulp extension:create`
4. When prompted, enter the following information:



Important: Unless otherwise directed, create all entries using only alphanumeric characters without spaces.

Pressing **enter** results in the default value.

- Set Your Extension Fantasy Name – Provide a name for your extension as you want it to appear in the NetSuite user interface. This can contain special characters and spaces. For example: **My Cool Extension!**
- Set Your Extension Name – Provide a name for your extension's files and folders within your development environment.
- Set the Vendor Name – Enter your business, vendor, or partner name.
- Assign a Version Number – Add a version number for your extension.



Important: To take advantage of extension update requirements, use semantic versioning (SemVer) notation. For more information, see <https://semver.org/>.

- Set a Description for Your Extension – Provide some text to describe your extension. This string can contain spaces and special characters.
 - This Extension Supports – Select one of the following options:
 - SuiteCommerce Online – To create an extension for SuiteCommerce or SuiteCommerce Advanced.
 - SuiteCommerce InStore – To create an extension for SuiteCommerce InStore.
- An asterisk (*) identifies the selected option. Use arrow keys to scroll. Press the spacebar to select an option, press **<a>** key to select all, or **<i>** key to invert your selections.
- Set the Initial Module Name – Name the module to be created as part of the baseline extension. Each extension needs at least one module to contain your extension files.
 - This Extension Will Be Applied To – Select one or more applications that your extension requires. Options include: Shopping, My Account, and Checkout.
 - For This Extension You Will Be Using – Select one or more file types that your extension should include. Your options are: JavaScript, SuiteScript, Templates, Sass, and Configuration files. The developer tools create a baseline example of these files to get you started.

- Select extension to add module – Select the name of the extension to contain this module. This option only appears if your Workspace directory contains more than one extension.

After following these prompts, you can optionally build on this baseline by adding additional modules. Consider the following topics:

- [Create Additional Modules for an Extension](#)
- [Create Custom Content Types for an Extension](#)

When you have completed building baseline extension files in your developer environment, you are ready to develop your extension. See [Extensions](#) for details.

What Happens When You Create Extension files?

When you run the `gulp extension:create` command, the extension developer tools:

- Create the Workspace directory in your top-level development directory, unless it already exists.
- Create a subdirectory to house all of your extension's code. This is where you develop your extension.
- Create a manifest.json file. This file includes all the information required to compile resources for your extension.

Example

You run the `gulp extension:create` command and build a baseline extension, **MyCoolExtension** with a module called **MyCoolModule**. You set this extension to include all available files.

In this example, your Workspace directory structure should look similar to following:

```
Workspace/
  MyCoolExtension/
    assets/
      fonts/
      img/
      services/
    Modules/
      MyCoolModule/
        Configuration/
        JavaScript/
        Sass/
        SuiteScript/
        Templates/
    manifest.json
```

Create Additional Modules for an Extension

The `gulp extension:create` command builds an extension with one baseline module. However, if your extension requires more than one module, the developer tools let you create more using the `--module` attribute.

To add an additional module to your baseline extension:

1. Create a baseline extension, if you have not done so already. See [Create a Baseline Extension](#) for details.
2. Open a command line or terminal.

3. Access the top-level extension development directory you created when you installed the extension developer tools.

4. Enter the following command:

```
gulp extension:create-module
```

5. When prompted, enter the following information:



Important: Unless otherwise directed, create all entries using only alphanumeric characters without spaces.

- Set the Module Name – Name the module to be created as part of the baseline extension. Use only alphanumeric characters without spaces.
 - For This Module You Will Be Using – Select one or more file types your extension requires. Your options are: JavaScript, SuiteScript, Templates, Sass, and Configuration files.
- An asterisk (*) identifies a selected option. Press the spacebar to select. Use arrow keys to scroll. Press the A key to select all. Press the I key to invert your selections.

6. Repeat these steps to add any additional modules as needed.



Important: NetSuite imposes certain record limits to control the consumption of web services. To avoid errors related to record limits when deploying themes and extensions, limit the number of files and folders at the same level to 100 when developing themes and extensions. Some options include introducing images across multiple folders and placing custom modules in a different top-level directory. For more information on record limiting, see *Web Services Governance Overview*.

When you have completed building baseline extension files in your developer environment, you are ready to develop your extension. See [Extensions](#) for details.

Create Custom Content Types for an Extension

When you create your baseline extension files, the developer tools provide a method to build your extension as a CCT. The developer tools provide on-screen instructions to assist you with developing your CCT. This requires:

- Creating a baseline extension
- Adding one or more baseline CCTs
- Customizing your CCT and preparing it for use with Site Management Tools



Note: To implement a CCT using Site Management Tools, you must configure the **CMS Adapter Version** to 3 in the SuiteCommerce Configuration record. This property is located in the **Integrations** tab and **Site Management Tools** subtab.

To add a CCT to your baseline extension:

1. Open a command line or terminal.
2. Access the top-level extension development directory you created when you installed the extension developer tools.
3. Create a baseline extension, if you have not done so already. See [Create a Baseline Extension](#) for details.
4. Enter the following command:

```
gulp extension:create-cct
```

5. When prompted, enter the following information:



Important: Unless otherwise directed, create all entries using only alphanumeric characters without spaces.

- Set the Label of the CCT – Name the CCT to be added to your baseline extension. This can contain alphanumeric and special characters.
 - Set the Name of the CCT – Provide a name for your CCT as it appears in your development files and folders. Use only alphanumeric characters without spaces.
 - Set a Description for your CCT – Provide some text to describe your CCT.
 - For This CCT You Will Be Using – Select one or more file types your CCT requires (Templates, Sass, Configuration, JavaScript, SuiteScript). An asterisk (*) identifies a selected application.
 - Select Extension to Add CCT – Select the name of the extension to contain this module. This option only appears if your Workspace directory contains more than one extension.
6. Repeat these steps to add any additional CCTs as needed.

To Build a CCT:

After you have successfully created your baseline CCT, you must set it up for use with Site Management Tools and customize your code to meet your needs.

1. Open the extension's manifest.json file. This can be found in Workspace/<EXTENSION_DIRECTORY>/.

This file lists the following as part of the manifest metadata. You need this information when setting up your records in NetSuite.

- **icon** – Equals a default icon that is visible in SMT Admin. As a default, the developer tools provide one to help you get started.
- **settings_record** – Equals the **ID** of the custom record you associate with this CCT.
- **registercct_id** – Equals the **Name** field of the CMS Content Type Record for your CCT. This is also the value of the **id** property within the **registerCustomContentType()** method of your CCT module's entry point JavaScript file.

2. Prepare your CCT for use with SMT.

Follow the instructions listed in the help topic [Custom Content Type](#). Build your custom record and your CMS Content Type Record using the parameters explained in Step 1.

3. Develop your CCT module.

Follow the instructions listed in the help topic [Create a Custom Content Type](#). Ensure that your entry point JavaScript file incorporates your CMS Content Type record name, as described in Step 1.

4. Deploy and activate your extension. Then log into SMT to test confirm that your CCT was added correctly. See the help topic [Site Management Tools](#) for details on using SMT.

- See [Deploy an Extension to NetSuite](#) for instructions on deploying your extension.
- See the help topic [Manage Themes and Extensions](#) for instructions on activating your extension.
- [Extensibility API](#)

Fetch Active Theme and Extension Files

Before you can deploy an extension or test it on a local server, you must first download files for the currently active theme by running the following command:

```
gulp extension:fetch
```

This command:

- Downloads HTML and Sass files for the currently active theme. Do not modify these files. The Sass and HTML files downloaded using this command are for reference only.
- Downloads files for any custom extensions that are currently activated.

The developer tools require theme files (HTML and Sass) to compile a local distribution for testing an extension on a local server. The developer tools also use these files to run a test compilation before deploying. When you fetch the active theme, you provide the necessary files in your local extension workspace. Later, when you test locally or deploy your extension, the developer tools can compile the code without error.

If you want to skip this step, you can run the `gulp extension:deploy --skip-compilation` command to deploy your extension files without running the compilation. For more Gulp.js commands used with extensions, see [Extension Developer Gulp Commands](#).

The `gulp extension:fetch` command also fetches files for any custom extensions. When you fetch files for an active extension, you download these development files to your local extension workspace. You can now customize these files further development and testing.

 **Warning:** When you fetch custom extension files, the developer tools download extension files that are activated for the specified domain. This action overwrites any extension files currently stored in your local workspace. This can result in loss of development files if you are not careful. To ensure that you do not lose any work under development, do not run the `gulp extension:fetch` command until you have deployed your extension work to NetSuite.

 **Note:** This command does not download files for published extensions. A published extension is an extension installed into your account as part of a published SuiteApp. You cannot edit published content due to copyright restrictions.

Fetch the active files:

1. In your local developer environment, open a command line or terminal and access the top-level extension development directory.
2. Run the following command:
`gulp extension:fetch`
3. When prompted, enter the following information:
 - NetSuite Email – Enter the email address associated with your NetSuite account.
 - NetSuite Password – Enter your account password.
 - Choose Your Target Account and Role – Select the NetSuite account where SuiteCommerce application is installed. You must specify the correct role with required permissions for connecting to NetSuite.

 **Important:** You must log in using the SCDeployer role to access your NetSuite account. Failure to use this role may result in an error. See [Developer Tools Roles and Permissions](#) for instructions on setting up this role.

- Choose Your Website – Select your website with the domain you want to access.
- Choose Your Domain – Select your domain with the active theme or extensions you want to download.
- Extensions to Fetch – If you have any active extensions for the domain selected, you can download the working files of any selected extensions. This is optional. Content downloads to your extension's Extras folder.

The `gulp extension:fetch` command creates a theme subdirectory in your Workspace/Extras directory. These files are for use by the developer tools only. Do not add, edit, delete, or move any files in this location.

You are now ready to perform either of the following tasks:

- [Test an Extension on a Local Server](#)
- [Deploy an Extension to NetSuite](#)

Test an Extension on a Local Server

You can test your extension customizations on a local server before deploying to NetSuite. The local server is installed as part of the Node.js installation and uses the Express web framework.

`gulp extension:local`

This command compiles all source files into combined files within a LocalDistribution/tmp directory.

When the server starts, Gulp.js initializes watch tasks that listen for changes to files in the JavaScript, Templates, or Sass directories. When you save changes to a file, gulp automatically recompiles the source files and updates the LocalDistribution directory. Gulp also outputs a message to the console when an update occurs.



Important: Typically, you test your code locally before deploying to a live site in a NetSuite account. However, if you are developing an extension that includes SuiteScript or configuration (JSON) files, you must deploy your files to your account and activate the extension for these changes to accessible by the local server. SuiteScript includes services, which do not exist in your account's backend until you deploy them. Likewise, changes to configuration JSON files do not apply to a domain until deployed. See the help topic [Manage Themes and Extensions](#).



Note: If you modify any manifest files, you must restart your local server to see changes.

To test your extension on a local server:

1. If you have not already done so, fetch the active theme.
This is required before you can test an extension locally. See [Fetch Active Theme and Extension Files](#) for details.
2. Open a command line or terminal and access the top-level development directory. This is the same directory created when you extracted the Extension Developer Tools.
3. Run the following command:

`gulp extension:local`

If this is the first time you are running `gulp extension:local` in this directory, this command creates a subdirectory called LocalDistribution. It then compiles the source files and outputs them to this directory.



Warning: Potential data loss. Besides compiling and deploying your extension to a local server, the `gulp extension:local` command updates the manifest.json file for the extension. This action overwrites any manual changes you made to this file. To preserve any manual changes to your manifest.json file, run the `gulp extension:local --preserve-manifest` command instead. See [Extension Manifest](#) for details on this file.

4. Navigate to the local version of the application using one of the following URLs:
 - **Shopping:** `http://<DOMAIN_NAME>/c.<ACCOUNT_ID>/<SSP_APPLICATION>/shopping-local.ssp`
 - **My Account:** `http://<DOMAIN_NAME>/c.<ACCOUNT_ID>/SSP_APPLICATION/my_account-local.ssp`

- **Checkout:** `http://<DOMAIN_NAME>/c.<ACCOUNT_ID>/SSP_APPLICATION/checkout-local.ssp`

In the above URL patterns, you must replace the following variables with values for your specific environment:

- `DOMAIN_NAME` — replace this value with the domain name configured for your NetSuite website record.

- `ACCOUNT_ID` — replace this value with your NetSuite account ID.

- `SSP_APPLICATION` — replace this value with the URL root that you are accessing.

For SuiteCommerce implementations, this variable should read `scs`. For example:

`http://www.mysite.com/c.123456/scs/shopping-local.ssp`

For SuiteCommerce Advanced (SCA), this variable equals the URL root of the SCA implementation. For example:

`http://www.mysite.com/c.123456/sca-dev-aconcagua/shopping-local.ssp`

The URLs you use should be similar to the following examples:



Note: When accessing the secure checkout domain using HTTPS on the local server, you must use a different URL. See [Secure HTTP \(HTTPS\) with the Local Server](#) for more information.

Deploy an Extension to NetSuite

The term **deploy** refers to what happens when you use Gulp.js to upload source files and any custom changes to a hosted site. To do this, you use the following command:

`gulp extension:deploy`

This command validates your code, copies them into a local DeployDistribution directory, and updates the manifest.json. The developer tools then upload these files to your NetSuite account, making them available for activation.



Note: For more Gulp.js commands used with extensions, see [Extension Developer Gulp Commands](#).

To deploy your extension to NetSuite:

1. If you have not already done so, fetch the active theme.

This is required before you can deploy an extension. See [Fetch Active Theme and Extension Files](#) for details.

2. In your local developer environment, open a command line or terminal and access the top-level development directory.
3. Run the following command:

`gulp extension:deploy`



Warning: Potential data loss. Besides compiling and deploying your extension to NetSuite, the `gulp extension:deploy` command updates the manifest.json file for the extension. This action overwrites any manual changes you made to this file. To preserve any manual changes to your manifest.json file, run the `gulp extension:deploy --preserve-manifest` command instead. See [Extension Manifest](#) for details on this file.

4. When prompted, enter the following information.

Unless otherwise directed, create all entries using only alphanumeric characters without spaces.



Note: These prompts appear the first time you deploy a new extension to NetSuite. Subsequent deployments of the same extension only prompt for your NetSuite password. To reset the login information and enter information again, use the `gulp extension:deploy --to` command to deploy your changes.

- Select Extension – If you have more than one extension in your developer environment, choose the extension you are deploying.
- NetSuite Email – Enter the email address associated with your NetSuite account.
- NetSuite Password – Enter your account password.
- Choose Your Target Account and Role – Select the NetSuite account where SuiteCommerce application is installed. You must specify the correct role with required permissions for connecting to NetSuite.



Important: You must log in using the SCDeployer role to access your NetSuite account. Failure to use this role may result in an error. See [Developer Tools Roles and Permissions](#) for instructions on setting up this role.

- Choose Your Website – Choose your site from the list.
- Vendor – Enter your partner or business vendor name.
- Name – Provide a name for your extension within your development environment.
- Fantasy Name – Provide a name for your extension as you want it to appear in the NetSuite user interface. This can contain special characters and spaces.
- Version – Add a version number for your extension.



Important: To take advantage of extension update requirements, use semantic versioning (SemVer) notation. For more information, see <https://semver.org/>.

- Description – Provide some text to describe your extension. This string can contain special characters and spaces.
- Select Supported Products – Select one of the following options:
 - SuiteCommerce Online – To create an extension for SuiteCommerce or SuiteCommerce Advanced.
 - SuiteCommerce InStore – To create an extension for SuiteCommerce InStore.

An asterisk (*) identifies a selected option. Press the spacebar to select. Use arrow keys to scroll. Press the A key to select all. Press the I key to invert your selections.

5. Activate your new extension for the domain or domains of your choice. See the help topic [Manage Themes and Extensions](#) for instructions.



Important: Any changes to your extension source code do not apply to your site until you activate the extension for a specific domain using the Manage Extensions wizard in NetSuite. The deploy process includes a notation reminding you of the domain and theme name to set during this process.

What Happens When You Deploy an Extension?

The deployment process is specific to uploading custom extensions to a location in your NetSuite File Cabinet. This process does not compile any files or associate any files with a site or domain. To do that, you must activate a theme and include your extensions as necessary.

During the deployment process:

- The development tools copy all of your custom extension development files (modules and assets) into the DeployDistribution folder in your top-level development directory. If this directory does not exist, the development tools create it.
- The extension development tools validate your code.
- The extension development tools upload these files to your NetSuite file cabinet (maintaining the same organizational structure). These are simply your development files. Nothing is compiled or activated at this time.

Developer Tools Reference

 **Applies to:** SuiteCommerce Web Stores | Aconcagua

This topic includes the reference materials for using the developer tools. The Gulp Command Reference provides details on each command using the theme and extension developer tools. Some of these topics provide important details for testing locally or deploying to NetSuite.

Developer Tools Roles and Permissions

 **Applies to:** SuiteCommerce Web Stores | SuiteCommerce InStore

Two-factor authentication (2FA) is mandatory on all NetSuite accounts using NetSuite 2018.2 and later. However, the SuiteCommerce developer tools do not currently support 2FA.

When using the SuiteCommerce developer tools to access a NetSuite account provisioned with 2FA, you must sign in with the **SCDeployer** role. This role includes all permissions required to deploy customizations but does not require two-factor authentication.

This section includes the following topics:

- [Confirm That You Have the SCDeployer Role](#) – Explains how to determine if you already have the SCDeployer role.
- [Prepare the SCDeployer Role](#) – Explains how to prepare the SCDeployer role for use.
- [Set Up the SCDeployer Role Manually](#) – This topic explains how to set up the SCDeployer role manually. This applies to implementations of SuiteCommerce Advanced (Aconcagua and earlier) only.



Important: Because the developer tools do not accommodate 2FA, any attempt to deploy or fetch themes and extensions or deploy core SCA modules without properly setting up and using the SCDeployer role results in errors. See [Error Messages When Running the Developer Tools](#) for troubleshooting these errors.

Confirm That You Have the SCDeployer Role

If you have the SuiteCommerce Extension Management SuiteApp (2018.2 release or later) installed in your NetSuite account, you should have the SCDeployer role. However, you can check your NetSuite account for confirmation.

To determine if you have the SCDeployer role in your account:

1. In NetSuite, go to Setup > Users/Roles > Manage Roles.
2. Scroll to see if the SCDeployer role exists in the list.
3. If the role exists, see [Assign the SCDeployer Role to an Employee Record](#).
If the role does not exist, see [Set Up the SCDeployer Role Manually](#) to create it.

Prepare the SCDeployer Role

After you have confirmed that the SCDeployer role exists in your account, perform the following steps to prepare it for use in the developer tools:

- [Assign the SCDeployer Role to an Employee Record](#)
- [Prepare Developer Tools](#)

Assign the SCDeployer Role to an Employee Record

Before a developer can use the SuiteCommerce developer tools, you must assign the SCDeployer role to the Employee record for each employee that requires developer access to NetSuite.

Assign the SCDeployer role in **Role** field of the Employee record (**Access** tab, **Role** subtab). See the help topic [Assigning Roles to an Employee](#) more information.

Prepare Developer Tools

The developer tools initially prompt you for login credentials when you initially use the following commands to access NetSuite:

- `gulp theme:fetch`
- `gulp theme:deploy`
- `gulp extension:fetch`
- `gulp extension:deploy`
- `gulp deploy`

The developer tools store the login credentials (including the role) in the `.nsdeploy` file. If you used the SuiteCommerce developer tools prior to setting up the SCDeployer role, you might need to reset the login credentials.

To reset the developer tools login credentials, add the `--to` parameter to the applicable gulp command within the developer tools. This prompts the developer to recreate the login credentials, overwriting the `.nsdeploy` file.

When prompted to choose a role using any of the aforementioned commands, select the SCDeployer role to gain access to NetSuite accounts without requiring two-factor authentication.

Set Up the SCDeployer Role Manually

 **Applies to:** SuiteCommerce Advanced

If you are implementing the Aconcagua release of SuiteCommerce Advanced and earlier, you must create the SCDeployer role manually.

If you do not have the SCDeployer role in your NetSuite account, perform the following steps:

- [Create the SCDeployer Role](#)
- [Edit Script Deployment Records](#)

Create the SCDeployer Role

To create a SCDeployer role:

1. In NetSuite, go to Setup > Users/Roles > Manage Roles > New.
2. Fill in the Role record as described in the [Customizing or Creating NetSuite Roles](#) topic.
Include the following changes:
 - In the **Name** field, title the role **SCDeployer**.
 - In the **ID** field, enter **_sc_developer**.
 - In the **Two-Factor Authentication Required** field, select **Not required**.
 - On the **Permission** tab, add the following permissions:

Subtab	Role	Level
Lists	Documents and Files	Full
	Custom Record Entries	Full
	Website (External) Publisher	Full
Setup	Allow JS/HTML Uploads	Full
	Custom Lists	Full
	Set Up Web Site	Full
	SuiteScript	View
	Web Services	Full

3. Save the Role record.
4. Assign the SCDeployer role to the Employee record for each person requiring developer access to NetSuite. See [Assign the SCDeployer Role to an Employee Record](#) for more information.
5. If necessary, prepare the developer tools to allow NetSuite access using the SCDeployer Role. See [Prepare Developer Tools](#) for more information.

Edit Script Deployment Records

As part of the manual role set up, you must also manually edit your account's Script Deployment records.

To edit Script Deployment records:

1. In NetSuite, go to Customization > Scripting > Script Deployments.
2. Set the following filters:
 - **Type:** RESTlet
 - **API Version:** 1.0
3. Add the **SCDeployer** role to **Roles** list on the **Audience** tab for each of the following Script Deployment records, depending on your implementation:

SuiteCommerce, SuiteCommerce Advanced (Aconcagua and Later), SCIS (2018.2 and later):

Edit each of the following Script Deployment records.

- customdeploy_ext_mech_extension_service
- customdeploy_ext_mech_file_service
- customdeploy_ext_mech_skin_service
- customdeploy_ext_mech_web_service_rest

SuiteCommerce Advanced and Site Builder Extensions (Kilimanjaro release and Earlier):

Edit the script associated with your implementation.

- customdeploy_ns_sca_deployer_kilimanjaro
- customdeploy_ns_sca_deployer_elbrus
- customdeploy_ns_sca_deployer_mont blanc
- customdeploy_ns_sca_deployer_vinson

- customdeploy_ns_sca_deployer_denali
- customdeploy_ns_sbe_deployer_elbrus
- customdeploy_ns_sbe_deployer_kilimanjaro
- customdeploy_ns_sbe_deployer_mont_blanc
- customdeploy_ns_sbe_deployer_vinson
- customdeploy_ns_sbep_deployer_denali
- customdeploy_ns_sbep_deployer_mont_blanc
- customdeploy_ns_sbep_deployer_vinson
- customdeploy_ns_sbep_deployer_elbrus
- customdeploy_ns_sbep_deployer_kilimanjaro

4. Save the Script Deployment record.

Gulp Command Reference for Themes and Extensions

The following tables lists Gulp.js commands required when using the theme or extension developer tools with your SuiteCommerce site:

- [Theme Developer Gulp Commands](#)
- [Extension Developer Gulp Commands](#)

Note: For a list of commands used with the SuiteCommerce Advanced (SCA) developer tools, see [Theme Developer Gulp Commands](#).

Theme Developer Gulp Commands

Command	Description
gulp theme:fetch	This command downloads the active theme and extension files (Sass, HTML, and other assets) for the specified domain. You must have already activated a theme using the Manage Extensions wizard in NetSuite for this command to run. This command places these files in the top-level directory's Theme directory. If this is the first time running this command, the development tools create subdirectories for these files.
gulp theme:fetch --to	This command fetches a theme and extensions from NetSuite as usual, but it prompts for login credentials, ignoring the .nsdeploy file.
gulp theme:local	This command compiles all Sass and HTML template files for a theme into a functional application. This command also updates the theme manifest.json. After compilation, this command starts a local server. This server watches for changes to Sass and HTML template files. After the server starts, any changes you make to your theme files or extension overrides are automatically recompiled and visible in the browser.
gulp theme:local --preserve-manifest	This performs the same action as <code>gulp theme:local</code> , but it does not update the manifest. Use this command to test your theme locally if you have made any manual changes to the manifest.json file for the theme.

Command	Description
gulp theme:deploy	<p>This command compiles Sass, HTML, and asset files into a DeployDistribution folder. This command also updates the theme manifest.json.</p> <p>If you are deploying changes to a published theme, the development tools prompt you for a Vendor Name, Theme Name, Theme Version, Description, and Application when you initially run this command. This command forces you to name a new theme.</p> <p>If you are deploying customizations to a custom theme, this command does not prompt you for this information and gives you the option to create a new theme or update the existing theme.</p> <p>The development tools then create a folder for the theme in the NetSuite file cabinet and deploy the customized theme's code.</p> <p>In addition to compiling the application, this command creates the .nsdeploy file, if it does not already exist.</p>
gulp theme:deploy --preserve-manifest	<p>This performs the same action as <code>gulp theme:deploy</code>, but it does not update the manifest. Use this command to deploy your theme if you have made any manual changes to the manifest.json file for the theme.</p>
gulp theme:deploy --advanced	<p>This command deploys as an update of the current custom theme, but resets the prompts regarding the Vendor Name, Theme Name, Theme Version, Description, and Application. This command also rewrites this information in the theme's manifest.json file.</p> <p>This command only takes effect on custom themes.</p>
gulp theme:deploy --create	<p>This command creates a new theme instead of updating the existing theme.</p>
gulp theme:deploy --skip-compilation	<p>This command deploys the current contents of the DeployDistribution folder without compiling the application. Although the developer tools do not deploy a compiled set of files, the default action is to compile files as a final test of your code. Using this command skips this check.</p>
gulp theme:deploy --source templates	<p>This command only deploys the HTML template files.</p>
gulp theme:deploy --source sass	<p>This command only deploys Sass files.</p>
gulp theme:deploy --source assets	<p>This command only deploys theme asset files.</p>
gulp theme:deploy --source skins	<p>This command only deploys theme skin preset files.</p>
gulp extension:deploy --source <multiple>	<p>Separate multiple source deployments by commas to deploy more than one type of source code. For example, <code>gulp extension:deploy --source templates, sass</code></p>
gulp theme:deploy --to	<p>This command deploys to NetSuite as usual, but resets the login credentials, rewriting the .nsdeploy file.</p>
gulp theme:local styleguide	<p>This command compiles your Sass, parses all KSS blocks declared in the Sass files, and creates a style guide accessible in your localhost (localhost:3000/). See Style Guide for more information.</p>
gulp theme:update-manifest	<p>This command updates the theme's manifest.json file without requiring a deployment.</p>
gulp validate	<p>This command validates the theme's manifest.json file and confirms that the file does not list any files that do not exist in the theme folder.</p>

Command	Description
gulp clear	This command removes the DeployDistribution and LocalDistribution directories and the .nsdeploy file.
gulp	This command displays a list of all gulp commands.

Extension Developer Gulp Commands

Command	Description
gulp extension:create	This command creates an example extension (with one example module) to use as a baseline for development. This command creates all necessary folders to store and maintain your customizations within your top-level extension development directory.
gulp extension:create-module	This command creates an additional module within your extension. This command prompts you for information about the module you want to create and the extension within which you want to create it.
gulp extension:fetch	<p>This command downloads the active theme and compiles all theme resources (Sass, HTML, and other assets). This command places theme files in the top-level extension development directory's Extras folder. If this is the first time running this command, the development tools create subdirectories for these files.</p> <div style="border: 1px solid #0070C0; padding: 10px; margin-top: 10px;"> Note: Any downloaded theme files are only provided for testing your extensions locally. You do not customize any theme files in your top-level extension development directory. </div> <p>If you also choose to continue development on a previously deployed, custom extension, this command also downloads these files, placing them in your Workspace/<EXTENSION_FOLDER>. The extensions must be active using the Manage Extensions wizard in NetSuite.</p> <div style="border: 1px solid #0070C0; padding: 10px; margin-top: 10px;"> Note: This command does not download published extensions. You cannot customize published content. </div>
gulp extension:fetch <arg>	<p>This command downloads the files of a specific extension, where <arg> is the name of the extension. This can be helpful if you want to use a previously deployed extension as a baseline for a new one. You can fetch multiple extensions, separated by a comma.</p> <p>For example: <code>gulp extension:fetch --fetch Badges,CartExtension</code>.</p> <div style="border: 1px solid #0070C0; padding: 10px; margin-top: 10px;"> Note: Extensions must be active to download code or to view them on a local server. </div>
gulp extension:local	<p>This command compiles your custom extension files into a functional application and places them in a LocalDistribution folder.</p> <p>After compilation, this command starts a local server. This server watches for changes to any extension files. After the server starts, any changes you make to your extension files are automatically recompiled and visible in the browser.</p>
gulp extension:local --preserve-manifest	This performs the same action as <code>gulp extension:local</code> , but it does not update the manifest. Use this command to test your extension locally if you have made any manual changes to the manifest.json file for the extension.

Command	Description
gulp extension:deploy	<p>This command compiles your custom extension files into a functional application and places them into a DeployDistribution folder.</p> <p>If you have more than one extension in your top-level development directory, this command prompts you to declare which extension to deploy.</p> <p>If you have never deployed the extension, this command prompts you for information about the extension.</p> <p>The development tools then create a folder for the extension in the NetSuite file cabinet and deploy the customized extension code.</p> <p>In addition to compiling the application, this command creates the .nsdeploy file, if it does not already exist.</p>
gulp extension:deploy --preserve-manifest	<p>This performs the same action as <code>gulp extension:deploy</code>, but it does not update the manifest. Use this command to deploy your extension if you have made any manual changes to the manifest.json file for the extension.</p>
gulp extension:deploy --advanced	<p>This command deploys an update of the extension, but resets the prompts regarding the Vendor Name, Extension Name, Version, Description, Application, etc. This command also rewrites this information in the extension manifest.json file.</p>
gulp extension:deploy --skip-compilation	<p>This command deploys the current contents of the DeployDistribution folder without compiling the application. Although the developer tools do not deploy a compiled set of files, the default action is to compile files as a final test of your code. Using this command skips this check.</p>
gulp extension:deploy --source configuration	<p>This command only compiles and deploys configuration JSON files.</p>
gulp extension:deploy --source javascript	<p>This command only compiles and deploys JavaScript files.</p>
gulp extension:deploy --source ssp-libraries	<p>This command only compiles and deploys ssp-libraries.</p>
gulp extension:deploy --source services	<p>This command only compiles and deploys services.</p>
gulp extension:deploy --source sass	<p>This command only compiles and deploys Sass files.</p>
gulp extension:deploy --source templates	<p>This command only compiles and deploys HTML files.</p>
gulp extension:deploy --source assets	<p>This command only compiles and deploys assets.</p>
gulp extension:deploy --source <multiple>	<p>Separate multiple source deployments by commas to deploy more than one type of source code. For example, <code>gulp extension:deploy --source templates,sass</code></p>
gulp extension:update-manifest	<p>This command updates the extension's manifest.json file without requiring a deployment.</p>
gulp validate	<p>This command validates the theme's manifest.json file and confirms that the file does not list any files that do not exist in the theme folder.</p>
gulp clear	<p>This command removes the DeployDistribution and LocalDistribution directories and the .nsdeploy file.</p>
gulp	<p>This command displays a list of all gulp commands.</p>

Theme Development Files and Folders

This section describes the various files and folders included in your top-level theme development directory. You create this when you extract the zip file containing the theme developer tools.



Note: Some of the files and folders listed herein do not appear until you run one or more Gulp.js commands. The following sections point this out where applicable.

The Theme Development Directory

The top-level theme development directory contains the following files/folders. Some of the files/folders listed below do not appear until after you run your customizations to a local server or deploy them to NetSuite. You name this top-level directory when you extract the theme development tools.

File/Folder	Description
DeployDistribution/	Created when you run the <code>gulp theme:deploy</code> command, this directory contains all of the files associated with the compiled application. After compilation, Gulp.js deploys the contents of this directory to your NetSuite file cabinet. Do not manually edit the files in this directory.
gulp/	This directory contains all of the files required by Gulp.js. Do not manually edit the files in this directory.
LocalDistribution/	Created when you run the <code>gulp theme:local</code> command, this directory contains all of the files associated with the compiled application used by the local server. When you run <code>gulp theme:local</code> , Gulp.js deploys the contents of this directory to the local Node.js server. Do not manually edit the files in this directory.
node_modules	Created when you run the <code>npm install</code> command, this directory stores the dependencies and other files required by the development tools. Do not manually edit this file.
ns_npm_repository	Do not manually edit this file.
Workspace/	Created when you initially run the <code>gulp theme:fetch</code> command, this directory maintains all downloaded and customized theme and extension HTML, Sass, and asset files. This is the directory where you maintain all of your theme and extension customizations. See The Theme Workspace for detailed information on the contents of this directory.
.jshintrc	Do not manually edit this file.
distro.json	Do not manually edit this file.
gulpfile.js	This file contains all the JavaScript code necessary to run Gulp.js. Do not manually edit this file.
javascript-libs.js	Do not manually edit this file.
package.json	This file maintains dependencies required to operate the theme development tools. Do not manually edit this file.
version.txt	This file maintains versioning information for SuiteCommerce. Do not manually edit this file.

The Theme Workspace

The Workspace directory is the location within your top-level theme development directory where you create themes. This directory contains a subdirectory for the downloaded theme and an Extras directory to maintain all extension-related source files. When you run the `gulp theme:fetch` command, the development tools delete all Workspace directory subfolders and download the active theme and extension source files, building a new Workspace environment each time.

```
<TopLevelDevelopmentDirectory>
  Workspace/
    Extras/
      <THEME_DIRECTORY>/
```

The Theme Directory

When you access your theme development directory and run the `gulp theme:fetch` command, the development tools create a subdirectory to store the source files for the active theme. The name of this directory matches the name of the theme that is active when you run the `gulp theme:fetch` command. Perform all theme customizations and template overrides here.

The theme directory contains the following files or folders:

File/Folder	Description
assets/	This directory maintains any images or fonts associated with the theme. These assets include fonts, logos, icons, and other images related to your site that are not managed by NetSuite. This is also the location where you save any new assets you introduce as part of your theme customizations or extension overrides.
Modules/	This directory contains individual modules as subdirectories associated with the theme. Each module defines a specific area of functionality (feature or utility) for your site and contains Template and Sass files in respective subdirectories. You customize these Sass and HTML files directly.
Overrides/	<p>This directory contains any HTML and Sass associated with all extensions that were active when you ran the <code>gulp theme:fetch</code> command. This directory is initially empty, but its structure matches that of the files in the Extras directory.</p> <p>If you are customizing HTML and Sass files associated with an extension, you place copies of those files here to maintain your overrides.</p> <p>See Override Active Extension Files for more details.</p>
Skins/	This directory contains any skin preset files for your theme. For more information on skins, see Create Skins for more details.
manifest.json	This file maintains all extensible resources for the theme and declares any overrides for extension customizations. The development tools automatically edit this file to include any necessary overrides when you run the <code>gulp theme:deploy</code> command. For more information on this file, see Theme Manifest .

Example:

Your domain has an active theme, **SuiteCommerceBaseTheme**. You run the `gulp theme:fetch` command and specify your domain. The development tools clear any existing Workspace directory contents and download all theme-related HTML, Sass, and asset files into the SuiteCommerceBaseTheme directory. This is your workspace for creating and editing your theme. If the theme you download includes any previously deployed overrides, the development tools place them in the Overrides directory.

In this example, your theme workspace structure should look similar to following:

```
<TopLevelDevelopmentDirectory>/
  Workspace/
    SuiteCommerceBaseTheme/
      assets/
        img/
        fonts/
      Modules/
        AddressModule@1.0.0/
          Sass/
          Templates/
      Overrides/
      manifest.json
```

The Extras Directory (Theme Development)

When you access your theme development directory, the/Workspace/Extras/Extensions directory contains subdirectories for each extension active when downloaded using the `gulp theme:fetch` command. Each extension subdirectory contains the following files or folders:



Important: At this time, you can only create your own themes. Therefore, do not manually edit or remove files found in the Extras/Extensions directory. You can only customize extension-related HTML and Sass files using the Override method. See [Override Active Extension Files](#).

File/Folder	Description
assets/	This directory maintains any images or fonts associated with the associated extension. These assets include fonts, logos, icons, and other images related to your site that are not managed by NetSuite.
Modules/	This directory contains a module folder that maintains all HTML templates and Sass associated with the extension. These files are provided here for your reference only.
manifest.json	<p>This file lists all extension-related JavaScript, SSP libraries, configuration, HTML templates, Sass, and assets related to the active extensions downloaded when you ran the <code>gulp theme:fetch</code> command. For more information on this file, see Theme Manifest.</p> <p>Do not manually edit this file.</p>

If your domain does not have any active extensions when you run the `gulp theme:fetch` command, or an active extension does not contain any Sass, HTML, or assets, the development tools do not create a folder for that extension.



Note: The Extras subdirectory also contains an application_manifest.json file. This file confirms that you have a valid SSP Application version that supports the Themes and Extensions. Do not move, delete, or edit this file.

Extension Development Files and Folders

This section describes the various files and folders included in your top-level extensions development directory. You create this when you extract the zip file containing the extension developer tools.



Note: Some of the files and folders listed herein do not appear until you run one or more Gulp.js commands. The following sections point this out where applicable. However, for a full list of Gulp.js commands, see [Gulp Command Reference for Themes and Extensions](#).

The Top-Level Extension Development Directory

Your top-level extensions development directory contains the following files/folders. Some of the files/folders listed below do not appear until after you run your customizations to a local server or deploy them to NetSuite. You name this top-level directory when you extract the theme development tools.

File/Folder	Description
DeployDistribution/	Created the first time you run the <code>gulp extension:deploy</code> command, this directory contains all of the files associated with the compiled application. After compilation, Gulp.js deploys the contents of this directory to your NetSuite file cabinet. Do not manually edit the files in this directory.
gulp/	Created when you extract the extension developer tools, this directory contains all of the files required by Gulp.js. Do not manually edit the files in this directory.
LocalDistribution/	Created the first time you run the <code>gulp extension:local</code> command, this directory contains all of the files associated with the compiled application used by the local server. When you run <code>gulp extension:local</code> , Gulp.js deploys the contents of this directory to the local Node.js server. Do not manually edit the files in this directory.
node_modules	Created when you run the <code>npm install</code> command, this directory stores the dependencies and other files required by the development tools. Do not manually edit this file.
ns_npm_repository	Created when you install the extension developer tools, this folder contains important files for the NPM package manager. Do not manually edit this file.
Workspace/	Created the first time you run the <code>gulp extension:fetch</code> or <code>gulp extension:create</code> command, this directory maintains all of your extension files under development. This directory also includes an Extras/ folder to maintain theme files for local testing. See The Extensions Workspace Directory for detailed information on the contents of this directory.
.jshintrc	Do not manually edit this file.
distro.json	Do not manually edit this file.
gulpfile.js	This file contains all the JavaScript code necessary to run Gulp.js. Do not manually edit this file.
javascript-libs.js	Do not manually edit this file.
package.json	This file maintains dependencies required to operate the theme development tools. Do not manually edit this file.
version.txt	This file maintains versioning information for SuiteCommerce. Do not manually edit this file.

The Extensions Workspace Directory

The Workspace directory resides in your top-level extensions development directory. This directory contains a subdirectory for each extension you are creating plus an Extras directory to store the active theme source files.

```
<Top-LevelDevelopmentDirectory>/
  Workspace/
    Extras/
    <EXTENSION_DIRECTORY>/
```

The Extension Directory

When you run the `gulp extension:create` command, the developer tools create the Workspace directory (unless it already exists) and places a basic extension with source files for you to begin creating your own extension.

Each extension directory contains the following files or folders:

File/Folder	Description
assets/	This directory maintains any images or fonts associated with the extension. These assets include fonts, logos, icons, and other images related to your site that are not managed by NetSuite.
Modules/	This directory contains individual modules as subdirectories associated with the extension. Each module defines a specific area of functionality (feature or utility) for your site and contains the JavaScript, SuiteScript, Configuration, Templates, and Sass for your extension.
manifest.json	This file maintains all extensible resources for the extension. The development tools automatically edit this file to include any necessary overrides when you run the <code>gulp extension:deploy</code> command. For more information on this file, see Extension Manifest .

Example

You run the `gulp extension:create` command from your top-level extension development directory. During this task, you name your extension **MyCoolExtension** and choose to create all options for extension files (templates, sass, configuration, etc.). You also choose not to create a CCT.

After this command is complete, your Workspace directory contains the following:

```
<TopLevelDevelopmentDirectory>/
  Workspace/
    MyCoolExtension/
      assets/
        fonts/
        img/
        services/
        MyCoolModule.Service.ss
      Modules/
        MyCoolModule/
          Configuration/
          JavaScript/
          Sass/
          SuiteScript/
          Templates/
        manifest.json
```

The MyCoolModule directory contains an example file for each files type. You use these files as an example to build your new extension. For example, the JavaScript folder contains JavaScript collection, model, router, view, and entry point files.

The Extras Directory

When you run the `gulp extension:fetch` command, the developer tools create the Workspace directory (unless it already exists) and downloads the source files for the active theme. These files are provided for reference during local testing only. Do not move, delete, add, or edit the files in the Extras subdirectory.

The Extras subdirectory contains the following files or folders:

File/Folder	Description
assets/	This directory maintains any images or fonts associated with the active theme. These assets include fonts, logos, icons, and other images related to your site that are not managed by NetSuite.
Modules/	This directory contains a module folder that maintains all HTML templates and Sass associated with the theme. These files are provided here for your reference only.
manifest.json	This file lists all extension-related HTML templates, Sass, and assets related to the active theme downloaded when you ran the <code>gulp extension:fetch</code> command. Do not manually edit this file.

Note: The Extras subdirectory also contains an application_manifest.json file. This file confirms that you have a valid SSP Application version that supports the Themes and Extensions. Do not move, delete, or edit this file.

Example:

You run the `gulp extension:fetch` command.

Your domain has an active theme, **ActiveTheme1**, and an active extension, **MyCoolExtension**. You run the `gulp theme:fetch` command and specify your domain. The extension developer tools download all theme-related HTML, Sass, and asset files into the Extras/ActiveTheme1/ directory.

In this example, your Workspace directory structure should look similar to following:

```
<TopLevelDevelopmentDirectory>/
  Workspace/
    Extras/
      ActiveTheme1/
        assets/
        Modules/
        Overrides/
        manifest.json
      MyCoolExtension/
```

Mixed Domains in a Local Server

Applies to: SuiteCommerce Web Stores

When redirecting between domains while testing on your local server, the default behavior is for the application to load the **production** version of the initial page in the new domain.

For example, suppose you start in the **local** version of the Shopping domain using the shopping-local.ssp URL. On login, you are redirected to the **production** version of the Checkout domain at the my-account.ssp URL. To return to the local version of that domain you must manually edit the URL.

Change: https://checkout.netsuite.com/cxxxxxxxx/sca-dev-denali/my_account.ssp?n=3

To: https://checkout.netsuite.com/cxxxxxxxx/sca-dev-denali/my_account-local.ssp?n=3

Also, in some browsers, when you manually update this URL you may encounter blank page loads with console errors similar to the following:

Mixed Content: The page at 'https://checkout.netsuite.com/cxxxxxxxx/sca-dev-denali/my_account-local.ssp?n=3' was loaded over HTTPS, but requested an insecure stylesheet '<http://localhost:7777/css/myaccount.css>'. This request has been blocked; the content must be served over HTTPS.

For example, Chrome returns this error to protect you from a site that is not secure. If this happens in Chrome, in the address bar you will see a shield. Click on the shield and select **Load unsafe scripts**. The page then loads normally.



Note: You can also launch the Chrome browser with the flag `--allow-running-insecure-content` or you can install the certificates in your local web server.

Secure HTTP (HTTPS) with the Local Server

Applies to: SuiteCommerce Web Stores

The **gulp local** command starts two instances of the local server at the following URLs:

- <http://localhost:7777>
- <https://localhost:7778>

The instance running at port 7778 provides a secure domain using HTTPS. This enables you to test the application using secure domains. However, before using secure domains on the local server, you must perform the following:

- Modify the **distro.json**.
- Modify the root URL of the **shopping-local.ssp** file.
- Generate the required SSL certificates and private keys.
- Configure the **KEYPEM** and **CERTPEM** environment variables.
- Install the certificates on your system.

Generate SSL Certificates and Private Keys

To access a secure domain via HTTPS when running the local server, you must use an SSL certificate and a private key. Since the local server is intended for testing and not a production environment, you can create a self-signed certificate locally and do not need to use a third-party certificate provider.

To generate an SSL Certificate and a Private Key

1. Download and Install OpenSSL

See the help topic [Download and Install OpenSSL](#) for more information.

2. Generate an RSA private key.

- a. Run the following command:

```
openssl genrsa -des3 -out ca.key 1024
b. Enter and confirm a password for the certificate.
```

You will use this password in the remaining procedures for creating a certificate and private key.

This command outputs the RSA private key in a file called `ca.key`.

3. Create a new SSL certificate.

a. Run the following command:

```
openssl req -new -sha256 -key ca.key -out ca.csr
```

This command uses the RSA private key created in the previous step.

b. Accept the default value for the `localhost` field. The other fields are not required to create the certificate used by the local server when running HTTPS.

This command outputs the SSL certificate in a file called `ca.csr`.

4. Create a self-signed certificate:

```
openssl x509 -req -days 3600 -in ca.csr -out ca.crt -signkey ca.key
```

If you are prompted to enter a password, use the password you entered when generating the RSA key.

5. Create a server key:

```
openssl genrsa -des3 -out server.key 1024
```

This command outputs the server private key to a file called `server.key`.

6. Create a certificate signing request (CSR):

```
openssl req -new -sha256 -key server.key -out server.csr
```

This command outputs the CSR to a file called `server.csr`.

7. Remove the password from the server certificate.

This step is optional. If you encounter problems with the password, you can remove it from the certificate.

a. Copy the `server.key` file to `server.key.org`.

b. Run the following command to generate a new `server.key` file that has no password:

```
openssl rsa -in server.key.org -out server.key
```

This command creates a new private key called `server.key`. The local server uses this file when creating a secure domain. Therefore, you should move it to a permanent location.

8. Create a self-signed server certificate:

```
openssl x509 -req -sha256 -days 3600 -in server.csr -signkey server.key -out server.crt
```

This command creates a new server certificate called `server.crt`. The local server uses this file when creating a secure domain. Therefore, you should move it to a permanent location.

Configure the KEYPEM and CERTPEM Environment Variables

After generating a server certificate and private key, you must define environment variables that point to these files.

Using the method for setting environment variables for your operating system, create the following:



Note: You must set these environment variables before running the local server.

KEYPEM	<path_to_file>/server.key
CERTPEM	<path_to_file>/server.crt

On Windows, for example, you can set these environment variables as in the following example:

```
set KEYPEM=c:\OpenSSL-Win64\server.key
set CERTPEM=c:\OpenSSL-Win64\server.crt
```

Install the Generated Certificates

After generating the SSL and server certificates, you must enable them to work with your web browser. On Windows, you can use the Certificate Import Wizard.

To install generated certificates:

1. Run the `server.crt` file you generated using OpenSSL.
2. Click **Install Certificate**.
3. Click **Next**.
4. Choose **Place all certificates in the following store**, then click **Browse**.
5. Choose **Trusted Root Certification Authorities**, then click **OK**.
6. Click **Next**.
7. Verify that your settings are correct, then click **Finish**.
8. Click **Yes** to verify that you want to install the certificate on your system.

After installing the server certificate, you should repeat these procedures to install the `ca.crt` file generated in a previous step.

Modify the distro.json File

To access a secure domain on the local sever, you must ensure that the `https` object exists in the `local` object of `taskConfig`. After adding the `https` object, add an entry for the HTTPS port, certificate and key.

Your `distro.json` file should look similar to the following:

```
"tasksConfig": {
  "local": {
    "http": {
      "port": 7777
    },
    "lessSourcemap": false,
    "jsRequire": true,
    "https": {
      "port": 7778,
      "key": "KEYPEM",
      "cert": "CERTPEM"
    }
  }
}
```

```

    }
},
...

```

Note: You must set the **key** and **cert** properties as shown above. The local server uses these values to determine the environment variables used to local the certificate and key required to use HTTPS.

Modify the Root URL of the Shopping SSP Application

To use HTTPS with the local server, you must change the value of the **ROOT** variable in the shopping SSP application.

To modify the root URL of the shopping SSP application:

1. Open the **index-local.ssp** file.

This file is located in <SCA_Source_Root>/Modules/suitecommerce/ShoppingApplication@x.y.z/Internals.

2. Change the value of the **ROOT** variable:

```
var ROOT = 'https://localhost:7778/'
```

3. Compile and deploy the application using the following command:

gulp deploy

Since the above procedure changes a backend file, you must deploy the files to NetSuite. In the process of compiling the application, this command creates the **shopping-local.ssp** file based on the **index-local.ssp** file modified above.

Access the Local Server Using a Secure URL

To access the local server using the local server, you must use the URL of your secure domain.

To access the local server using a secure domain:

1. Run the following command:

gulp local

2. Access the secure domain of the local server using a URL of the following form:

https://checkout.netsuite.com/c.<account_id>/<SSP_application>/shopping-local.ssp

For example, your URL should look similar to the following:

https://checkout.netsuite.com/c.123456/sca-dev-montblanc/shopping-local.ssp

Deploy to a NetSuite Sandbox

Applies to: SuiteCommerce Web Stores

SuiteCommerce also enables you to deploy to a sandbox account. See the help topics [NetSuite Sandbox](#) and [FAQ: NetSuite Sandbox](#) for more information about sandbox accounts.



Note: Before deploying to your sandbox account, you should verify that the SuiteCommerce bundles are installed.

To deploy to a sandbox account:

1. Go to your command line or terminal window.
2. Enter the following command from the top-level directory of the SuiteCommerce source files (the same directory used during the developer tools installation).

gulp deploy

If this is the first time you are running **gulp deploy** in this directory, this command creates a sub directory called DeployDistribution. It then compiles the source files and outputs them to this directory.

3. When prompted, enter the email and password of your sandbox account.



Note: The developer tools do not support emails or passwords containing special characters such as + and %.

4. When prompted, select the sandbox account where SuiteCommerce is installed.
5. When prompted, navigate to Web Site Hosting Files > Live Hosting Files > SSP Applications > NetSuite Inc. - SCA <version> Development.

After you enter your connection setting, the contents of the DeployDistribution folder on your local system are uploaded to the NetSuite file cabinet of your sandbox account. This process may take a few minutes. Wait for the process to complete before proceeding.

Deploy to a Custom SSP Application

Applies to: SuiteCommerce Advanced

During installation of SuiteCommerce Advanced (SCA), two SSP applications are automatically configured in your account and you can easily deploy to the Development SSP application as described in the section [Deploy to NetSuite](#). However, you can also deploy your files to a **custom** SSP application.

Deploying to a custom SSP application consists of the following three steps:

- [Step 1: Create a Custom SSP Application](#)
- [Step 2: Deploy Local Files to Your SSP Application](#)
- [Step 3: Configure the SSP Application](#)



Note: Before deploying to a Custom SSP Application, the SCA bundle must be installed in your account. You can not deploy local files into an account without the bundle installed.

Step 1: Create a Custom SSP Application

The NetSuite SSP application record defines the folder in the NetSuite file cabinet where your website customization assets are stored. If you are unfamiliar with SSP applications in NetSuite, review the following section in the NetSuite Help Center before proceeding:

[SSP Application Overview](#)



Important: If you are updating a site that has been previously developed and deployed to a custom SSP application, you do not need to create a new SSP application. Instead you can connect to the existing SSP application. You need to know the Application Folder and the Application Name of the SSP application record corresponding to the site you are working on. You can find this information on the SSP Application Record.

To create a new SSP application:

1. Go to Setup > SuiteCommerce Advanced > SSP Applications > New.
2. Go to the NetSuite Help topic [Create a SuiteScript 1.0 SSP Application Record](#) and follow the steps outlined there for creating the application.

Creating an SSP application results in a directory structure in the NetSuite File Cabinet with the following path: <**HTML Hosting Root**> /**SSP Applications**/**Application Publisher**/<**Application Name**>. This is where application files will be deployed to using Gulp.

For example, suppose you have a website, MyNetSuiteSite.com, where you want the home page to point to the SCA Shopping application. The configuration of your SSP application could look like this:

- **HTML Hosting Root:** Live Hosting Files
- **Application Publisher:** My Company Name
- **Application Name:** SuiteCommerceDevSite

Step 2: Deploy Local Files to Your SSP Application

After creating your SSP application, use Gulp to deploy the files generated earlier in the Distribution folder to your NetSuite File Cabinet. You will need the following information to complete these steps:

- Your NetSuite username and password
-
- Note:** The developer tools do not support emails or passwords containing special characters such as + and %.
- Details from the SSP application record created in [Step 1: Create a Custom SSP Application](#): HTML Hosting Root, Application Publisher, Application Name

To deploy local files using Gulp:

1. Return to your command line or terminal window.
2. From the root directory of the SCA source files (the same directory used during the developer tools installation), enter the following command.

`gulp deploy`



Note: You should run `gulp deploy` in the directory above the Distribution folder.

3. When prompted, enter your NetSuite email and password.



Note: The developer tools do not support emails or passwords containing special characters such as + and %.

4. When prompted, navigate to the following using the details from your SSP application record:



Tip: These values correspond to the path to your SSP application in the NetSuite File Cabinet: <HTML Hosting Root>: /SSP Applications/<Application Publisher>/<Application Name>

- Hosting Files folder: This is the HTML Hosting Root of your SSP application.
- Application Publisher: This is the Publisher of your SSP application.
- SSP Application: This is the Name of your SSP application.

After all of the connection settings are entered, files from the Distribution folder on your local system are pushed to the NetSuite file cabinet. This process may take a few minutes. Wait for the process to complete before proceeding.



Note: The first time `gulp deploy` is run, the connection settings are saved to a `.nsdeploy` file in the root directory of your source SCA files. On subsequent deployments only the login credentials are required. If you need to change the SSP application you are deploying to, you can manually edit the `.nsdeploy` file with the updated information. For details, see [Changing Your Connection Information](#).

Step 3: Configure the SSP Application

Now that all of the necessary files are available to the SSP application within NetSuite, there are a couple more configuration steps to complete before your site uses the uploaded SCA applications.

1. Go to Setup > SuiteCommerce Advanced > SSP Applications and then click **Edit** next to the SSP application created in Step 1 above.
2. In the Libraries subtab under Scripts, click **Add**.
3. Navigate to the `ssp_libraries.js` file in your SSP application folder and click **Add**.

The `ssp_libraries.js` file was pushed to the file cabinet with the `gulp deploy` command. It will reside in the root folder of your SSP application at <HTML Hosting Root>: /SSP Applications/<Application Publisher>/<Application Name>.



Note: Depending on the number of files in your File Cabinet you may need to scroll through a long list! Also, do not depend on the search. The files available may not be refreshed, resulting in a no matches found error even though the file does exist.

4. In the Touch Points tab, define the Touch Points for this SSP application.

Touch Points are the entry points for your web store. For each touch point defined here, your website links to an SSP application page.

- a. In the Name field, select the desired Touch Point.
- b. In the Entry Page, field select the .ssp file that should be the starting point for this Touch Point.
- c. Click **Add**.

For example, if you want your web store to use the Shopping application when a user goes to your Home page, select the **View Homepage** Touch Point and set the Entry page as the `shopping.ssp` file.

5. Click **Save** on the record.

You are returned to the list of available SSP applications.

6. Deploy the SSP application to your site.

- a. Click **View** next to your SSP application.
- b. Click **Link to Site**.
- c. In the Site dropdown field, select the site you want to deploy this application to.



Note: Only sites already set up are available for selection. If you have not already set up a web site record go to Setup > SuiteCommerce Advanced > Web Site Set Up > New. For detailed instructions, see the help topic [Getting Started](#).

- d. Click **Save**.

After the SSP application has been deployed to your site you can view the site by navigating to the domain defined for that site.

Troubleshooting the Developer Tools

Applies to: SuiteCommerce Web Stores

The following are known problems or errors that you may encounter when installing or running the developer tools:

Maximum Number (100) of Records Allowed for a UPSERTLIST Operation Has Been Exceeded

This error can occur if you deploy a theme or extension that includes more than 100 files or folders at the same level. This can include custom modules, images, and other assets.

NetSuite imposes this limit to control the consumption of web services. To avoid this error, limit the number of files and folders at the same level to 100 when developing themes and extensions. Some options include introducing images across multiple folders and placing custom modules in a different top-level directory.

For more information on record limiting, see *Web Services Governance Overview*.

Error Messages When Running the Developer Tools

You may encounter the following errors when you deploy themes, extensions, and other customizations to NetSuite. They may occur when NetSuite runs with two-factor authentication and you do not properly sign in with the **SCDeployer** role:

Error INVALID_LOGIN_ATTEMPT Invalid login attempt.

This error occurs when deploying or fetching using a role that requires 2FA. To fix this error, log into NetSuite using the developer tools and choose the SCDeployer role. Include the **--to** parameter to any fetch or deploy command to trigger the developer tools to prompt you for new login information.

Error INSUFFICIENT_PERMISSION You do not have privileges to view this page

This error occurs if the SCDeployer role is added to the Employee Record, but the **Audience** in the Restlet Script Deployment is not added correctly for the role.

See [Edit Script Deployment Records](#) for more information on setting this up correctly.

Error When Running Gulp.js Commands

When running gulp with an unsupported version of Node.js you may encounter a Sass-related error similar to the following:

```

gyp ERR! build error
gyp ERR! stack Error: `make` failed with exit code: 2
gyp ERR! stack     at ChildProcess.onExit
(/home/sg/netsuite/ml/node_modules/gulp-sass/node_modules/node-sass/node_modules/pangyp/lib/build.js:272:23)
gyp ERR! stack     at emitTwo (events.js:87:13)
gyp ERR! stack     at ChildProcess.emit (events.js:172:7)
gyp ERR! stack     at Process.ChildProcess._handle.onexit
(internal/child_process.js:200:12)
gyp ERR! System Linux 3.13.0-37-generic
gyp ERR! command "/usr/local/bin/node"
"/home/sg/netsuite/ml/node_modules/gulp-sass/node_modules/node-sass/node_modules/pangyp/bin/node-gyp"
"rebuild"
gyp ERR! cwd
/home/sg/netsuite/ml/node_modules/gulp-sass/node_modules/node-sass
gyp ERR! node -v v4.0.0
gyp ERR! pangyp -v v2.3.2
gyp ERR! not ok
Build failed

```

Resolution: Ensure that you are using Node.js version 0.12.x or older.

Gulp.js Version Mismatch

When running gulp, you may see a warning message related to a version mismatch. This is due to a mismatch between the version installed globally on your system and the version used by the developer tools.

Resolution: This warning is expected and does not cause problems with the developer tools.

Warning Messages Related to GIT

This error is caused by the Sass compiler which expects a GIT server to be installed.

Resolution: This warning is expected and does not cause problems with the developer tools.

ENOSPC Error When Running Local Server

On UNIX systems, the `gulp watch` command (used when running a local server) may return this exception. This is caused when the `gulp` process exceeds the limit of files that can be watched by the system.

Resolution: Enter the following command to resolve this issue:

```
echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf && sudo sysctl -p
```

EMFILE Error When Running a Local Server

On UNIX systems, the `gulp local` command may return this exception if the gulp process exceeds the limit of files that can simultaneously be opened.

Resolution: Enter the following command to increase the system limit on the number of files that can simultaneously be opened.

```
ulimit -n 2048
```

Sass Compilation Errors when Running Gulp

i Applies to: Mont Blanc and later

Changes to the developer tools source code may be required to mitigate errors in Sass compilation when deploying.

SCA uses gulp-sass for compiling .scss files. Node-sass (v3.5), which is a dependency of gulp-sass, has caused some Sass code that previously compiled fine to throw fatal errors, particularly in Bootstrap. These errors are returned when running the `gulp local` command to start your local server or when deploying with `gulp deploy`.

The errors returned begin as follows:

```
formatted Error: You may not @extend an outer selector from within @media you may only @extend selectors within the same directive. . [etc]
```

To correct the compilation errors:

1. Open the command line in the project <Source Code> directory.
2. Execute the following commands to remove the 3.5 dependency and install the expected version.


```
npm cache clear
npm uninstall gulp-sass node-sass
npm config set save-exact true
npm install --save node-sass@3.4.1 gulp-sass@2.1.0
```

i Note: This will modify your package.json. Preserve these changes.

3. Run `gulp local` to validate that the error is resolved.

If you have also installed gulp-sass globally, you might need to uninstall the globals.

To uninstall gulp-sass globals:

1. Open the command line in the top-level directory of your Mont Blanc source code.
2. If you are a **Windows** user, execute the following command as either Administrator or as a User:


```
npm uninstall -g node-sass gulp-sass
```
3. If you are a **Linux / Mac** user, execute the following command as Administrator:


```
npm uninstall -g node-sass gulp-sass sudo
npm uninstall -g node-sass gulp-sass
```

Source Code Error When Running Gulp

i Applies to: Mont Blanc

When deploying source files, you might encounter the following error:

```
SOURCE CODE ERROR. Error: You may not @extend an outer selector from within @media. You may only @extend
selectors within the same directive.
```

If this occurs, a mismatch of node-sass versions exists between the package.json files located in the top-level directory and the node_modules/gulp-sass directory. To fix this error, perform the following.

To correct the Source Code Error:

1. Open the command line in the top-level directory of your source code.
2. Execute the following commands:

Windows Users

```
npm install -g npm
npm cache clear
npm uninstall gulp-sass
npm uninstall node-sass
npm config set save-exact true
npm install node-sass@3.4.1 --save
npm install --save gulp-sass@2.1.0
```

Linux/Mac Users

```
sudo npm install -g npm
npm cache clear
npm uninstall gulp-sass
npm uninstall node-sass
npm config set save-exact true
npm install node-sass@3.4.1 --save
npm install --save gulp-sass@2.1.0
```

i Note: This modifies the package.json file. Preserve these changes.

3. Run `gulp local` to validate that the error is resolved.

If you have also installed gulp-sass globally, you might need to uninstall the globals.

To uninstall gulp-sass globals:

1. Open the command line in the top-level directory of your source code.
2. If you are a **Windows** user, execute the following command as either Administrator or as a User:
`npm uninstall -g node-sass gulp-sass`
3. If you are a **Linux / Mac** user, execute the following command as Administrator:
`npm uninstall -g node-sass gulp-sass`
`sudo npm uninstall -g node-sass gulp-sass`

Overview

 **Applies to:** SuiteCommerce Web Stores

SuiteCommerce lets you configure common properties and modify application behavior for a specified domain using the NetSuite user interface rather than by customizing source code. The SuiteCommerce Configuration record feature provides this capability and is installed with the SuiteCommerce Configuration bundle.

You can:

- Configure properties for a domain
- Read configuration file types information
- Browse the site configuration properties reference section
- Configure and maintain Site Management Tools

See the following topics for more information:

- [Configure Properties](#) – This section explains how to configure properties for a domain and how to copy a SuiteCommerce configuration record.
- [Configuration Properties Reference](#) – This section contains a list of all the configuration properties supported by SuiteCommerce. Properties are grouped by functionality based on how they appear in the Configuration record, listed alphabetically by tab and subtab.
- [Configuration File Types](#) – This section explains the different configuration files you may encounter when implementing SuiteCommerce. The types of files and their purpose differ depending on the version of SuiteCommerce you are implementing.
- [Site Management Tools Configuration](#) – This section contains information about how to configure and maintain Site Management Tools.

Configure Properties

 **Applies to:** SuiteCommerce Web Stores | Aconcagua | Kilimanjaro | Elbrus | Vinson

These topics explain how to configure your SuiteCommerce site using the SuiteCommerce Configuration record. This requires installing the SuiteCommerce Configuration bundle. See the help topic [Install Your SuiteCommerce Application](#) for more information.

This topic explains how to:

- [Configure Properties for a Domain](#)
- [Copy a SuiteCommerce Configuration Record](#)



Note: To configure properties for SuiteCommerce Advanced sites implementing the Mont Blanc release or earlier, you must extend source JavaScript files. For details, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).



Important: If you are using SuiteCommerce Advanced (SCA) and migrating to a later release from Denali or Mont Blanc, you can copy previous configuration files and include them in the Vinson JavaScript files directly. If doing so, you do not need to install the SuiteCommerce Configuration bundle. These existing configuration files take priority over the SuiteCommerce Configuration record. However, if installing Vinson release or later as your initial SCA installation, you must install the SuiteCommerce Configuration bundle to configure your site.



Note: Before configuring SuiteCommerce, ensure that you have created a website and domain for your NetSuite account. See the help topics [Getting Started](#) and [Set Up a Web Store Domain](#) for more information.

Configure Properties for a Domain

Use the domain Configuration record when you want to change domain properties.

To configure properties for a domain:

1. Go to Setup > SuiteCommerce Advanced > Configuration.
2. Select the site that you want to configure from the **Select Website** list.
3. Select the specific domain that you want to configure from the **Select Domain** list.
4. Click **Continue**.
5. Configure properties for the associated domain by clicking the tab and subtab of the feature you want to configure. See [Configuration Properties Reference](#) for details.
6. Save the record.



Important: Saving changes to the SuiteCommerce Configuration record creates a custom record for the selected domain. To configure multiple domains, you must save a SuiteCommerce Configuration record for each domain individually.

You can also copy the configuration of one domain (the origin domain) to another (the destination domain) by using the **Copy Configuration** feature. See [Copy a SuiteCommerce Configuration Record](#).



Note: If your domain is mapped to a CDN, any changes made to the Shopping Application using the SuiteCommerce Configuration record take approximately five minutes to take effect. This delay occurs because the configuration information from the record is published to the sc.shopping.environment.ssp file, which has a five-minute CDN cache by default.

Copy a SuiteCommerce Configuration Record

Use the Copy Configuration function to copy a SuiteCommerce configuration record from one domain to another.

Common scenarios where this function is useful include:

- When you move a site from staging to production
- When you move from one SuiteCommerce release to another



Note: The two domains do not have to be on the same release of SuiteCommerce. For example, you can copy a Kilimanjaro SuiteCommerce Configuration record to another Kilimanjaro domain or to an Aconcagua domain. Since it is common for later versions of SuiteCommerce to have fields that do not exist in previous releases, the tool copies the values of all fields that are common between the two releases and uses default values for the additional fields.

Using the Copy Configuration function, you select the origin domain, select the destination domain, then click Copy Configuration.

In the procedure below:

- **Origin Website/Domain** refers to the selection of the source configuration record. The tool copies the configuration record values from this selected website/domain. The values in this record are not changed.
- **Destination Website/Domain** refers to the selection of the configuration record you are overwriting. The tool populates the field values from the origin configuration record into this destination configuration record.

To copy a SuiteCommerce Configuration record:



Important: This procedure overwrites the destination domain configuration record with the content of the origin domain configuration record. Copy to a test domain (especially if you are copying from a Sandbox domain) prior to copying to a production domain and ensure that the configuration values being copied align with your configuration objectives for the destination domain.

1. Go to Setup > SuiteCommerce Advanced > Configuration.
2. Click **Copy Configuration**.

3. Select the origin website from the **Origin Website** list.
4. Select the origin domain from the **Origin Domain** list.
5. Select the destination site from the **Destination Website** list.
6. Select the destination domain from the **Destination Domain** list
7. Click **Copy Configuration**.
8. Optional: To change any of the values in the destination SuiteCommerce Configuration record, see: [Configure Properties for a Domain](#).

 **Note:** The Copy Configuration function is a part of the SuiteCommerce Configuration bundle.

Configuration Properties Reference

ⓘ Applies to: SuiteCommerce Web Stores | Aconcagua | Kilimanjaro | Elbrus | Vinson

This section contains a list of all the configuration properties supported by SuiteCommerce. In this section, properties are grouped by functionality based on how they appear in the Configuration record, listed alphabetically by tab and subtab.

- [Advanced Tab](#)
- [Checkout Tab](#)
- [Integrations Tab](#)
- [Layout Tab](#)
- [Legacy Tab](#)
- [Multi-Domain Tab](#)
- [My Account Tab](#)
- [Search Tab](#)
- [Shopping Tab](#)
- [Shopping Catalog Tab](#)
- [Store Locator Tab](#)

ⓘ Note: SuiteCommerce Configuration is available as a separate bundle and requires SuiteCommerce or SuiteCommerce Advanced (Vinson or later). If you do not have the SuiteCommerce Configuration bundle installed, this reference still applies to configuration properties available in your implementation.

Each property in this reference includes a brief description of the property plus the following information. In many cases, this reference provides links to topics with detailed information.

- **ID** – the name of the property in the JavaScript and JSON configuration files. This value must only contain alphanumeric characters and periods. No special characters. In some cases, the ID differs in pre-Vinson implementations.
- **UI Location** – the tab and subtab location in the Configuration record's user interface. This is where you configure the property in Vinson implementations and later of SuiteCommerce Advanced and in SuiteCommerce.
- **Configuration file (pre-Vinson)** – the JavaScript file where you configure the property in pre-Vinson implementations of SuiteCommerce Advanced. This only applies to pre-Vinson implementations of SuiteCommerce Advanced.

Advanced Tab

The settings on this tab let you configure advanced properties for your SuiteCommerce site. This tab includes configurable properties grouped in the following subtabs:

- [Backend Subtab](#)
- [Cache Subtab](#)
- [Custom Fields Subtab](#)

- Filter Site Subtab
- Favicon Path Subtab
- Image Resize Subtab
- Pagination Subtab
- Search Results Subtab

Backend Subtab

These settings configure SuiteScript properties. You can configure a variety of objects server-side. Among other things, modifying these objects can often improve performance of your website by restricting search results.

nlapiSearchRecord Results Per Page

This integer specifies the default number of search results displayed per page. Any SuiteCommerce feature that displays search results uses this value by default.

ID	suitescriptResultsPerPage
UI Location	Advanced > Backend
JSON file	SuiteScript.json
Configuration file (pre-Vinson)	Configuration.js

Items Fields Advanced Name

This string sets the **Field Set** to be used when using the Item Search API to return item details. By default, this is set to **order**. If you are using a different Field Set name, you must define that Field Set name here.

Using a different Field Set name for each application can improve site performance. For each application, only those details required are returned for the pages specific to that implementation. For example, in the Shopping item details page, you may want to display a different set of fields than when your customer is in the My Account pages. In this case you can define two Field Sets: **details** to be used in ShopFlow and **myDetails** to be used in the My Account pages.

More Information: [Configure Minimum Setup Requirements](#)

ID	fieldKeys.itemsFieldsAdvancedName
UI Location	Advanced > Backend
JSON file	SuiteScript.json
Configuration file (pre-Vinson)	Configuration.js

Shopping's Order

This array defines the keys of the Shopping application's order record. For example: shipaddress, summary, promocodes etc. To maximize performance, limit the set to only those fields required for your implementation.

ID	orderShoppingFieldKeys.keys
UI Location	Advanced > Backend
ID (pre-Vinson)	order_shopping_field_keys
Configuration file (pre-Vinson)	Configuration.js

Shopping's Order Items

This array defines the item fields returned by the Commerce API within the Shopping application.

ID	orderShoppingFieldKeys.items
UI Location	Advanced > Backend
ID (pre-Vinson)	order_shopping_field_keys.items
Configuration file (pre-Vinson)	Configuration.js

Checkout's Order

This array defines the keys of the Checkout application's order record. For example: shipaddress, summary, promocodes etc.

ID	orderCheckoutFieldKeys.keys
UI Location	Advanced > Backend
ID (pre-Vinson)	order_checkout_field_keys
Configuration file (pre-Vinson)	Configuration.js

Checkout's Order Items

This array defines the item fields returned by the Commerce API within the Checkout application.

ID	orderCheckoutFieldKeys.items
UI Location	Advanced > Backend
ID (pre-Vinson)	order_checkout_field_keys.items
Configuration file (pre-Vinson)	Configuration.js

Item Fields Standard

This array defines the set of fields to be returned by the Commerce API. To optimize performance, limit the set included here to only those fields required for your website. Also, use this object to ensure that custom item fields in your account are included in the Item Search API results. Add the custom field name to the list.

ID	fieldKeys.itemsFieldsStandardKeys
-----------	-----------------------------------

UI Location	Advanced > Backend
Configuration file (pre-Vinson)	Configuration.js

Cache Subtab

These settings configure caching behavior in SuiteCommerce.

Content Page CDN

This string specifies the CDN cache duration for content pages. Possible values are Short, Medium, and Long.

More Information: [CDN Caching](#)

ID	cache.contentPageCdn
UI Location	Advanced > Cache
JSON file	ShoppingApplication.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Content Page TTL

This integer specifies the duration of the application Time To Live (TTL) cache for content pages. This value is specified in seconds. The total value must be between 300 (5 minutes) and 7200 (2 hours).

ID	cache.contentPageTtl
UI Location	Advanced > Cache
JSON file	ShoppingApplication.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Site Cache Setting (Pre-Mont Blanc Only)

Specifies the duration of the site settings cache. This caches all properties (except touch points) using the application cache. By default this is set to 7200. This value is defined in seconds and must be between 300 (5 minutes) and 7200 (2 hours).

Note: This property does not exist with Mont Blanc release of SuiteCommerce Advanced and later or with SuiteCommerce. To configure your site cache settings in a pre-Mont Blanc implementation, edit the siteSettings property in the backend Configuration.js file. For detailed information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

ID	cache.siteSettings
UI Location	none
Configuration file (pre-Vinson)	Configuration.js

Custom Fields Subtab

ⓘ Applies to: SuiteCommerce Web Stores | Aconcagua | Kilimanjaro | Elbrus

Sales Order

These settings specify which custom transaction body fields (by Field ID) you want exposed to your web store.

More Information: [Commerce Custom Fields](#)

ID	customFields.salesorder
UI Location	Advanced > Custom Fields

Favicon Path Subtab

ⓘ Applies to: SuiteCommerce Web Stores | Aconcagua | Kilimanjaro

Favicon of the Website

This string specifies the path to the favicon for the website. This property is blank by default.

ID	faviconPath
UI Location	Advanced > Favicon Path
JSON file	Favicon.json

Filter Site Subtab

These settings filter record lists by website.

Filter Site Option

This string specifies how record lists are filtered. The default value is **current**. Possible values are (as string):

- **all** – filters records for all websites in the current NetSuite account.
- **current**– filters records only in the current account.
- **siteIds** – filters records for a specific website by Site ID.

ID	filterSite.option
UI Location	Advanced > Filter Site
ID (pre-Vinson)	filter_site
JSON file	FilterSite.json

Configuration file (pre-Vinson)	Configuration.js
--	------------------

Filter Site IDs

This array filters records for web sites specified by site ID (see [Filter Site Option](#)). This value contains an array of comma-separated numeric IDs.

ID	filterSite.ids
UI Location	Advanced > Filter Site
ID (pre-Vinson)	None
JSON file	FilterSite.json
Configuration file (pre-Vinson)	Configuration.js

Image Resize Subtab

These settings map image-resize values to names as used in the application (Setup > Website > Advanced > Image Resizing). The current sizes used in SuiteCommerce are:

- thumbnail: 175 x 175 (used in the search results)
- main: 600 x 600 (used as main image in the PDP)
- tinythumb: 50 x 50 (used as the tiny thumbs in the PDP's image gallery)
- zoom: 1200 x 1200 (used in the PDP's zoomed image)
- fullscreen: 1600 x 1600
- homeslider: 200 x 200
- homecell: 125 x 125

Each Image Resize property contains the following properties:

- **Size** (string) – defines the image size name used in the application.
- **Value** (string) – specifies the corresponding value given in the website record (Image Size ID).

More Information: [Setting Up Image Resizing for SuiteCommerce](#)

IDs	imageSizeMapping imageSizeMapping.id imageSizeMapping.value
UI Location	Advanced> Image Resize
JSON file	ApplicationSkeleton.Layout.Images.json
Configuration file (pre-Vinson)	SC.Configuration.js

Pagination Subtab

These settings configure the pagination settings when multiple pages are returned. If enabled, the Page List displays links for multiple-page browsing.

Desktop > Show Page List

This boolean displays or hides the Page List on desktop browsers.

ID	defaultPaginationSettings.showPageList
UI Location	Advanced > Pagination
JSON file	Pagination.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js, SC.Shopping.Configuration.js

Desktop > Pages To Show

This number specifies the maximum number of pages displayed in the page list on desktop browsers. The default value is 4.

ID	defaultPaginationSettings.pagesToShow
UI Location	Advanced > Pagination
JSON file	Pagination.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js, SC.Shopping.Configuration.js

Desktop > Show Page Indicator

This boolean displays or hides a current page indicator in the page list on desktop browsers.

ID	defaultPaginationSettings.showPageIndicator
UI Location	Advanced > Pagination
JSON file	Pagination.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js, SC.Shopping.Configuration.js

Phone > Show Page List

This boolean displays or hides the Page List on mobile browsers.

ID	defaultPaginationSettingsPhone.showPageList
UI Location	Advanced > Pagination
JSON file	Pagination.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js, SC.Shopping.Configuration.js

Phone > Pages To Show

This number specifies the maximum number of pages displayed in the page list on mobile browsers. The default value is 4.

ID	defaultPaginationSettingsPhone.pagesToShow
UI Location	Advanced > Pagination
JSON file	Pagination.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js, SC.Shopping.Configuration.js

Phone > Show Page Indicator

This boolean displays or hides a current page indicator in the page list on mobile browsers.

ID	defaultPaginationSettingsPhone.showPageIndicator
UI Location	Advanced > Pagination
JSON file	Pagination.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js, SC.Shopping.Configuration.js

Tablet > Show Page List

This boolean displays or hides the Page List on tablet browsers.

ID	defaultPaginationSettingsTablet.showPageList
UI Location	Advanced > Pagination
JSON file	Pagination.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js, SC.Shopping.Configuration.js

Tablet > Pages To Show

This number specifies the maximum number of pages displayed in the page list on tablet browsers. The default value is 4.

ID	defaultPaginationSettingsTablet.pagesToShow
UI Location	Advanced > Pagination
JSON file	Pagination.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js, SC.Shopping.Configuration.js

Tablet > Show Page Indicator

This boolean displays or hides a current page indicator in the page list on tablet browsers.

ID	defaultPaginationSettingsTablet.showPageIndicator
UI Location	Advanced > Pagination
JSON file	Pagination.json

Configuration file (pre-Vinson)

SC.MyAccount.Configuration.js, SC.Shopping.Configuration.js

Search Results Subtab

These settings configure options to be passed when querying the Search API.

Search API Fieldsets

This array contains the following properties:

- **ID** (string) – specifies the search type of the resource. Options include:
 - Facets
 - itemDetails
 - relatedItems
 - correlatedItems
 - merchandizingZone
 - typeAhead
 - itemsSearcher
 - CmsAdapterSearch
- **Fieldset** (string) – specifies the fieldset parameter value.
- **Include** (string) – specifies the include parameter value.

More Information: [Define Field Sets](#)

IDs	searchApiMasterOptions searchApiMasterOptions.id searchApiMasterOptions.fieldset searchApiMasterOptions.include
UI Location (Vinson)	Advanced > Search Results
JSON file	ItemsSearchAPI.json
Configuration file (pre-Vinson)	SC.Configuration.js

Checkout Tab

The settings on this tab let you configure properties related to the Checkout application. This tab includes overall checkout properties plus the following subtabs:

- [Credit Card Subtab](#)
- [Forms Subtab](#)
- [Payment Methods Subtab](#)

Enable Pickup In Store

This property applies to:

- SuiteCommerce Advanced - Elbrus and later
- SuiteCommerce

This boolean enables the Pickup In Store feature for SuiteCommerce.

More Information: [Pickup In Store](#)

ID	isPickupInStoreEnabled
UI Location	Checkout
JSON file	PickupInStore.json

Pickup In Store Sales Order Custom Form ID

This property applies to:

- SuiteCommerce Advanced - Elbrus and later
- SuiteCommerce

This string specifies the ID for the sales order form customized for the Pickup In Store feature.

More Information: [Pickup In Store](#)

ID	pickupInStoreSalesOrderCustomFormId
UI Location	Checkout
JSON file	PickupInStore.json

Skip Checkout Login

This boolean enables anonymous users to skip the login/register page when they navigate to Checkout. The user is redirected to the first checkout step. This is disabled by default.

With this property enabled, the checkout application does not establish a guest session until data is sent to the backend for validation (any Ajax call to the server). Simply entering a full name does not create a session because the order has not been submitted to the backend. Without validation, some necessary session information is not yet known. For example, whether the user has permissions to pay with terms (invoices).

After the guest session is created, the Checkout application refers to the user as **Guest**. If the user wants to log in as a registered user after this time, they must log out and either log in as a registered user or create an account. The application also gives a guest user the option to create an account after the checkout process is completed.



Important: Invoices can not be set as a payment option for a One Page Checkout flow with this property enabled. This property is not supported on Site Builder sites or when **Multiple Ship To** is enabled and configured for your web store.

More Information: [Checkout](#)

ID	checkoutApp.SkipLogin
ID (pre-Vinson)	checkout_skip_login

UI Location	Checkout
JSON file	CheckoutApplication.json
Configuration file (pre-Vinson)	Configuration.js

Enable Multiple Shipping

This boolean lets users specify multiple shipping addresses for an order.

More Information: [Multiple Ship To](#)

ID	isMultiShippingEnabled
UI Location	Checkout
JSON file	CheckoutApplication.json
Configuration file (pre-Vinson)	Configuration.js

Remove PayPal Addresses

This property applies to:

- SuiteCommerce Advanced - Elbrus and later
- SuiteCommerce

This boolean lets users remove addresses from PayPal after placing the order, as the addresses are not valid for NetSuite.

More Information: [PayPal Address not Retained in Customer Record](#)

ID	removePaypalAddress
UI Location	Checkout
JSON file	CheckoutApplication.json

Use Standard Header and Footer

This property applies to:

- SuiteCommerce Advanced - Aconcagua and later
- SuiteCommerce

There are two types of headers and footers:

- **Standard:** Used on non-checkout site pages. These headers and footers include navigation and other links.
- **Checkout:** Used on checkout pages. These headers and footers are the default headers and footers for checkout pages and do not include navigation or other links. These headers and footers are designed to minimize the chances of a site user navigating away from the site prior to completing a purchase.

The **Use Standard Header and Footer** boolean, when enabled (checked), causes the standard site header and footer to display on checkout pages. This option is disabled by default. When disabled (cleared), the checkout pages use the default (simplified) headers and footers.

ID	useStandardHeaderFooter
UI Location	Checkout
JSON file	CheckoutApplication.json

Enable 3D Secure Payments

This property applies to:

- SuiteCommerce Advanced - Kilimanjaro and later
- SuiteCommerce

This boolean enables 3D Secure payments for a site.

More Information: [3D Secure Payment Authentication](#)

ID	isThreeDSecureEnabled
UI Location	Checkout
JSON file	CheckoutApplication.json

Checkout Steps

This string specifies a checkout flow configuration from the drop down list. Each checkout flow guides the user through the same tasks, but uses a different sequence of pages as defined below.

More Information: [Checkout Flows](#)

ID	checkoutApp.checkoutSteps
UI Location	Checkout
JSON file	CheckoutApplication.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

PayPal Logo URL

This string specifies the URL of the PayPal logo. SuiteCommerce displays this logo in the check out process.

ID	checkoutApp.paypalLogo
UI Location	Checkout
ID (pre-Vinson)	paypal_logo
JSON file	CheckoutApplication.json

Configuration file (pre-Vinson)	SC.Checkout.Configuration.js
--	------------------------------

Invoice Terms and Conditions

This string specifies the text of the invoice terms and conditions in HTML format.

ID	checkoutApp.invoiceTermsAndConditions
UI Location	Checkout
JSON file	CheckoutApplication.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

Credit Card Subtab

These settings configure how credit card information is displayed.

Show Credit Card Help

This boolean displays or hides help text for finding a credit card security code.

ID	creditCard.showCreditCardHelp
UI Location	Checkout > Credit Cards
JSON file	CreditCard.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

Credit Card Help Title

This string specifies the title of the credit card help link. This title is displayed only if the **Show Credit Card Help** property is enabled.

ID	creditCard.creditCardHelpTitle
UI Location	Checkout > Credit Cards
JSON file	CreditCard.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

CVV All Cards Image

This string specifies the URL of the image showing the card verification value (CVV) number on credit cards. This image displays a number located on the back of the credit card.

ID	creditCard.imageCvvAllCards
UI Location	Checkout > Credit Cards

JSON file	CreditCard.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

CVV American Card Image

This string specifies the URL for the image showing the card verification value (CVV) number on American credit cards. This image shows a number located on the front of the credit card.

ID	creditCard.imageCvvAmericanCard
UI Location	Checkout > Credit Cards
JSON file	CreditCard.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

Credit Card Secure Info

This string specifies the security-related text displayed in the user interface.

ID	creditCard.creditCardShowSecureInfo
UI Location	Checkout > Credit Cards
JSON file	CreditCard.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

Forms Subtab

These settings configure the behavior of checkout forms.

Applies to:

- SuiteCommerce Advanced - Vinson and later
- SuiteCommerce

More Information: [Checkout](#)

Auto Populate Name and Email

This boolean enables or disables auto-population of a guest shopper's name and email in any forms during checkout. If this field is disabled, no fields will populate automatically. If this field is enabled, **Login as Guest, Show Name** and **Login as Guest, Show Email** options must be enabled (as applicable) to populate automatically.

ID	autoPopulateNameAndEmail
UI Location	Checkout > Forms
JSON file	CheckoutApplication.json

Login as Guest, Show Name

This boolean displays or hides first and last name input fields when a user registers as a guest. If enabled, and the **Auto Populate Name and Email** property is also enabled, these names will be required and automatically populate throughout the checkout process where applicable. If disabled, no fields populate automatically.

ID	forms.loginAsGuest.showName
UI Location	Checkout > Forms
JSON file	CheckoutApplication.json

Login as Guest, Show Email

This boolean displays or hides an email address input field when a user registers as a guest. If enabled, and the **Auto Populate Name and Email** property is also enabled, this address will be required and automatically populate throughout the checkout process where applicable. If disabled, no fields populate automatically.

ID	forms.loginAsGuest.showEmail
UI Location	Checkout > Forms
JSON file	CheckoutApplication.json

Show Address Line 2

This boolean displays or hides a secondary address line during Checkout.

ID	forms.address.showAddressLineTwo
UI Location	Checkout > Forms
JSON file	CheckoutApplication.json

Payment Methods Subtab

This array specifies the Payment Method record in NetSuite using the Payment Method ID.

- **Key** (string) – This number must match the property key of the objects returned by the JavaScript call (run in the Browser Console): `SC.ENVIRONMENT.siteSettings.paymentmethods`. Enter the key value returned to configure each method. To edit or add new payment methods, in NetSuite go to Setup > Accounting > Accounting Lists, then filter by **Payment Method**.
- **Regex** (string) – defines the regular expression that matches the payment method.
- **Description** (string) – defines a text description of the payment method.

More Information: [Creating a Payment Method](#)

IDs	paymentmethods paymentmethods.key
------------	--------------------------------------

	paymentmethods.regex paymentmethods.description
UI Location	Checkout > Payment Methods
JSON file	PaymentMethods.json
Configuration file (pre-Vinson)	SC.Configuration.js

Integrations Tab

The settings on this tab let you configure properties related to different integration options available for your site, such as Google, Twitter, and Facebook. This tab includes the following subtabs:

- [AddThis Subtab](#)
- [Bronto Subtab](#)
- [Categories Subtab](#)
- [Facebook Subtab](#)
- [Google AdWords Subtab](#)
- [GooglePlus Subtab](#)
- [Google Tag Manager Subtab](#)
- [Google Universal Analytics Subtab](#)
- [Pinterest Subtab](#)
- [Site Management Tools Subtab](#)
- [Twitter Subtab](#)

AddThis Subtab

These settings configure the AddThis sharing service. This service defines which SEO services are active on your SuiteCommerce site.

ID	addThis
UI Location	Integrations > addThis
JSON file	addThis.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Enable

This boolean enables or disables the AddThis sharing service.

ID	addThis.enable
UI Location	Integrations > addThis
JSON file	addThis.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Public ID

This string specifies your AddThis Profile ID.

ID	addThis.pubId
UI Location	Integrations > addThis
JSON file	addThis.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Toolbox Class

This string specifies the class to be added to the AddThis Toolbox.

ID	addThis.toolboxClass
UI Location	Integrations > addThis
JSON file	addThis.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Services to Show

This array lists the social services to be displayed in the AddThis buttons. Each Service contains the following properties:

- **Key** (string) – defines the sharing service. For example: facebook, pinterest, twitter.
- **Value** (string) – defines a text description to display for the associated service.

IDs	addThis.servicesToShow addThis.servicesToShow.key addThis.servicesToShow.value
UI Location	Integrations > addThis
JSON file	addThis.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Options

This array defines the configuration options of the AddThis sharing service.

- **Key** (string) – defines the AddThis configuration variable.
- **Value** (string)– defines each configuration variable according to the AddThis service.

IDs	addThis.options addThis.options.key addThis.options.value
UI Location	Integrations > addThis

JSON file	addThis.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Bronto Subtab

These settings configure Bronto integration in SuiteCommerce.

Bronto is a NetSuite company that provides an advanced marketing automation engine and solutions for shopping cart abandonment, post-purchase campaigns and so forth. Bronto can be easily integrated with your SuiteCommerce web store.

The following Bronto Applications are supported in SuiteCommerce:

- Cart Recovery
- Conversion Tracking
- Coupon Manager
- Pop-up Manager

More Information: [Bronto Integration](#)

Account ID

This string specifies your Bronto account ID.

ID	bronto.accountId
UI Location	Integrations > Bronto
JSON file	BrontoIntegration.json
Configuration file (pre-Vinson)	SC.Configuration.js

Adapter URL

This string specifies the URL of the Bronto adapter.

ID	bronto.adapterUrl
UI Location	Integrations > Bronto
JSON file	BrontoIntegration.json

Categories Subtab

These settings configure properties related to Commerce Categories.

More Information: [Commerce Categories](#)

Category Navigation Menu Level Deepness

This integer specifies how many levels of the category hierarchy to show in the Categories navigation menu.

ID	categories.menuLevel
UI Location	Integrations > Categories
JSON file	Categories.json

Show Categories Navigation Menu

This boolean enables or disables the Categories navigation menu.

ID	categories.addToNavigationTabs
UI Location	Integrations > Categories
JSON file	Categories.json

Side Menu > Sort By

This string specifies the Category record field to act as the primary sort field in the Categories sidebar. This is set to sequencenumber by default. In this case, the sidebar categories sort by the sequence number set in NetSuite.

ID	categories.sideMenu.sortBy
UI Location	Integrations > Categories
JSON file	Categories.json

Side Menu > Additional Fields

This string specifies any additional fields added to the default list returned by the ItemSearch API for use in the application.

The default list returned is:

- internalid
- name
- sequencenumber
- urlfragment
- displayinsite

ID	categories.sideMenu.additionalFields
UI Location	Integrations > Categories
JSON file	Categories.json

Side Menu > Collapsible

This boolean specifies if the Categories side menu is collapsible and expandable.

ID	categories.sideMenu.uncollapsible
-----------	-----------------------------------

UI Location	Integrations > Categories
JSON file	Categories.json

Side Menu > Show Max

This integer specifies maximum number of categories to show in the list before displaying a **Show More** link. The default is 5.

ID	categories.sideMenu.showMax
UI Location	Integrations > Categories
JSON file	Categories.json

Side Menu > Collapsed

This boolean specifies if the Categories sidebar is collapsed when the page loads. The default is to appear expanded.

ID	categories.sideMenu.collapsed
UI Location	Integrations > Categories
JSON file	Categories.json

Sub Categories > Sort By

This string specifies the Category record field to act as the primary sort field for sub categories. This is set to sequencenumber by default. In this case, the sidebar categories will sort by the sequence number set in NetSuite.

ID	categories.subCategories.sortBy
UI Location	Integrations > Categories
JSON file	Categories.json

Sub Categories > Additional Fields

This string specifies any additional fields added to the default list returned by the ItemSearch API for use in the application for sub categories.

The default list returned is:

- internalid
- name
- description
- sequencenumber
- urlfragment
- thumbnailurl

- displayinsite

ID	categories.subCategories.fields
UI Location	Integrations > Categories
JSON file	Categories.json

Category Fields

This string specifies any optional fields added to the default list for category fields. Category Fields are the fields that appear in the Category page.

ID	categories.category.fields
UI Location	Integrations > Categories
JSON file	Categories.json

Breadcrumb Fields

This string specifies any additional fields added to the default list returned by the ItemSearch API for use in the breadcrumb navigation. The default list returned is:

- internalid
- name
- displayinsite

ID	categories.breadcrumb.fields
UI Location	Integrations > Categories
JSON file	Categories.json

Menu > Sort By

This string specifies the Category record field as the primary sort field in the Navigation menu. This is set to sequencenumber by default. In this case, the navigation menu categories will sort by the sequence number set in NetSuite.

ID	categories.menu.sortBy
UI Location	Integrations > Categories
JSON file	Categories.json

Menu Fields

This property specifies any additional fields added to the default list returned by the ItemSearch API for use in the Navigation menu.

The default list returned is:

- internalid
- name
- sequencenumber
- displayinsite

ID	categories.menu.fields
UI Location	Integrations > Categories
JSON file	Categories.json

Facebook Subtab

These settings configure Facebook popup integration in SuiteCommerce. The `popupOptions` configured here get passed as the third parameter into the `window.Open` method of `SocialSharing.Plugins.Facebook.js`.

More Information: [Facebook Share](#)

Enable

This boolean enables or disables the Facebook addThis sharing service.

ID	facebook.enable
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Application ID

This string sets the custom application ID to associate the domain to your application ID.

ID	facebook.appId
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Facebook Popup Status

This string specifies if the Facebook popup is enabled.

ID	facebook.popupOptions.status
UI Location	Integrations > Facebook
JSON file	facebook.json

Configuration file (pre-Vinson)	SC.Shopping.Configuration.js
--	------------------------------

Facebook Popup Resizable

This string specifies if the Facebook popup is resizable.

ID	facebook.popupOptions.resizable
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Facebook Popup Scrollbars

This string specifies if the Facebook popup displays scrollbars.

ID	facebook.popupOptions.scrollbars
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Facebook Popup Personalbar

This string specifies if the Facebook popup displays a personal bar.

ID	facebook.popupOptions.personalbar
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Facebook Popup Directories

This string specifies if the Facebook popup displays a directories bar.

ID	facebook.popupOptions.directories
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Facebook Popup Location

This string specifies if the Facebook popup displays a location bar.

ID	facebook.popupOptions.location
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Facebook Popup Toolbar

This string specifies if the Facebook popup displays a tool bar.

ID	facebook.popupOptions.toolbar
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Facebook Popup Menubar

This string specifies if the Facebook popup displays a menu bar.

ID	facebook.popupOptions.menubar
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Facebook Popup Width

This integer specifies the width (in pixels) of the Facebook popup.

ID	facebook.popupOptions.width
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Facebook Popup Height

This integer specifies the height (in pixels) of the Facebook popup.

ID	facebook.popupOptions.height
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Facebook Popup Left

This integer specifies the left offset (in pixels).

ID	facebook.popupOptions.left
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Facebook Popup Top

This integer specifies the top offset (in pixels).

ID	facebook.popupOptions.top
UI Location	Integrations > Facebook
JSON file	facebook.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Google AdWords Subtab

These settings configure Google AdWords.

More Information: [Google Ads](#)

Google AdWords ID

This string specifies your Google AdWords ID. This value corresponds to the conversion tag you created in your Google AdWords account.

ID	googleAdWordsConversion.id
UI Location	Integrations > Google Adwords
JSON file	GoogleAdWords.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js SC.MyAccount.Configuration.js SC.Checkout.Configuration.js

AdWords Value

This string specifies the value property you defined in the conversion tag you created in your Google AdWords account.

ID	googleAdWordsConversion.value
UI Location	Integrations > Google Adwords

JSON file	GoogleAdWords.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js SC.MyAccount.Configuration.js SC.Checkout.Configuration.js

Google AdWords Label

This string specifies your Google AdWords label. This value corresponds to the conversion tag you created in your Google AdWords account.

ID	googleAdWordsConversion.label
UI Location	Integrations > Google Adwords
JSON file	GoogleAdWords.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js SC.MyAccount.Configuration.js SC.Checkout.Configuration.js

GooglePlus Subtab

These settings configure GooglePlus integration in SuiteCommerce.

Enable

This boolean enables or disables the GooglePlus addThis property.

ID	googlePlus.enable
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

GooglePlus Popup Status

This string specifies if the GooglePlus popup is enabled.

ID	googlePlus.popupOptions.status
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

GooglePlus Popup Resizable

This string specifies if the GooglePlus popup is resizable.

ID	googlePlus.popupOptions.resizable
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

GooglePlus Popup Scrollbars

This string specifies if the GooglePlus popup displays scrollbars.

ID	googlePlus.popupOptions.scrollbars
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

GooglePlus Popup Personalbar

This string specifies if the GooglePlus popup displays a personal bar.

ID	googlePlus.popupOptions.personalbar
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

GooglePlus Popup Directories

This string specifies if the GooglePlus popup displays a directories bar.

ID	googlePlus.popupOptions.directories
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

GooglePlus Popup Location

This string specifies if the GooglePlus popup displays a location bar.

ID	googlePlus.popupOptions.location
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json

Configuration file (pre-Vinson)	SC.Shopping.Configuration.js
--	------------------------------

GooglePlus Popup Toolbar

This string specifies if the GooglePlus popup displays a tool bar.

ID	googlePlus.popupOptions.toolbar
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

GooglePlus Popup Menubar

This string specifies if the GooglePlus popup displays a menu bar.

ID	googlePlus.popupOptions.menubar
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

GooglePlus Popup Width

This integer specifies the width (in pixels) of the GooglePlus popup.

ID	googlePlus.popupOptions.width
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

GooglePlus Popup Height

This integer specifies the height (in pixels) of the GooglePlus popup.

ID	googlePlus.popupOptions.height
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

GooglePlus Popup Left

This integer specifies the left offset (in pixels) of the GooglePlus popup.

ID	googlePlus.popupOptions.left
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

GooglePlus Popup Top

This integer specifies the top (in pixels) of the GooglePlus popup.

ID	googlePlus.popupOptions.top
UI Location	Integrations > GooglePlus
JSON file	googlePlus.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Google Tag Manager Subtab

These settings configure properties related to the Google Tag Manager.

More: [Google Tag Manager](#)

Google Tag Manager ID

This string specifies the value of the Google Tag Manager ID. Setting this property enables this feature on your SuiteCommerce site.

More Information: [Step 2: Sign Up for GTM](#)

ID	tracking.googleTagManager.id
UI Location	Integrations > Google Tag Manager
JSON file	GoogleTagManager.json
Configuration file (pre-Vinson)	SC.Configuration.js

Google Tag Manager Data Name

This string specifies the data layer object that contains all of the information that you want to pass to Google Tag Manager.

ID	tracking.googleTagManager.dataLayerName
UI Location	Integrations > Google Tag Manager
JSON file	GoogleTagManager.json
Configuration file (pre-Vinson)	SC.Configuration.js

Is A Multi-Domain Site?

This property applies to:

- SuiteCommerce Advanced - 2018.2 and later
- SuiteCommerce

This boolean specifies if shopping and checkout use separate domains. Enable this option to preserve data between sessions when using separate domains for shopping and checkout.

ID	tracking.googleTagManager.isMultiDomain
UI Location	Integrations > Google Tag Manager
JSON file	GoogleTagManager.json

Google Universal Analytics Subtab

These settings configure properties for Google Universal Analytics. Google Universal Analytics is a third party analytics solution that can help you evaluate traffic on your website using data based on visitor tracking information.

More Information: [Configuring GUA](#)

Google Universal Analytics ID

This string specifies the Google Universal Analytics Tracking ID.

SuiteCommerce inserts this property into each page of the application. To locate the tracking ID, login to your Google Analytics account and go to Admin > Property > Property Settings.

More Information: [Set Up GUA Account and Website Property](#)

ID	tracking.googleUniversalAnalytics.propertyID
UI Location	Integrations > Google Universal Analytics
JSON file	GoogleUniversalAnalytics.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

Google Universal Analytics Domain

This string specifies the domain used for Google Universal Analytics. The value you enter depends on which application you want to use Google Universal Analytics.

More Information: [Configuring GUA](#)

ID	tracking.googleUniversalAnalytics.domainName
UI Location	Integrations > Google Universal Analytics
JSON file	GoogleUniversalAnalytics.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

Google Universal Analytics Secure Domain

This string specifies the secure domain used for Google Universal Analytics. Possible values are the same as the **Google Universal Analytics Domain** property.

ID	tracking.googleUniversalAnalytics.domainNameSecure
UI Location	Integrations > Google Universal Analytics
JSON file	GoogleUniversalAnalytics.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

Pinterest Subtab

These settings configure Pinterest integration in SuiteCommerce.

More Information: [Pinterest](#)

Enable Hover

This boolean enables or disables the Pinterest Pin It to hover over the image.

ID	pinterest.enableHover
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Enable Button

This boolean enables or disables the Pinterest PinIt button.

ID	pinterest.enableButton
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Image Size

This string specifies the Pinterest image size ID to show on Pinterest.

ID	pinterest.imageSize
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pinterest Popup Status

This string specifies if the Pinterest popup is enabled.

ID	pinterest.popupOptions.status
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pinterest Popup Resizable

This string specifies if the Pinterest popup is resizable.

ID	pinterest.popupOptions.resizable
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pinterest Popup Scrollbars

This string specifies if the Pinterest popup displays scroll bars.

ID	pinterest.popupOptions.scrollbars
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pinterest Popup Personalbar

This string specifies if the Pinterest popup displays a personal bar.

ID	pinterest.popupOptions.personalbar
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pinterest Popup Directories

This string specifies if the Pinterest popup displays a directories bar.

ID	pinterest.popupOptions.directories
-----------	------------------------------------

UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pinterest Popup Location

This string specifies if the Pinterest popup displays a location bar.

ID	pinterest.popupOptions.location
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pinterest Popup Toolbar

This string specifies if the Pinterest popup displays a tool bar.

ID	pinterest.popupOptions.toolbar
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pinterest Popup Menubar

This string specifies if the Pinterest popup displays a menu bar.

ID	pinterest.popupOptions.menubar
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pinterest Popup Width

This number specifies the width (in pixels) of the Pinterest popup.

ID	pinterest.popupOptions.width
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pinterest Popup Height

This number specifies the height (in pixels) of the Pinterest popup.

ID	pinterest.popupOptions.height
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pinterest Popup Left

This number specifies the left offset (in pixels) of the Pinterest popup.

ID	pinterest.popupOptions.left
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pinterest Popup Top

This number specifies the top offset (in pixels) of the Pinterest popup.

ID	pinterest.popupOptions.top
UI Location	Integrations > Pinterest
JSON file	pinterest.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Site Management Tools Subtab

These settings configure properties for the Site Management Tools (SMT).

More Information: [Site Management Tools](#)

Use Site Management Tools

This boolean enables or disables SMT integration in SuiteCommerce. Content Delivery and SMT are mutually exclusive features. If you want to use Content Delivery, this property must be unchecked.

ID	cms.useCMS
UI Location	Integrations > Site Management Tools
JSON file	CMS.json
Configuration file (pre-Vinson)	Configuration.js

Disable ESC Key to Login

This property applies to:

- SuiteCommerce Advanced - Elbrus and later
- SuiteCommerce

This boolean enables or disables SMT's escape-to-login feature. This property is unchecked by default, allowing users to press the ESC key to login to SMT. To disable your site's escape-to-login feature, check this property.

More Information: [Disable Esc Key to Log In](#)

ID	cms.escapeToLoginDisabled
UI Location	Integrations > Site Management Tools
JSON file	CMS.json

Landing Pages URL

This property applies to:

- SuiteCommerce Advanced - Elbrus and later
- SuiteCommerce

This string enables you to configure the base URL for Site Management Tools landing pages. By default this property is blank, and landing pages use the URL of the web store. When working in a sandbox account, set the property to the URL for sandbox, for example, <https://system.sandbox.netsuite.com>.

ID	cms.baseUrl
UI Location	Integrations > Site Management Tools
JSON file	CMS.json

CMS Adapter Version

This string specifies the CMS Adapter version (2 or 3) to be used by Site Management Tools. The default value for this field is 3. If you are implementing custom content types (CCTs) on your site, you must set this property to 3.

More information: [Site Management Tools](#)

ID	cms.adapterVersion
UI Location	Integrations > Site Management Tools
JSON file	CMS.json

Time to Wait for CMS Content (ms)

This string specifies the maximum time in milliseconds to wait for CMS content to render. If this maximum time is exceeded, the app renders then CMS content renders later. You might also have a flickering problem. The default value for this field is 200.

More information: [Site Management Tools](#)

ID	cms.contentWait
UI Location	Integrations > Site Management Tools
JSON file	CMS.json

Twitter Subtab

These settings configure Twitter popup settings.

More Information: [Twitter Product Cards](#)

Enable

This boolean specifies if the Twitter addThis property is enabled.

ID	twitter.enable
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter Popup Status

This string specifies if the Twitter popup is enabled.

ID	twitter.popupOptions.status
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter Popup Resizeable

This string specifies if the Twitter popup is resizable.

ID	twitter.popupOptions.resizable
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter Popup Scrollbars

This string specifies if the Twitter popup displays scroll bars.

ID	twitter.popupOptions.scrollbars
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter Popup Personalbar

This string specifies if the Twitter popup displays a personal bar.

ID	twitter.popupOptions.personalbar
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter Popup Directories

This string specifies if the Twitter popup displays a directories bar.

ID	twitter.popupOptions.directories
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter Popup Location

This string specifies if the Twitter popup displays a location bar.

ID	twitter.popupOptions.location
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter Popup Toolbar

This string specifies if the Twitter popup displays a tool bar.

ID	twitter.popupOptions.toolbar
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter Popup Menubar

This string specifies if the Twitter popup displays a menu bar.

ID	twitter.popupOptions.menubar
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter Popup Width

This number specifies the width (in pixels) of the Twitter popup.

ID	twitter.popupOptions.width
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter Popup Height

This number specifies the height (in pixels) of the Twitter popup.

ID	twitter.popupOptions.height
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter Popup Left

This number specifies the left offset (in pixels) of the Twitter popup.

ID	twitter.popupOptions.left
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter Popup Top

This number specifies the top offset (in pixels) of the Twitter popup.

ID	twitter.popupOptions.top
-----------	--------------------------

UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Twitter VIA

This string specifies the Twitter account of the original tweeter. For example: @MerchantName.

ID	twitter.popupOptions.via
UI Location	Integrations > Twitter
JSON file	twitter.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Layout Tab

The settings on this tab let you configure proper ties related to your site layout. This tab includes the following subtabs:

- Bottom Banner Images Subtab
- Carousel Images Subtab
- Color Palettes Subtab
- Cookies Warning Banner Subtab
- Footer Subtab
- Header Subtab
- Images Subtab
- Light Colors Subtab
- Navigation Subtab

Bottom Banner Images Subtab

This array specifies the banner images that appear at the bottom of the application. Each banner image requires the **URL** property. This string defines the URL of the bottom banner image. The order in the array specifies the order in the banner.

 **Note:** SuiteCommerce Advanced developers on the Kilimanjaro release or earlier: Three default images are distributed with the SCA bundle. These images are located in the ShoppingApplication module. You can override these default images, but the images must be in the file cabinet.

ID	home.bottomBannerImages
UI Location	Layout > Bottom Banner Images
JSON file	Home.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Carousel Images Subtab

This array specifies the carousel images that appear at the top of the SuiteCommerce home page. Each carousel image requires the **URL** property. This string defines the URL of the carousel image. The order in the array specifies the order in the carousel.

Note: The three default images are distributed with the SCA bundle, located in the ShoppingApplication module. You can override these, but the images must be in the file cabinet at Web Site Hosting Files > Live Hosting Files > SSP Applications > [SSP Application] > Development > img.

See: [File Types Recognized in the File Cabinet](#) for a list of supported image file types.

ID	home.carouselImages
UI Location	Layout > Carousel Images
JSON file	Home.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Color Palettes Subtab

These settings define the default color palette or let you create a new one. A color palette maps a color defined in NetSuite to a CSS color value. A color palette maps a color label to its hexadecimal value.

More Information: [Faceted Navigation](#)

Each color palette contains the following:

- **Palette ID** (string) – specifies the internal id for the color. The default color palette is called default. When creating a custom color palette, use only number values from 1 to 20.
- **Color Name** (string) – specifies the name of the color.
- **Color Value** (string/object definition) – specifies the hexadecimal value of the color or an object definition of an image.

In some cases, you may want a color option that cannot be generated from a hexadecimal value, such as an image of fabric. In these cases, you can define the image as an object in the **Color Value** instead of a hexadecimal value. In this object definition, define the following:

- **type**: as image
- **src**: as the URL of the desired image
- **width** and **height**: as pixels

Example: `{type: 'image', src: '/siteImgFolder/colorImage.png', width: 29, height: 29}.`

IDs	layout.ColorPalette layout.ColorPalette.paletteId layout.ColorPalette.colorName layout.ColorPalette.colorValue layout.ColorPalette.imgsrc layout.ColorPalette.imgheight layout.ColorPalette.imgwidth
IDs (pre-Elbrus)	facetsColorPalette

Note: These properties appear in the Shopping Catalog tab and Facet Color Palettes subtab in Vinson implementations of SCA.	facetsColorPalette.paletteId facetsColorPalette.colorName facetsColorPalette.colorValue
UI Location	Layout > Color Palettes
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Cookies Warning Banner Subtab

These settings configure the cookie warning banner. To display this banner, you must enable the Show Cookie Consent Banner property in NetSuite.

ID	cookieWarningBanner
UI Location	Layout > Cookie Warning Banner
JSON file	Cookies.json
Configuration file (pre-Vinson)	SC.Configuration.js

User May Dismiss Banner

This boolean specifies if the user can close the cookie warning banner.

ID	cookieWarningBanner.closable
UI Location	Layout > Cookie Warning Banner
JSON file	Cookies.json
Configuration file (pre-Vinson)	SC.Configuration.js

Save in Cookie

This boolean specifies if the value of the **User May Dismiss Banner** property is stored in a browser cookie.

ID	cookieWarningBanner.saveInCookie
UI Location	Layout > Cookie Warning Banner
JSON file	Cookies.json
Configuration file (pre-Vinson)	SC.Configuration.js

Anchor Text

This string specifies the text used in the link to a modal dialog whose content is defined in the **Message** property. The default value of this property is **Learn More**.

ID	cookieWarningBanner.anchorText
-----------	--------------------------------

UI Location	Layout > Cookie Warning Banner
JSON file	Cookies.json
Configuration file (pre-Vinson)	SC.Configuration.js

Message

This string specifies the text of the modal dialog. This text is displayed when the user clicks the link defined in the **Anchor Text** property.

ID	cookieWarningBanner.message
UI Location	Layout > Cookie Warning Banner
JSON file	Cookies.json
Configuration file (pre-Vinson)	SC.Configuration.js

Footer Subtab

This subtab moved to the **Legacy** tab in the **2018.2** release of SuiteCommerce. See [Footer Subtab](#) for more information on the properties related to Footer.

Header Subtab

These settings specify properties of the application header.

Hide Currency Selector

This boolean hides or displays the currency selector in the application. Enable this property to hide the selector.

ID	header.notShowCurrencySelector
UI Location	Layout > Header
JSON file	Header.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

Logo URL

This string specifies the location of the logo image file that is displayed at the top of the application.

ID	header.logoUrl
UI Location	Layout > Header
JSON file	Header.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

Images Subtab

These settings configure general properties for site images.

Image Not Available URL

This string specifies the URL to the default image displayed when an item image is not configured.

ID	imageNotAvailable
UI Location	Layout > Images
JSON file	ApplicationSkeleton.Layout.Images.json
Configuration file (pre-Vinson)	SC.Configuration.js

Light Colors Subtab

This array specifies color labels in the color pallet that have a similar hue to the background color. The color picker for each of these colors is displayed with an outer border.

ID	layout.lightColors
ID (Pre-Elbrus)	lightColors
Note: In the Vinson release of SCA, this property is located on the Shopping Catalog tab and Light Colors subtab.	
UI Location	Layout > Light Colors
JSON file	ApplicationSkeleton.Layout.Colors.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Navigation Subtab

These settings configure defaults for the main navigational links of the application (Home, Shop, etc.) used to construct site navigation.

This array specifies the following for each link:

- **Text** (string) – specifies the link text displayed in the user interface.
- **HREF** (string) – specifies the href attribute of the navigation link. For example: **/search** results in navigation to the search results page. This is ignored for parent entries.
- **Level** (string) – specifies the hierarchical level of this navigation link.
- **data-touchpoint** (string) – specifies the touch point of the navigational link, if applicable.
- **data-hashtag** (string) – specifies the data hashtag of the link.
- **Class names** (string) – specifies any additional HTML class names added to the link.
- **ID** (string) – specifies the internal identifier of the link. Parent entries must have an ID for children to reference.

- **Parent ID** (string) – specifies the parent ID of the entry if it is a child view.

IDs	navigationData navigationData.text navigationData.href navigationData.level navigationData.dataTouchpoint navigationData.dataHashtag navigationData.classnames navigationData.id navigationData.parentId
UI Location	Layout > Navigation
Configuration file (pre-Vinson)	SC.Configuration.js

Legacy Tab

- Newsletter Subtab
- Footer Subtab

This tab provides access to properties that may transition to Site Management Tools or Extensions.

The 2018.1 release of SuiteCommerce includes the templates necessary to access the configuration properties located on the Legacy tab subtabs. If your implementation of SuiteCommerce began with the 2018.2 release or later, you do not have the templates necessary to access these properties and you should not use this tab.

Newsletter Subtab

In SuiteCommerce Advanced Aconcagua and earlier, this subtab is located under the **Shopping** tab. In SuiteCommerce Advanced 2018.2 and later, this subtab is not available.

This feature and its settings require:

- SuiteCommerce Advanced – Vinson, Elbrus, Kilimanjaro, or Aconcagua.
- SuiteCommerce 2018.1 release.

The following settings specify properties for the Newsletter opt-in feature.

When a user submits an email address for a newsletter and the SuiteScript API fails to return any matching Customer or Lead records, SuiteCommerce creates a new Lead record in NetSuite. SCA then sets this new record's **Global Subscription Status** field to Soft Opt-In. However, in these situations, NetSuite requires populating the new Lead record's **Name** or **Company Name** fields with information. You configure the value of these entries. The default value for each field is **unknown**.

When a user submits an email address for a newsletter and the SuiteScript API fails to return any matching Customer or Lead records, SuiteCommerce creates a new Lead record in NetSuite. SCA then sets this new record's **Global Subscription Status** field to **Full Subscription (Not opted in; send all messages)**. However, in these situations, NetSuite requires populating the new Lead record's **Name** or **Company Name** fields with information. You configure the value of these entries. The default value for each field is **unknown**.

More Information: [Newsletter](#)

Generic First Name

This string specifies the generic first name to populate in a new Lead record when a user opts in for the newsletter. This satisfies the record's requirement to populate this field.

ID	newsletter.genericFirstName
UI Location	Legacy > Newsletter
UI Location (Aconcagua and Earlier)	Shopping > Newsletter
JSON file	NewsLetter.json

Generic Last Name

This string specifies the generic last name to populate in a new Lead record when a user opts in for the newsletter. This satisfies the record's requirement to populate this field.

ID	newsletter.genericLastName
UI Location	Legacy > Newsletter
UI Location (Aconcagua and Earlier)	Shopping > Newsletter
JSON file	NewsLetter.json

Company Name

This string specifies the generic company name to populate in a new Lead record when a user opts in for the newsletter. This satisfies the record's requirement to populate this field.

ID	newsletter.company
UI Location	Legacy > Newsletter
UI Location (Aconcagua and Earlier)	Shopping > Newsletter
JSON file	NewsLetter.json

Footer Subtab

These settings configure the footer navigation links.

Footer Navigation

This array specifies the navigation links that appear in the footer. You can edit the default links or create new ones.

Each link requires the following properties:

- **Text** (string) – defines the text of the link that appears in the footer.
- **URL** (string) – defines the target URL of the link.

IDs	footer.navigationLinks footer.navigationLinks.text footer.navigationLinks.href
UI Location	Legacy > Footer
UI Location (Aconcagua and Earlier)	Layout > Footer
JSON file	footer.json
Configuration file (pre-Vinson)	SC.Configuration.js

Multi-Domain Tab

The settings on this tab let you configure properties related to multiple domains. This tab includes the following subtabs:

- [Hosts Subtab](#)
- [Translations Subtab](#)

Hosts Subtab

These settings configure information about each host for websites with multiple languages or regions. You must define a host for each language.

More Information: [Localization](#)

List of Hosts

Specifies the host for each translated site. Use the following list to add your hosts by ID. For example, you can configure two different hosts. You set one for English and U.S. Dollars only. You set the remaining site for English and French with two currencies, U.S. Dollar and Euros.



Important: Each language must have its own domain.

Each host requires the following:

- **Host ID** – specifies the host address for the translated content.
- **Language** – specifies the languages available for the associated host location.
- **Language Title** – specifies the display label of the currency within the Location dropdown selector on the website.
- **Language Domain** – specifies the domain name for this language. This must exactly match the hosted domain for the associated language.
- **Currency** – specifies the display label of the currency within the Location dropdown selector on the website.
- **Subsidiary** – specifies the subsidiary for the host.
- **Location** – specifies the location for the host.

IDs	multiDomain.hosts
------------	-------------------

	multiDomain.hosts.host multiDomain.hosts.language multiDomain.hosts.languageTitle multiDomain.hosts.languageDomain multiDomain.hosts.currency multiDomain.hosts.subsidiary multiDomain.hosts.location
UI Location	Multi-Domain > Hosts
ID (pre-Vinson)	hosts
JSON file	Languages.json
Configuration file (pre-Vinson)	Configuration.js

Hosts Array Parameters (pre-Vinson)

In pre-Vinson implementations, the **hosts** array, located in the backend Configuration.js file, maps languages and currencies to specific host locations. This array requires a **title** property and both the **currencies** and **languages** arrays, as defined below.

i Note: The **hosts** array contains sample code that is hidden within a multi-line comment block. Removing the comment tags reveals this code, but the properties require customization unique to your site as defined below.

- **title** (string) – specifies the display label of the location within the Location drop-down selector on the website.
- **currencies** (array) – specifies the currencies available for the associated host location.
 - **title** (string) – specifies the display label of the currency within the Location drop-down selector on the website.
 - **code** (string) – specifies the code for the location. This maps the currency code with the specific host location and must exactly match the three-letter International Standards Organization (ISO) currency code symbol defined in the backend, or the currency will not appear in the list. See the help topic [Currency Management](#) for more information.
- **languages** (array) – specifies the languages available for the associated host location
 - **title** (string) – specifies the display label of the currency within the Location drop-down selector on the website.
 - **host** (string) – specifies the host address for the translated content. This must exactly match the hosted domain for the associated language. For more information on setting up domains within SuiteCommerce, see the help topic [Domains](#)
 - **locale** (string) – specifies the country code and culture code for the associated language per ISO standards (ISO 639 and ISO 3166 respectively). This must match the locale property defined in the backend, or the language will not appear in the list. Example: en_US.

Your website must have a unique locale for each hosted domain and a unique domain for each language.

IDs (pre-Vinson)	hosts hosts.title hosts.currencies hosts.currencies.title hosts.currencies.code hosts.languages
-------------------------	--

	hosts.languages.title hosts.languages.host hosts.languages.locale
UI Location	None
Configuration file (pre-Vinson)	Configuration.js

Translations Subtab

These settings configure multiple domains for a website to enable multiple languages or regions.

List of Translations

This array specifies new string translations or updates for existing strings. NetSuite recommends using this method to make small changes to the translated content on your site. If making significant changes to translated content on your site, see the help topic [The Translation Process](#) for details.

Each translation array contains the following properties:

- **Key** (string) – specifies the string being translated.
- **Language** (string) – defines the localized string for each locale.

More Information: [Localization](#)

IDs	extraTranslations extraTranslations.key extraTranslations.cs_CZ (Czech Republic) extraTranslations.da_DK (Danish) extraTranslations.de_DE (German) extraTranslations.en_US (English – US) extraTranslations.es_AR (Spanish – Argentina) extraTranslations.es_ES (Spanish – Spain) extraTranslations.fr_CA (French – Canada) extraTranslations.fr_FR (French – France) extraTranslations.it_IT (Italian) extraTranslations.ja_JP (Japanese) extraTranslations.ko_KR (Korean) extraTranslations.nl_NL (Dutch – Netherlands) extraTranslations.pt_BR (Portuguese – Brazil) extraTranslations.ru_RU (Russian) extraTranslations.sv_SE (Swedish) extraTranslations.th_TH (Thai) extraTranslations.tr_TR (Turkish) extraTranslations.zh_CN (Chinese) extraTranslations.zh_TW (Chinese – Taiwan)
UI Location	Multi-Domain > Translations
JSON file	Translations.json

My Account Tab

The settings on this tab let you configure properties related to the My Account application. This tab includes the following subtabs:

- Addresses Subtab
- Cases Subtab
- List Header Subtab
- Overview Subtab
- Quotes Subtab
- Return Authorization Subtab
- SCIS Integration Subtab
- Transaction List Columns Subtab

Addresses Subtab

These settings configure properties related to addresses entered in My Account pages.

Require a Phone Number With Addresses

This boolean determines if users are required to include a phone number when saving an address to their account. If this property is checked, users must include a phone number when entering an address. If this property is unchecked, no phone number is required. This property is unchecked by default.

ID	addresses.isPhoneMandatory
UI Location	My Account > Addresses
JSON file	Address.json

Cases Subtab

These settings configure the initial required default case status values.

Status Start ID

This string specifies the NetSuite status SuiteCommerce uses to indicate that a case has been started. Options include:

- 1 – Not Started
- 2 – In Progress
- 3 – Escalated
- 4 – Re-opened
- 5 – Closed

ID	cases.defaultValues.statusStart.id
UI Location	My Account > Cases
JSON file	Case.json
Configuration file (pre-Vinson)	Configuration.js

Status Close ID

This string specifies the NetSuite status SuiteCommerce uses to indicate the case has been closed. Options include:

- 1 – Not Started
- 2 – In Progress
- 3 – Escalated
- 4 – Re-opened
- 5 – Closed

ID	cases.defaultValues.statusClose.id
UI Location	My Account > Cases
JSON file	Case.json
Configuration file (pre-Vinson)	Configuration.js

Origin ID

This string specifies the internal identifier of the case origin.

ID	cases.defaultValues.origin.id
UI Location	My Account > Cases
JSON file	Case.json
Configuration file (pre-Vinson)	Configuration.js

List Header Subtab

These settings specify properties that determine how lists of data and collections are displayed. The ListHeader module uses these properties.

Filter Range Quantity Days

This integer specifies the number of days within the date range used when displaying data.

ID	listHeader.filterRangeQuantityDays
UI Location	My Account > List Header
JSON file	ListHeader.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js

Overview Subtab

These settings configure the My Account page settings.

Customer Support URL

This string specifies the URL for your customer support page.

ID	overview.customerSupportURL
UI Location	My Account > Overview
JSON file	Overview.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js

Recent Orders Quantity to Show

This number specifies the quantity of recent orders displayed in the My Account Overview page. By default, the last three orders are displayed.

ID	overview.homeRecentOrdersQuantity
UI Location	My Account > Overview
JSON file	Overview.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js

Home Banners

This array specifies the banner to display in the index of My Account. You can configure this to customize promotions for clients. Each banner requires the following properties:

- **Image URL** (string) – specifies the source image and URL.
- **Link URL** (string) – specifies the URL link when a user clicks the image.
- **Link Target** (string) – specifies the link target.

IDs	overview.homeBanners overview.homeBanners.imageSource overview.homeBanners.linkURL overview.homeBanners.linkTarget
UI Location	My Account > Overview
JSON file	Overview.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js

Quotes Subtab

These settings configure the behavior of the Quotes feature in SuiteCommerce.

More Information: [Quotes](#)

Show Request A Quote Hyperlink

This property applies to:

- SuiteCommerce Advanced - 2018.2 and later
- SuiteCommerce

This boolean specifies whether the Request a Quote hyperlink displays. Select to display the Request a Quote hyperlink. The default value is: true.

ID	quote.showHyperlink
UI Location	My Account > Quotes
JSON file	RequestQuoteAccessPoint.json

Hyperlink Text

This property applies to:

- SuiteCommerce Advanced - 2018.2 and later
- SuiteCommerce

This string specifies the text of the Request a Quote hyperlink. The default value is: Request a Quote.

ID	quote.textHyperlink
UI Location	My Account > Quotes
JSON file	RequestQuoteAccessPoint.json

Purchase Ready Status ID

This string specifies the Customer Status that enables a quote to be purchased. This field must match the Internal ID of the Customer Status required for that customer to be able to make a purchase from the quote.

In NetSuite, the Estimate record's Status field lists all available Customer Statuses for the quote. When the merchant manually changes this field to the Customer Status whose Internal ID matches the **Purchase Ready Status ID**, all website links to Review or Purchasing pages are enabled. The user can then make a purchase.

More Information: [Set the Customer Internal ID](#)

ID	quote.purchaseReadyStatusId
UI Location	My Account > Quotes
ID (pre-Vinson)	purchase_ready_status_id
JSON file	Quote.json
Configuration file (pre-Vinson)	Configuration.js

Invoice Form ID

This string specifies Internal ID of the invoice form that SCA uses to generate a sales order using terms.

More Information: [Define an Invoice Form](#)

ID	quoteToSalesorderWizard.invoiceFormId
UI Location	My Account > Quotes
ID (pre-Vinson)	invoice_form_id
JSON file	QuoteSalesorderWizard.json
Configuration file (pre-Vinson)	Configuration.js

Days to Expiration

This value specifies the number of days you want all quotes to remain valid after initial submittal. If this value is 0, the number of days to expiration is determined by the value in the **Default Quote Expiration (In Days)** field on the Sales Preferences record. See the help topic [Sales Force Automation Preferences](#). All quotes use the same expiration configuration. After this time expires, the quote can no longer become a purchase.

More Information: [Set Quote Expiration](#)

ID	quote.daysToExpire
UI Location	My Account > Quotes
ID (pre-Vinson)	days_to_expire
JSON file	Quote.json
Configuration file (pre-Vinson)	Configuration.js

Disclaimer Summary

This string specifies the sales representative disclaimer summary that appears under the Order Summary area of the Quote Details page. This message appears if no sales representative is assigned to the quote. What appears on your site depends on how your sales representative information is set up in your account.

More Information: [Customize Sales Representative Information](#)

ID	quote.disclaimerSummary
UI Location	My Account > Quotes
JSON file	Quote.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js

Disclaimer

This string specifies the sales representative information disclaimer that appears at the bottom of the Quote Details page. This message appears if no sales representative is assigned to the quote. What appears on your site depends on how your sales representative information is set up in your account.

More Information: [Customize Sales Representative Information](#)

ID	quote.disclaimer
UI Location	My Account > Quotes
JSON file	Quote.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js

Default Phone Number for Sales Rep

This string specifies the default sales representative phone number. This phone number appears on the Quote Details page if a sales associate is assigned to the quote, but the phone number has not been assigned. What appears on your site depends on how your sales representative information is set up in your account.

More Information: [Customize Sales Representative Information](#)

ID	quote.defaultPhone
UI Location	My Account > Quotes
JSON file	Quote.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js

Default Email for Sales Rep

This string specifies the default sales representative Email address. This email address appears on the Quote Details page if a sales associate is assigned to the quote, but the email address has not been assigned. What appears on your site depends on how your sales representative information is set up in your account.

More Information: [Customize Sales Representative Information](#)

ID	quote.defaultEmail
UI Location	My Account > Quotes
JSON file	Quote.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js

Return Authorization Subtab

These settings configure properties related to the Return Authorization feature.

Cancel URL Root

This string specifies a custom domain used to override the domain used to cancel a return authorization.

ID	returnAuthorization.cancelUrlRoot
-----------	-----------------------------------

UI Location	My Account > Return Authorization
JSON file	ReturnAuthorization.json
Configuration file (pre-Vinson)	Configuration.js

Return Authorizations Reasons

This array defines the return reasons that are available to the user. Each reason specifies the following properties:

- **Reason** (string) – specifies the text of the return reason.
- **ID** (number) – specifies the internal ID of the return reason. Each item must have a unique ID.
- **Order** (number) – specifies the order of the return reason in a drop down list.
- **Is Other** (boolean) – specifies if the return reason is Other.

IDs	returnAuthorization.reasons.text returnAuthorization.reasons.id returnAuthorization.reasons.order returnAuthorization.reasons.isOther
UI Location	My Account > Return Authorization
JSON file	ReturnAuthorization.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js

SCIS Integration Subtab

Is SCIS Integration Enabled

This boolean enables or disables SCIS integration with your SuiteCommerce site.

ID	isSCISIntegrationEnabled
UI Location	My Account > SCIS Integration
JSON file	ReturnAuthorization.json
Configuration file (pre-Vinson)	Configuration.js

Location Type Mapping Store Internal ID

This string specifies the location-type identifier that SuiteCommerce uses to associate a transaction with an in-store purchase. 1 = store, 2 = online.

ID	locationTypeMapping.store.internalid
UI Location	My Account > SCIS Integration
JSON file	ReturnAuthorization.json

Configuration file (pre-Vinson)	Configuration.js
--	------------------

Location Type Mapping Store Name

This string specifies a descriptive text associated with the locationTypeMapping.store.internalid property.

ID	locationTypeMapping.store.name
UI Location	My Account > SCIS Integration
JSON file	ReturnAuthorization.json
Configuration file (pre-Vinson)	Configuration.js

Transaction Record Origin Mapping

This array specifies frontend properties related to what is displayed in a user's account. Each transaction record origin mapping object requires the following properties:

- **ID** (string) – specifies the origin of the purchase record (backend, instore, online).
- **Origin** (number) – specifies frontend properties related to what is displayed in a user's account. This is related to the locationTypeMapping.store.internalid property.
- **Name** (string) – specifies the text displayed in the Origin column of a user's My Purchases list.
- **Detailed Name** (string) – specifies the text displayed in the Purchase Details page.

IDs	transactionRecordOriginMapping transactionRecordOriginMapping.id transactionRecordOriginMapping.origin transactionRecordOriginMapping.name transactionRecordOriginMapping.detailedName
UI Location	My Account > Transaction Record Origin Mapping
JSON file	MyAccount.json
Configuration file (pre-Vinson)	SC.MyAccount.Configuration.js

Transaction List Columns Subtab

The **Transaction List Columns** subtab lets you configure how transaction column fields appear in the My Account area of your site. When you enable column management for a list, you can specify what fields appear.

You can configure the following transaction lists:

- Order History
- Quotes
- Returns
- Open Invoices
- Paid Invoices



Note: If you enable column management for a list, the configuration settings on the **Transaction List Columns** subtab take precedence over any other configuration. For example, if you are integrating SCIS with your site and enable columns management using the **Transaction List Columns** subtab, your My Account transaction columns render according to these settings. To retroactively add any columns from your SCIS integration, you create them manually as described below.

More information: [My Account Transaction Lists](#)

Enable Return Authorization Columns Management

This boolean enables or disables column management for the **Returns** transaction list. If this property is checked, the MyAccount application uses the properties set in the **Transaction List Columns Return Authorization** table to render the Return Authorization list on your site. This property is disabled by default.

ID	transactionListColumns.enableReturnAuthorization
UI Location	My Account > Transaction List Columns
JSON file	MyAccountReturnAuthorizationListsColumns.json

Enable Quotes Columns Management

This boolean enables or disables column management for the **Quotes** transaction list. If this property is checked, the MyAccount application uses the properties set in the **Transaction List Columns Quote** table to render the Quotes list on your site. This property is disabled by default.

For this column to appear, you must enable the Quotes feature for your account. See the help topic [Quotes](#) for more information.

ID	transactionListColumns.enableQuote
UI Location	My Account > Transaction List Columns
JSON file	MyAccountQuoteListsColumns.json

Enable Order History Columns Management

This boolean enables or disables column management for the **Recent Purchases** transaction list. If this property is checked, the MyAccount application uses the properties set in the **Transaction List Columns Order History** table to render the Purchase History list on your site. This property is disabled by default.

ID	transactionListColumns.enableOrderHistory
UI Location	My Account > Transaction List Columns
JSON file	MyAccountOrderHistoryListsColumns.json

Enable Invoice Columns Management

This boolean enables or disables column management for the Open Invoices transaction list. If this property is checked, the MyAccount application uses the properties set in the **Transaction List Columns**

Open Invoices and **Transaction List Columns Paid Invoices** tables to render the Open and Paid Invoices list on your site. This property is disabled by default.

ID	transactionListColumns.enableInvoice
UI Location	My Account > Transaction List Columns
JSON file	MyAccountInvoiceListsColumns.json

Transaction List Columns Return Authorization

This array lets you add columns to the **Returns** transaction list. The order of the columns in this table determines the order they render on your site (top-down, left-to-right).

Add a line for each field you intend to display on your site. The ID and Label properties are strings. The **ID** property must match the Field ID of the field you want to render. The default properties are:

- trandate
- quantity
- amount_formatted

 **Note:** The **Return No.** column renders by default and cannot be configured.

ID	transactionListColumns.returnAuthorization transactionListColumns.returnAuthorization.id transactionListColumns.returnAuthorization.label
UI Location	My Account > Transaction List Columns
JSON file	MyAccountReturnAuthorizationListsColumns.json

Transaction List Columns Quote

This array lets you add columns to the **Quotes** transaction list. The order of the columns in this table determines the order they render on your site (top-down, left-to-right).

Add a line for each field you intend to display on your site. The ID and Label properties are strings. The **ID** property must match the Field ID of the field you want to render. The default properties are:

- trandate
- duedate
- total_formatted

 **Note:** The **Quote No.** column renders by default and cannot be configured.

ID	transactionListColumns.quote transactionListColumns.quote.id transactionListColumns.quote.label
UI Location	My Account > Transaction List Columns

JSON file	MyAccountQuoteListsColumns.json
------------------	---------------------------------

Transaction List Columns Order History

This array lets you add columns to the **Order History** transaction list. The order of the columns in this table determines the order they render on your site (top-down, left-to-right).

Add a line for each field you intend to display on your site. The ID and Label properties are strings. The **ID** property must match the Field ID of the field you want to render. The default properties are:

- date
- amount
- status

 **Note:** The **Purchase No.** and **Track Items** columns render by default and cannot be configured.

ID	transactionListColumns.orderHistory transactionListColumns.orderHistory.id transactionListColumns.orderHistory.label
UI Location	My Account > Transaction List Columns
JSON file	MyAccountOrderHistoryListsColumns.json

Transaction List Columns Open Invoices

This array lets you add columns to the **Open Invoices** transaction list. The order of the columns in this table determines the order they render on your site (top-down, left-to-right).

Add a line for each field you intend to display on your site. The ID and Label properties are strings. The **ID** property must match the Field ID of the field you want to render. The default properties are:

- duedate
- trandate
- amount

 **Note:** The **Invoice No.** column renders by default and cannot be configured.

ID	transactionListColumns.invoiceOpen transactionListColumns.invoiceOpen.id transactionListColumns.invoiceOpen.label
UI Location	My Account > Transaction List Columns
JSON file	MyAccountInvoiceListsColumns.json

Transaction List Columns Paid Invoices

This array lets you add columns to the **Paid Invoices** transaction list. The order of the columns in this table determines the order they render on your site (top-down, left-to-right).

Add a line for each field you intend to display on your site. The ID and Label properties are strings. The **ID** property must match the Field ID of the field you want to render. The default properties are:

- trandate
- closedate
- amount



Note: The **Invoice No.** column renders by default and cannot be configured.

ID	transactionListColumns.invoicePaid transactionListColumns.invoicePaid.id transactionListColumns.invoicePaid.label
UI Location	My Account > Transaction List Columns
JSON file	MyAccountInvoiceListsColumns.json

Search Tab

The settings on this tab let you configure properties related to site search. This tab includes the following subtabs:

- [Result Display Options Subtab](#)
- [Result Sorting Subtab](#)
- [Search Results Subtab](#)
- [Search Results per Page Subtab](#)
- [Type Ahead Subtab](#)

Result Display Options Subtab

These settings specify display options for search results, grouped by browser type (desktop, phone, and tablet).

Each display option contains the following properties (note the different property IDs for each browser type):

- **ID** (string) – specifies the internal identifier of the display option.
- **Name** (string) – specifies the label of the display option.
- **Template** (string) – specifies the template used to display the search result option.
- **Columns** (number) – specifies the number of columns used to display search results.
- **Icon** (string) – specifies the icon for the display option.
- **Is Default** (boolean) – specifies that the display option is the default.

Desktop

This array specifies the display options for search results for desktop browsers.

IDs	itemsDisplayOptions itemsDisplayOptions.id itemsDisplayOptions.name itemsDisplayOptions.template itemsDisplayOptions.columns itemsDisplayOptions.icon itemsDisplayOptions.isDefault
UI Location	Search > Result Display Options
JSON file	SearchItemDisplayOptions.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Phone

This array specifies the display options for search results for phone browsers.

IDs	itemsDisplayOptionsPhone itemsDisplayOptionsPhone.id itemsDisplayOptionsPhone.name itemsDisplayOptionsPhone.template itemsDisplayOptionsPhone.columns itemsDisplayOptionsPhone.icon itemsDisplayOptionsPhone.isDefault
UI Location	Search > Result Display Options
JSON file	SearchItemDisplayOptions.json

Tablet

This array specifies the display options for search results for tablet browsers.

IDs	itemsDisplayOptionsTablet itemsDisplayOptionsTablet.id itemsDisplayOptionsTablet.name itemsDisplayOptionsTablet.template itemsDisplayOptionsTablet.columns itemsDisplayOptionsTablet.icon itemsDisplayOptionsTablet.isDefault
UI Location	Search > Result Display Options
JSON file	SearchItemDisplayOptions.json

Result Sorting Subtab

These settings specify the sorting options available in the Sort By drop down menu, grouped by browser type (desktop, phone, or tablet). Sort fields must first be defined in the Web Site Setup record. For details on defining sort fields, see the help topic [Select and Configure Sort Fields](#).

Each sort property contains the following (note the different property IDs for each browser type):

- **ID** (string) – specifies the ID of the search option.

- **Name** (string) – specifies the label used to identify the sort option.
- **Is Default** (boolean) – specifies the default active search option.

Desktop

This array specifies the sorting options available in the Sort By drop down menu on desktop browsers.

IDs	sortOptions sortOptions.id sortOptions.name sortOptions.isDefault
UI Location	Search > Result Sorting
JSON file	SearchSort.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Phone

This array specifies the sorting options available in the Sort By drop down menu on phone browsers.

IDs	sortOptionsPhone sortOptionsPhone.id sortOptionsPhone.name sortOptionsPhone.isDefault
UI Location	Search > Result Sorting
JSON file	SearchSort.json

Tablet

This array specifies the sorting options available in the Sort By drop down menu on tablet browsers.

IDs	sortOptionsTablet sortOptionsTablet.id sortOptionsTablet.name sortOptionsTablet.isDefault
UI Location	Search > Result Sorting
JSON file	SearchSort.json

Search Results Subtab

These settings configure properties related to the search feature in the application header.

More Information: [Select and Configure Sort Fields](#)

Search Results Title

This string specifies the title for the facet browse view that appears in the application header.

ID	defaultSearchTitle
UI Location	Search > Search Results
JSON file	SearchResults.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Search Keyword Maximum Length

This number specifies the maximum number of characters the user can enter in the search field.

ID	searchPrefs.maxLength
UI Location	Search > Search Results
JSON file	SearchResults.json
Configuration file (pre-Vinson)	SC.Configuration.js

Search Results per Page Subtab

These settings specify the options that appear in the Results Per Page drop down menu. This array contains the following properties:

- **Items** (number) – specifies the number of search results displayed by this option.
- **Name** (string) – specifies the label for this option.
- **Is Default** (boolean) – specifies that this option is the default.

IDs	resultsPerPage resultsPerPage.items resultsPerPage.name resultsPerPage.isDefault
UI Location	Search > Search Results per Page
JSON file	SearchResultsPerPage.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Type Ahead Subtab

These settings define configuration properties for the Type Ahead feature. This feature causes SuiteCommerce to begin returning search results while the user is entering text in the search text box.

Min Length

This number specifies the number of characters the user must enter before beginning the search.

ID	typeahead.minLength
UI Location	Search > Type Ahead

JSON file	SearchTypeAhead.json
Configuration file (pre-Vinson)	SC.Configuration.js

Maximum Results

This number specifies the maximum number of search results that are returned.

ID	typeahead.maxResults
UI Location	Search > Type Ahead
JSON file	SearchTypeAhead.json
Configuration file (pre-Vinson)	SC.Configuration.js

Sort

This string specifies the sort order of items returned by the Item Search API. By default, items are sorted by relevance in ascending order. However, you can sort by other criteria like the name property.

ID	typeahead.sort
UI Location	Search > Type Ahead
JSON file	SearchTypeAhead.json
Configuration file (pre-Vinson)	SC.Configuration.js

Shopping Tab

The settings on this tab let you configure properties related to the Shopping application and includes the following subtabs:

- [Item Options Subtab](#)
- [Newsletter Subtab](#)
- [Quick Order Subtab](#)
- [Reviews Subtab](#)
- [Wishlist Subtab](#)

Item Options Subtab

This subtab moved to the **Shopping Catalog** tab in the **Elbrus** release of SuiteCommerce Advanced. See [Item Options Subtab](#) for more information on the properties related to Item Options.

Newsletter Subtab

This subtab moved to the **Legacy** tab in the **2018.2** release of SuiteCommerce. See [Newsletter Subtab](#) for more information on the properties related to Newsletter.

Quick Order Subtab

ⓘ Applies to: SuiteCommerce Web Stores

These settings configure properties related to Quick Orders.

More Information: [Quick Order](#)

This feature and its settings require:

- SuiteCommerce Advanced - 18.2 or later
- SuiteCommerce

Show Quick Order Hyperlink

This boolean specifies whether the Quick Order hyperlink displays. Select to display the Quick Order hyperlink. The default value is: true.

ID	quickOrder.showHyperlink
UI Location	Shopping > Quick Order
JSON file	QuickOrder.json

Hyperlink Text

This string specifies the text of the Quick Order hyperlink. The default value is: Quick Order

ID	quickOrder.textHyperlink
UI Location	Shopping > Quick Order
JSON file	QuickOrder.json

Reviews Subtab

These settings configure properties related to product reviews.

More Information: [Product Reviews](#)

Max Flag Count

This number specifies the number of flags a review must receive before being marked as flagged by users.

ID	productReviews.maxFlagsCount
UI Location	Shopping > Reviews
JSON file	ProductReviews.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Login Required

This boolean specifies if users must be logged in to create a review.

ID	productReviews.loginRequired
UI Location	Shopping > Reviews
JSON file	ProductReviews.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Flagged Status

This number specifies the number of times a review is flagged. If the value of the productReviews.maxFlagsCount property is reached, then this property is set to that value.

ID	productReviews.flaggedStatus
UI Location	Shopping > Reviews
JSON file	ProductReviews.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Approved Status

This number specifies the number of times the review has been approved by a user.

ID	productReviews.approvedStatus
UI Location	Shopping > Reviews
JSON file	ProductReviews.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Pending Approval Status

This number specifies the number of pending approvals the review has.

ID	productReviews.pendingApprovalStatus
UI Location	Shopping > Reviews
JSON file	ProductReviews.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Results per Page

This number specifies the number of product reviews displayed per page.

ID	productReviews.resultsPerPage
UI Location	Shopping > Reviews
JSON file	ProductReviews.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Max Rate

This number specifies the maximum rating an item can receive.

ID	productReviews.maxRate
UI Location	Shopping > Reviews
JSON file	ProductReviews.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Compute Overall

This boolean, if checked, causes the application to compute and display the overall rating for the product.

ID	productReviews.computeOverall
UI Location	Shopping > Reviews
JSON file	ProductReviews.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Filter Options

This array specifies how product reviews are filtered and displayed. Each filter option defines the following properties:

- **ID** (string) – specifies the internal identifier of the filter option.
- **Name** (string) – specifies the label of the filter option as it appears in the user interface.
- **Params** (string) – defines a JSON object that declares the URL parameters of the filter option.
- **Is Default** (boolean) – specifies the default filter option displayed in the drop down list.

IDs	productReviews.filterOptions productReviews.filterOptions.id productReviews.filterOptions.name productReviews.filterOptions.params productReviews.filterOptions.isDefault
UI Location	Shopping > Reviews
JSON file	ProductReviews.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Sort Options

This array specifies how product reviews are sorted. Each sort option defines the following properties:

- **ID** (string) – specifies the internal identifier of the sort option.
- **Name** (string) – specifies the label of the sort option as it appears in the user interface.
- **Params** (string) – defines a JSON object that declares the URL parameters of the sort option.
- **Is Default** (boolean) – specifies the default sort option displayed in the drop down list.

IDs	productReviews.sortOptions productReviews.sortOptions.id productReviews.sortOptions.name productReviews.sortOptions.params productReviews.sortOptions.isDefault
UI Location	Shopping > Reviews
JSON file	ProductReviews.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Wishlist Subtab

The settings on this subtab specify properties related to product lists. Product Lists (Wish Lists) provide the ability to group lists of items to be purchased from your web store. This subtab is part of the Shopping tab and contains the following properties:

More Information: [Product Lists](#)

Enable Modifications by Customers

This boolean specifies if user is able to modify (add, edit, or delete) a private product list.

ID	productList.additionEnabled
UI Location	Shopping > Wishlist
JSON file	ProductList.json
Configuration file (pre-Vinson)	SC.Configuration.MyAccount.js

Login Required

This boolean specifies if users must be logged in to modify a product list.

ID	productList.loginRequired
UI Location	Shopping > Wishlist
JSON file	ProductList.json
Configuration file (pre-Vinson)	SC.Configuration.MyAccount.js

List Templates

This array specifies the pre-defined product lists (templates) that are automatically available to your web store users. Product lists defined here are **predefined** by default, meaning the user cannot modify or delete them. By default, a single predefined list (My List) is available. New customers will have these template lists by default. Associated records are created when a user adds a product to the list.

Each product list contains the following properties:

- **Template ID** (string) – specifies the internal identifier for this template. You must ensure that this value is unique.
- **Name** (string) – specifies the name of the product list that appears in the user interface.
- **Description** (string) – specifies a description of the product list that appears in the user interface.
- **Scope ID** (number) – specifies the internal scope ID.
- **Scope Name** (string) – specifies whether the product list is public or private. When scope is not explicitly declared, the default applied is **private**.
- **Type ID** – (string) – specifies the internal type ID.
- **Type Name** – (string) – defines the list as quote or later. Quote specifies the list to be added to a request for quote. Later specifies the list to be saved in the cart for later addition.

More Information: [Product ListsSave For Later](#)

IDs	productList.listTemplates productList.listTemplates.templateId productList.listTemplates.name productList.listTemplates.description productList.listTemplates.scopeId productList.listTemplates.scopeName productList.listTemplates.typeId productList.listTemplates.typeName
UI Location	Shopping > Wishlist
ID (pre-Vinson)	product_lists_templates
JSON file	ProductList.json
Configuration file (pre-Vinson)	SC.Configuration.MyAccount.js

Display Modalities for Product List Items

This array specifies display options for product list items in a user's My Account page. You can display items in various formats in a similar fashion to viewing items in the product display pages. For example, users can view items in a condensed list without images or in a list layout with images. By default, the condensed and list views are included. Each display modality contains the following properties:

- **ID** (string) – specifies the display modality ID.
- **Name** (string) – specifies the description of the modality.
- **Columns** (number) – specifies the number of columns in the Product List.
- **Icon** (string) – Specifies the item icon.
- **Is Default** (boolean) – specifies the default modality.

IDs	productList.templates
------------	-----------------------

	productList.templates.id productList.templates.name productList.templates.columns productList.templates.isDefault
UI Location	Shopping > Wishlist
ID (pre-Vinson)	product_lists_templates
JSON file	ProductList.json
Configuration file (pre-Vinson)	SC.Configuration.MyAccount.js

Shopping Catalog Tab

The settings on this tab let you configure properties related to the shopping, including shopping cart, facets, and product details. This tab includes overall catalog properties plus the following subtabs:

- [Facets Subtab](#)
- [Facets Delimiters Subtab](#)
- [Facets SEO Subtab](#)
- [Item Options Subtab](#)
- [Multi-Image Option Subtab](#)
- [Product Details Information Subtab](#)
- [Recently Viewed Items Subtab](#)

Add to Cart Behavior

This string specifies the action that occurs when the user adds an item to the cart. Possible values are:

- showCartConfirmationModal (default)
- goToCart
- showMiniCart
- showCartConfirmationModal

ID	addToCartBehavior
UI Location	Shopping Catalog
JSON file	Cart.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Add to Cart from Facets View

This boolean specifies if a facet view displays the **Add to Cart** button on each search result.

ID	addToCartFromFacetsView
UI Location	Shopping Catalog

JSON file	Cart.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Allow Adding More Than One PromoCode

This property applies to:

- SuiteCommerce Advanced - Elbrus and later
- SuiteCommerce

This boolean enables or disables the SuitePromotions feature.

More Information: [Promotions](#)

ID	promocodes.allowMultiples
UI Location	Shopping Catalog
JSON file	Cart.json

Allow custom matrix child search in the item list

This property applies to:

- SuiteCommerce Advanced - Kilimanjaro and later
- SuiteCommerce

This boolean enables or disables custom matrix child item search within the Item list. If this property is enabled, the item search uses the field set defined in the **Matrix child items fieldset for search** property for the **Matrix Child Items (Detail)** field.

More Information: See the help topic [Item Search API Input Parameters \(matrixchilditems_fieldset parameter information\)](#).

ID	matrixchilditems.enabled
UI Location	Shopping Catalog
JSON file	Cart.json

Matrix child items fieldset for search

This property applies to:

- SuiteCommerce Advanced - Kilimanjaro and later
- SuiteCommerce

This string defines the field set used with the custom matrix child item search feature. The default value of this field uses the **matrixchilditems_search** field set. This is part of the default field set setup script. However, you can customize this to match any custom matrix child item field set.



Note: If this property is left blank, but the **Allow custom matrix child search in the item list** is enabled, the custom matrix child search feature will not process the custom field set. Instead, the application uses the default **matrixchilditems** field set. You can also set up this feature manually by customizing the Item Search API.

More Information: See the help topic [Item Search API Input Parameters \(matrixchilditems_fieldset parameter information\)](#).

ID	matrixchilditems.fieldset
UI Location	Shopping Catalog
JSON file	Cart.json

If the Information is Available to your Country, shows detailed taxes in the Cart

This property applies to:

- SuiteCommerce Advanced - Kilimanjaro and later
- SuiteCommerce

This boolean enables or disables the per line tax details in the Cart and purchase history. This applies to Australian and Canadian accounts using PST, GST, and VAT tax codes.

More Information: [Web Store Taxes](#)

ID	showTaxDetailsPerLine
UI Location	Shopping Catalog
JSON file	Cart.json

Default Search URL

This string specifies the default search URL.

ID	defaultSearchUrl
UI Location	Shopping Catalog
JSON file	DefaultSearchURL.json
Configuration file (pre-Vinson)	Configuration.js

Maximum Option Values Quantity Without Pusher

This property applies to:

- SuiteCommerce Advanced - Elbrus and later
- SuiteCommerce

This number property indicates the maximum number of option values to appear in line on any mobile device's product details pages before displaying a pusher. For example, if an item lists two dimensions with four values each, the item has eight total values to render in the PDP. Setting this property to 6 results in listing the first dimension and its four values plus the second dimension and its first two values on a mobile device's PDP before displaying a pusher. When a user touches the pusher, the next two item options appear in a new screen.

 **Note:** This property applies to the PDP on mobile devices only.

ID	ItemOptions.maximumOptionValuesQuantityWithoutPusher
UI Location	Shopping Catalog
JSON file	ProductDetails.ItemOptions.json

Facets Subtab

Facets as URL Parameters

This property applies to:

- SuiteCommerce Advanced - Elbrus and later
- SuiteCommerce

This boolean specifies if facets are treated as URL query string parameters or as part of the URL path.

If checked, all facets are treated as parameters. If unchecked, all facets are treated as part of the URL path. This property applies behavior to all facets unless an individual facet's **isParameter** property is specified. See [Facets](#) for details. This property is unchecked by default.

 **Note:** This can affect SEO. See the help topic [Facets as Parameters](#) for details.

ID	facetsAsUrlParameters
UI Location	Shopping Catalog > Facets
JSON file	Facets.json

Facets

More Information: [Faceted Navigation](#)

These settings configure the faceted navigation feature. SuiteCommerce uses modern faceted navigation for displaying product results. With faceted navigation, users can incrementally refine search queries based on specific item attributes defined within the facet configuration.

Each facet contains the following properties:

- **Item Field ID** (string) – specifies the internal identifier of the facet being customized. The value must match the URL Component of the associated facet field as set up in NetSuite. If not specified in the object, the default is the facet field's URL Component. If the URL Component is not set up in NetSuite, the default is the facet field's Field ID.
- **Name** (string) – specifies the display name for the facet as it appears in the browser. If not specified, the default equals the value of the id property.

- **URL** (string) – specifies the URL fragment that identifies the facet in the URL. If no value is defined, SuiteCommerce Advanced uses the NetSuite list record ID.
- This property only applies to pre-Vinson release of SCA.
- **Priority** (string) – sets the display order for the list of facet choices. Facets display in descending order of the priority value (largest priority value displays on top followed by smaller values). The priority value must be between 1 and 10. Default value is 5.
- **Behavior** (string) – sets type of facet editor as it appears in the browser. If not specified, the default is single. Possible values are:
 - **Single** (string) – displays a list from which users select a single choice.
 - **Multi** (string) – displays a list from which users select multiple choices.
 - **Range** (string) – displays a double slider from which users select a start and end value.
- **Template** (string) – specifies the template that defines the HTML source code for the facet. If not specified, the default is `facets_faceted_navigation_item_tpl`.
 - `facets_faceted_navigation_item_color_tpl` (string) – defines the template for a color facet.
 - `facets_faceted_navigation_item_range_tpl` (string) – defines the template for a ranged facet.
 - `facets_faceted_navigation_item_tpl` (string) – defines the template for any other facet.
- **Color Palette** (string) – sets the HTML color values displayed for the facet.
- **Collapsed** (boolean) – sets the default state of the facet. If selected, the facet renders in a collapsed state.
- **Non Collapsible** (boolean) – specifies if the facet is collapsible and expandable. If set to **Yes**, the user can collapse or expand the facet by clicking an up/down arrow icon. If set to **No**, the facet cannot be collapsed or expanded.
- **Show Heading** (boolean) – specifies if the facet heading displays in the browser. If set to **Yes** (checked/enabled), a heading matching the value set in the name property displays. If set to **No** (unchecked/disabled) or if left blank, the facet values display without a heading.
- **Title Format** (string) – specifies the format for the facet title displayed when the facet is selected. This can be a string like **from \$0 to \$1** for a range behavior or **foo \$(0) bar** for others. Also it can be a function that accepts the facet object as a parameter.
- **Title Separator** (string) – specifies a string used to separate facets in the page title. If not specified, the default is **,** (comma space).
- **Parser** (string) – includes the user's currency symbol (\$, £, etc.) to the price range. If this value is not set, SuiteCommerce Advanced does not display a currency symbol.
- **Is URL Parameter?** (boolean) – specifies if the facet is treated as a parameter or as part of the URL path. If **Facets as URL Parameters** is checked for all facets, any individual facet with **Is URL Parameter?** set to false (unchecked) acts as part of the URL path. Likewise, if **Facets as URL Parameters** is unchecked, any individual facet with **Is URL Parameter?** set to true (checked) acts as a parameter.

This property is available in SuiteCommerce and on the Elbrus release of SCA and later.

- **Max** (number) – Specifies the limit of options available for a facet. After this options limit is reached, a see more link appears. This applies to mulit behavior. This is particularly useful for facets with a large amount of options to render.

This property is available in SuiteCommerce and on the Elbrus release of SCA and later.

More Information: [Faceted Navigation](#)

IDs	facets facets.id
------------	---------------------

	facets.name facets.url (pre-Vinson) facets.priority facets.behavior facets.template facets.colors facets.collapsed facets.uncollapsible facets.showHeading facets.titleToken facets.titleSeparator facets.parser facets.isParameter (Elbrus and later) facets.max (Elbrus and later)
UI Location	Shopping Catalog > Facets
JSON file	Facets.json
Configuration file (pre-Vinson)	SC.Checkout.Configuration.js

Facets Delimiters Subtab

These settings specify the characters within the URL that appear between facets, facet names, and their values and options. Ensure that each facet delimiter is unique.

For example, if a user made the following facet selections:

- **Category** – Shoes (a facet with a single selection behavior).
- **Color** – black and brown (a facet with multiplet selection behavior).
- **Display** – set to list results as a table.

Based on the **default** delimiters, the URL will appear as:

`http://www.mystore.com/search/categories/shoes/color/black,brown?display=table`

Between Facet Name and Value

This string specifies the character that appears between the facet name and value. The default value is /. NetSuite recommends introducing one character as the value for this property to prevent unexpected results.

ID	facetDelimiters.betweenFacetNameAndValue
UI Location	Shopping Catalog > Facet Delimiters
JSON file	FacetsDelimiters.json
Configuration file (pre-Vinson)	SC.Configuration.js

Between Different Facets

This string specifies the character that appears between different facets. The default value is /. NetSuite recommends introducing one character as the value for this property to prevent unexpected results.

ID	facetDelimiters.betweenDifferentFacets
-----------	--

UI Location	Shopping Catalog > Facet Delimiters
JSON file	FacetsDelimiters.json
Configuration file (pre-Vinson)	SC.Configuration.js

Between Different Facets Values

This string specifies the character that appears between different facet values. The default value is ,.

ID	facetDelimiters.betweenDifferentFacetsValues
UI Location	Shopping Catalog > Facet Delimiters
JSON file	FacetsDelimiters.json
Configuration file (pre-Vinson)	SC.Configuration.js

Between Range Facets Values

This string specifies the characters that delimit facet ranges. The default value is to.

ID	facetDelimiters.betweenRangeFacetsValues
UI Location	Shopping Catalog > Facet Delimiters
JSON file	FacetsDelimiters.json
Configuration file (pre-Vinson)	SC.Configuration.js

Between Facets and Options

This string specifies the character that appears between a facet and its options. The default value is ?.

ID	facetDelimiters.betweenFacetsAndOptions
UI Location	Shopping Catalog > Facet Delimiters
JSON file	FacetsDelimiters.json
Configuration file (pre-Vinson)	SC.Configuration.js

Between Option Name and Value

This string specifies the character that appears between the option name and its value. The default value is =. NetSuite recommends introducing one character as the value for this property to prevent unexpected results.

ID	facetDelimiters.betweenOptionNameAndValue
UI Location	Shopping Catalog > Facet Delimiters
JSON file	FacetsDelimiters.json

Configuration file (pre-Vinson)	SC.Configuration.js
--	---------------------

Between Different Options

This string specifies the character that appears between two options. The default value is &. NetSuite recommends introducing one character as the value for this property to prevent unexpected results.

ID	facetDelimiters.betweenDifferentOptions
UI Location	Shopping Catalog > Facet Delimiters
JSON file	FacetsDelimiters.json
Configuration file (pre-Vinson)	SC.Configuration.js

Facets SEO Subtab

Use to define limits on the SEO-generated links in the facets browser. When these limits are reached, the URL is replaced with # in the generated links.

More Information: [SEO Page Generator](#)

Number of Facet Groups

This string specifies how many facet groups are indexed.

ID	facetsSeoLimits.numberOfFacetsGroups
UI Location	Shopping Catalog > Facets SEO
JSON file	FacetSeoLimits.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Number of Facet Values

This string specifies how many multi-facet groups are grouped together.

ID	facetsSeoLimits.numberOfFacetsValues
UI Location	Shopping Catalog > Facets SEO
JSON file	FacetSeoLimits.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Options

This string specifies which URL options are indexed. If an option is not included here, it is not indexed when appearing in a URL. Valid values are:

- order
- page
- show
- display
- keywords

ID	facetsSeoLimits.options
UI Location	Shopping Catalog > Facets SEO
JSON file	FacetSeoLimits.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Item Options Subtab

The Item Options subtab displays properties associated with how item options and custom transaction line fields display in your web store across the application. NetSuite provides some templates, as displayed in this tab.

Show Only the Fields Listed In: Item Options and Custom Transaction Line Fields

This property applies to:

- SuiteCommerce Advanced - Elbrus and later
- SuiteCommerce

This boolean specifies what item options and transaction line fields render in your web store. If unchecked (default), all custom transaction line fields and custom item options appear on your site. If checked, only those fields listed in the **Item Options and Custom Transaction Line Fields** table (on this subtab) appear.

More Information: [Commerce Custom Fields](#)

ID	ItemOptions.showOnlyTheListedOptions
UI Location	Shopping Catalog > Item Options
JSON file	Item.Options.json

Item Options and Custom Transaction Line Fields

 **Note:** This set of properties is titled **Item Options** in the Vinson release of SuiteCommerce Advanced.

This section specifies how some custom fields display in your web store. For SuiteCommerce and the Elbrus release of SuiteCommerce Advanced and later, these properties include Item Options and Custom Transaction Line fields. For Vinson release and earlier, these properties only apply to Item Options. Each field listed in this table can contain the following properties:

- **Item Options ID** (string) – specifies the internal identifier of the Custom Item Field being displayed. If declared, this matches the **ID** field of the corresponding Custom Item Field record in NetSuite. This property is only valid for the Vinson release of SuiteCommerce Advanced and earlier and is not required.
 - **Cart Option ID** (string) – specifies the internal identifier of the Item Option being displayed. This matches the **ID** field of the Item Option record you want to display on your site. This property is required and used as the primary key.
 - **Color Palette** (string) – maps a color label to its hexadecimal value. This object is used by the item options and faceted navigation.
 - **Label** (string) – specifies the text of the item option displayed in the user interface.
 - **URL Parameter Name** (string) – specifies the key of the option that appears in the URL. This property applies to Elbrus release and later.
 - **Use Labels on URL** (boolean) – specifies how the item option appears in the URL. If set to false (default), the URL displays the option's Internal ID (for example, `http://mysite.com/product/485?&color=1`). If set to true, the URL contains the option's internal ID label. For example, `http://mysite.com/product/485?&color=blue`. This property applies to Elbrus release and later.
 - **Sort Index** (integer) – specifies the position in which the current option displays in the PDP. The default value is 10. The lower the number, the sooner in the custom field renders. This property applies to Elbrus release and later.
 - **Selector Template** (string) – specifies the template that renders the item option or transaction column field in the PDP and the Cart. This template provides the user with a means to select an option (for example, Item Color = Blue).
- The associated field uses this template instead of the default selector template (as defined in the **Default Selector Templates by Item Option Type** property defined elsewhere on this subtab).
- **Show Option in Item Lists** (boolean) – specifies if the item option appears in the faceted search results page. This property applies to Elbrus release and later.
 - **Facet Cell Template** (string) – specifies the template used to render the item on the faceted search results page.
- The associated field uses this template instead of the default facet cell template (as defined in the **Default Facet Cell Templates by Item Option Type** property defined elsewhere on this subtab). This property applies to Elbrus release and later.

- **Selected Template** (string) – specifies the template used to render the item option or transaction column field in Checkout and My Account.
- The associated field uses this template instead of the default selected template (as defined in the **Default Selected Templates by Item Option Type** property defined elsewhere on this subtab).

More Information: [Commerce Custom Fields](#)

IDs	ItemOptions.optionsConfiguration ItemOptions.optionsConfiguration.cartOptionId ItemOptions.optionsConfiguration.colors ItemOptions.optionsConfiguration.label ItemOptions.optionsConfiguration.urlParameterName ItemOptions.optionsConfiguration.useLabelsOnUrl ItemOptions.optionsConfiguration.index ItemOptions.optionsConfiguration.templateSelector ItemOptions.optionsConfiguration.showSelectorInList ItemOptions.optionsConfiguration.templateFacetCell ItemOptions.optionsConfiguration.templateSelected
IDs (Pre-Elbrus)	itemOptions

Note: In Vinson release of SCA, these properties are located on the Shopping tab and Item Options subtab.	itemOptions.itemOptionId itemOptions.cartOptionId itemOptions.colors itemOptions.label itemOptions.url itemOptions.templateSelected
UI Location	Shopping Catalog > Item Options
JSON file	Item.Options.json
Configuration file (pre-Vinson)	SC.Configuration.js

Default Selected Templates by Item Option Type

This property applies to:

- SuiteCommerce Advanced - Elbrus and later
- SuiteCommerce

This array specifies default templates used to render a selected item option or transaction column field (by item option type) in Checkout and My Account. Each item option contains the following properties:

- **Option Type** (string) – specifies the option type that uses the listed template.
- **Template Name** (string) – specifies the default selected template used to render the field by item option type.

More Information: [Commerce Custom Fields](#)

IDs	ItemOptions.defaultTemplates.selectedByType ItemOptions.defaultTemplates.selectedByType.type ItemOptions.defaultTemplates.selectedByType.template
JSON file	Transaction.Line.Views.ItemOptions.json
UI Location	Shopping Catalog > Item Options

Default Selector Templates by Item Option Type

This property applies to:

- SuiteCommerce Advanced - Elbrus and later
- SuiteCommerce

This array specifies default templates used to select an item option (by type) in the Cart or Order Confirmation page. Each item option contains the following properties:

- **Option Type** (string) – specifies the option types that uses the listed template.
- **Template Name** (string) – specifies the default selector template used to render the item option type.

More Information: [Commerce Custom Fields](#)

IDs	ItemOptions.defaultTemplates.selectorByType ItemOptions.defaultTemplates.selectorByType.type ItemOptions.defaultTemplates.selectorByType.template
UI Location	Shopping Catalog > Item Options

JSON file	ProductViews.ItemOptions.json
------------------	-------------------------------

Default Facet Cell Templates by Item Option Type

This property applies to:

- SuiteCommerce Advanced - Elbrus and later
- SuiteCommerce

This array specifies default templates used to render an item option (by type) the facets search results page. Each item option contains the following properties:

- **Option Type** (string) – specifies the option types that use the listed template.
- **Template Name** (string) – specifies the default facets template used to render the item option type.

IDs	ItemOptions.defaultTemplates.facetCellByType ItemOptions.defaultTemplates.facetCellByType.type ItemOptions.defaultTemplates.facetCellByType.template
UI Location	Shopping Catalog > Item Options
JSON file	ProductViews.ItemOptions.json

Multi-Image Option Subtab

This subtab displays properties that configure what item option fields are used to display the final image for the selected product in the PDP. In SuiteCommerce and the Elbrus release of SCA and later, the image change capability extends to configuring two or more item option IDs.

More Information: [Setting Up Multiple Images for an Item](#)

ID	productline.multiImageOption
UI Location	Shopping Catalog > Multi-Image Option
UI Location (pre-Elbrus)	Layout > Images
JSON file	ProductLine.MultiImages.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Product Details Information Subtab

ⓘ Applies to: SuiteCommerce Web Stores | Aconcagua | Kilimanjaro | Elbrus

The Product Details Information subtab lets you add detailed information about an item in a stacked list on the Product Details Page. Each entry creates a tab in the PDP. This tab provides information stored in a Item record field and displays it in your web store. Each entry contains the following properties:

- **Name** (string) – specifies the label displayed (as a tab) on the PDP.
- **ID** (string) – specifies the Item record field value to display under the associated tab in the PDP.
- **Opened** (boolean) – specifies if the section is open when the page loads. This property applies to pre-Elbrus implementations of SCA.

- **Itemprop** (string) – specifies the `itemprop` HTML attribute (as defined by schema.org). This property is optional.

IDs	productDetailsInformation productDetailsInformation.name productDetailsInformation.contentFromKey productDetailsInformation.itemprop
IDs (pre-Elbrus)	itemDetails itemDetails.name itemDetails.contentFormatKey itemDetails.opened itemDetails.itemprop
UI Location	Shopping Catalog > Product Details Information
UI Location (pre-Elbrus)	Shopping Catalog > Item Details
JSON file	ProductDetails.Information.json
Configuration file (pre-Vinson)	SC.Configuration.js

For example:

The following configuration results in a tab on your PDP titled **Details** that displays information stored in the `storedetaildescription` field in the Item record.

The screenshot shows the 'Product Details Information' configuration screen. A new entry has been added to the list:

NAME *	ID *
Details	storedetaildescription

At the bottom, there are buttons for OK, Cancel, Insert, and Remove.

The screenshot shows the 'Details' tab on a Product Detail Information page. The content is as follows:

Details

Feels fresh and dry whether you're stuck in the office till dark or sneaking out for a of natural and synthetic fibers that push moisture to the outside of the garment for f

Features

Dri-release® blended yarns wick moisture and minimize odor
Chest pocket for small items

The Basics

Dri-release® fabric makes this an amazing shirt for travel.

Weight
8 oz. / 226 g.

Center Back Length
29" / 74 cm

Materials

Recently Viewed Items Subtab

These settings specify how recently viewed items are displayed.

Use Cookies for Recently Viewed Items

This boolean specifies if recently viewed items are tracked with a browser cookie.

ID	recentlyViewedItems.useCookie
UI Location	Shopping Catalog > Recently Viewed Items
JSON file	RecentlyViewedItems.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Number of Displayed Items

This string specifies the number of recently viewed items to display.

ID	recentlyViewedItems.numberofItemsDisplayed
UI Location	Shopping Catalog > Recently Viewed Items
JSON file	RecentlyViewedItems.json
Configuration file (pre-Vinson)	SC.Shopping.Configuration.js

Store Locator Tab

The settings on this tab let you configure properties related to the Store Locator feature. This tab includes the following subtabs:

- [Store Locator Subtab](#)
- [Store Locator Google Maps Subtab](#)

Store Locator Subtab

These settings specify the what information is returned when using the Store Locator feature.

This feature and its settings require:

- SuiteCommerce Advanced - Vinson and later
- SuiteCommerce

More Information: [Store Locator](#)

Icons > Stores

This string specifies the store marker icon that appears in the store list, next to the name of the store and in the Store Information popup (if enabled).

ID	storeLocator.icons.stores
UI Location	Store Locator > Store Locator
JSON file	StoreLocator.json

Icons > Position

This string specifies the store marker icon that appears on the map at the store's relative location.

ID	storeLocator.icons.position
UI Location	Store Locator > Store Locator
JSON file	StoreLocator.json

Icons > Autocomplete

This string specifies the store marker icon that appears to the left of the store name within the predicted results of the auto-complete list.

ID	storeLocator.icons.autocomplete
UI Location	Store Locator > Store Locator
JSON file	StoreLocator.json

Zoom Level in PDP

This number specifies the zoom level to be applied in the map when viewing specific store details (applied in StoreLocator.Details.View.js). Default value is 17.

ID	storeLocator.zoomInDetails
UI Location	Store Locator > Store Locator
JSON file	StoreLocator.json

Show Store Popup

This boolean enables or disables the Store Information popup on the map when a user hovers the pointer over the associated store in the Store Locator list. This is enabled by default.

ID	storeLocator.openPopupOnMouseOver
UI Location	Store Locator > Store Locator
JSON file	StoreLocator.json

Title

This string specifies the title of the Store Locator feature as rendered on the main view and in the header. The default is Store Locator.

ID	storeLocator.title
UI Location	Store Locator > Store Locator

JSON file	StoreLocator.json
------------------	-------------------

Is Enabled

This boolean enables or disables the Store Locator feature. This is enabled by default.

ID	storeLocator.isEnabled
UI Location	Store Locator > Store Locator
JSON file	StoreLocator.json

Radius

This number specifies the search radius from the current location or entered address in which to display stores on the map. The radius units can be configured for kilometers or miles using a different configuration property. The default radius is 50.

ID	storeLocator.radius
UI Location	Store Locator > Store Locator
JSON file	StoreLocator.json

Show Localization Map

This boolean enables or disables a search map on mobile devices. If this property is set to true, a small map appears at the top of the screen on mobile devices while searching for an address. This property applies to mobile devices only.

ID	storeLocator.showLocalizationMap
UI Location	Store Locator > Store Locator
JSON file	StoreLocator.json

Number of Stores per Page

This number specifies the number of stores listed per page if a shopper selects the [See complete list of stores](#) link. The default is 28.

ID	storeLocator.showAllStoresRecordsPerPage
UI Location	Store Locator > Store Locator
JSON file	StoreLocator.json

Default Type of Locations

This string specifies the type of stores to return (stores, warehouses, or both). If no property is specified, the map and list display both location types. Possible values are:

- 1 – returns only locations listed as **Stores** in NetSuite.
- 2 – returns only locations listed as **Warehouse**.
- Leave blank for both

ID	storeLocator.defaultTypeLocations
UI Location	Store Locator > Store Locator
JSON file	StoreLocator.json

Default Quantity of Locations

This number specifies the maximum number of locations returned if no results exist within the search radius specified using the **Radius** property.

ID	storeLocator.defaultQuantityLocations
UI Location	Store Locator > Store Locator
JSON file	StoreLocator.json

Distance Unit

This string specifies the measurement units of the configured radius. Possible values are **mi** (miles) or **km** (kilometers). The default is miles.

ID	storeLocator.distanceUnit
UI Location	Store Locator > Store Locator
JSON file	StoreLocator.json

Store Locator Google Maps Subtab

These settings specify the Google Maps properties for use with the Store Locator feature.

This feature and its settings require:

- SuiteCommerce Advanced - Vinson and later
- SuiteCommerce

More Information: [Store Locator](#)

Google > API Key

This string specifies the Google Maps API key for the mapping engine.

Important: The Store Locator feature uses the Google Maps JavaScript API. For this feature to work properly on your site, you must acquire an authentication API key from Google.

ID	storeLocator.apiKey
-----------	---------------------

UI Location	Store Locator > Store Locator Google Maps
JSON file	StoreLocatorGoogle.json

Google > Center Position Latitude

This number specifies the default Google-specific latitude position (as a string) to center the map when geolocation is disabled.

ID	storeLocator.mapOptions.centerPosition.latitude
UI Location	Store Locator > Store Locator Google Maps
JSON file	StoreLocatorGoogle.json

Google > Center Position Longitude

This number specifies the default Google-specific longitude position (as a string) to center the map when geolocation is disabled.

ID	storeLocator.mapOptions.centerPosition.longitude
UI Location	Store Locator > Store Locator Google Maps
JSON file	StoreLocatorGoogle.json

Google > Zoom Level

This number specifies the Google-specific default map zoom level. The default is 11.

ID	storeLocator.mapOptions.zoom
UI Location	Store Locator > Store Locator Google Maps
JSON file	StoreLocatorGoogle.json

Google > MapTypeControl

This boolean enables or disables the Google-specific Map Type Control. The Map Type Control lets the shopper choose the map type (roadmap, satellite, etc.). If this property is set to true, the map displays this control in the upper left corner of the map. Setting this property to false disables this control.

ID	storeLocator.mapOptions.mapTypeControl
UI Location	Store Locator > Store Locator Google Maps
JSON file	StoreLocatorGoogle.json

Google > StreetViewControl

This boolean enables or disables the Google-specific Street View Pegman icon. The Pegman icon is a Google-specific control that can be dragged onto the map to enable Street View. If this property is set

to true, the map displays this control in the lower right corner of the map. Setting this property to false disables the control.

ID	storeLocator.mapOptions.streetViewControl
UI Location	Store Locator > Store Locator Google Maps
JSON file	StoreLocatorGoogle.json

Google > MapTypeID

This string specifies the Google-specific default map type. Options include (as string):

- **roadmap** – displays the default road map view (default)
- **satellite** – displays Google Earth satellite images
- **hybrid** – displays a mixture of normal and satellite views
- **terrain** – displays a physical map based on terrain information

ID	storeLocator.mapOptions.mapTypeId
UI Location	Store Locator > Store Locator Google Maps
JSON file	StoreLocatorGoogle.json

Configuration File Types

ⓘ Applies to: SuiteCommerce Web Stores | Aconcagua | Kilimanjaro | Elbrus | Vinson

This section explains the different configuration files you may encounter when implementing a SuiteCommerce web store. The types of files and their purpose differ depending on your SuiteCommerce implementation.

- For SuiteCommerce and all Vinson implementations of SuiteCommerce Advanced (SCA) and later, see [JSON Configuration Files](#).
- For pre-Vinson implementations of SCA, see [JavaScript Configuration Files](#).

JSON Configuration Files

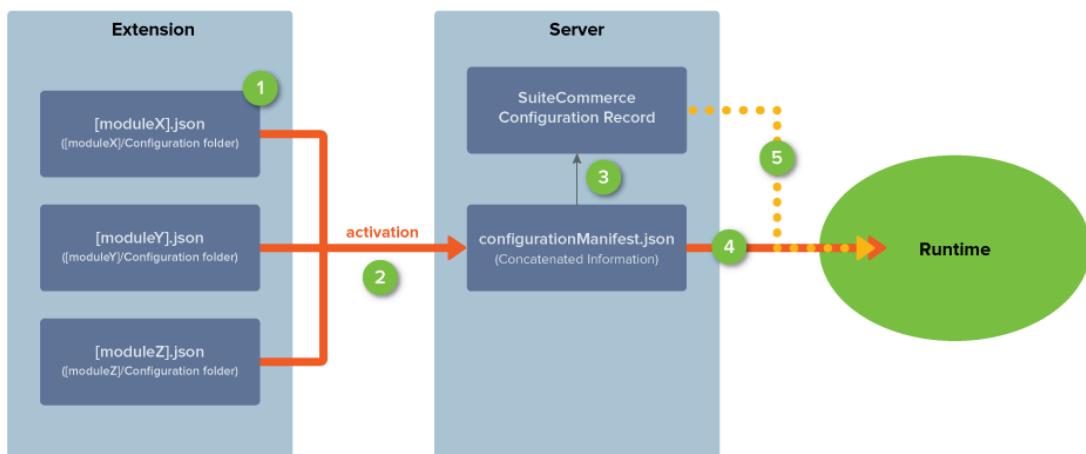
SuiteCommerce configuration requires the SuiteCommerce Configuration record. This presents a user interface to easily change configuration settings for a specified domain. Any changes made in this record save to a custom record and apply at runtime, as described below. The SuiteCommerce Configuration record is all that is required to configure properties for a domain.

The user interface of this record depends on the following files:

- **JSON files** – These files specify configurable properties and metadata related to a module. Modules that include configuration properties contain a subdirectory called **Configuration**, which contains these individual JSON files.
- **configurationManifest.json** – When you activate an extension that includes configuration files, the application concatenates the individual JSON files into the configurationManifest.json file, located in the File Cabinet. If testing locally, you must deploy and activate your extension to see these changes. Metadata stored in this auto-generated file determines the appearance and behavior of the SuiteCommerce Configuration record in NetSuite.

ⓘ Note: JSON configuration files affect the user interface of the SuiteCommerce Configuration record only. They do not configure a domain. To configure properties for a specific domain, see [Configure Properties](#).

The following diagram provides an overview of JSON configuration as it relates to SuiteCommerce web stores:



1. Individual JSON configuration files specify configurable properties and metadata to be used by the SuiteCommerce Configuration record. These files are stored in the modules using the related properties.
2. When you activate an extension for a domain, the application concatenates any JSON configuration files into one configurationManifest.json file.
3. The metadata in the configurationManifest.json file determines what appears in the user interface of the SuiteCommerce Configuration record. The metadata for each property specifies important metadata about each property, including the tab or subtab location, titles, possible values, default values of each property, etc. The configurationManifest.json file sets this information for all domains linked to the associated SSP Application.
4. At runtime, the application uses the default values declared in the configurationManifest.json file to implement configuration properties.
5. If you modify and save the SuiteCommerce Configuration record, these values exist in a custom record and merge with the data from the configurationManifest.json file at runtime. These changes take precedence over any corresponding values stored in the configurationManifest.json file for the associated domain.



Note: The manifest determines the user interface of the SuiteCommerce Configuration record and default property values used by all domains. When you save changes to the SuiteCommerce Configuration record, you create a custom record for the selected domain. To make changes to multiple domains, you must configure each domain individually or use the Copy Configuration functionality. If you do not save any changes to the SuiteCommerce Configuration record, only the values stored in the configurationManifest.json apply at runtime.

For example, the Case.json file determines how properties associated with the Case feature appear in the SuiteCommerce Configuration record user interface. This information can include the tab or subtab where a property appears, the default value, and optional values for the property, among other things. When you deploy and activate your extension, the individual JSON configuration files concatenate into one configurationManifest.json file. When you access the SuiteCommerce Configuration record, this manifest defines what you see in the user interface. You then use the record to configure your site.



Important: If you are implementing a SuiteCommerce Advanced site using a release prior to Aconcagua, this process occurs when you deploy your customizations to NetSuite. See [Create JSON Configuration Files](#) for details on how to add new or modify core JSON configuration files.

Create JSON Configuration Files

This section explains how to create a custom configuration file. This is useful when creating custom configuration properties for your extension. When you deploy your extension to NetSuite and activate it for a domain, the SuiteCommerce Configuration record's user interface reflects any additions specified.



Note: If you are implementing a SuiteCommerce Site using the Kilimanjaro release or earlier, these changes occur when you deploy your custom modules. If you are implementing the Mont Blanc release of SCA or earlier, you must customize configuration JavaScript files to introduce any changes to configuration. See [Customize and Extend Core SuiteCommerce Advanced Modules](#) for details.

JSON Configuration Files Schema

Creating a custom JSON configuration file requires using a custom module and building a new configuration file using JSON Schema V4. Follow this schema when introducing new properties associated with a custom module.



Note: This schema is compliant with Schema V4. JSON schemas, ensuring that all configuration files use a consistent structure and define required objects. For more information, see [json-schema.org](#).

General Structure

Each configuration file contains a root object that defines the configuration information for a module. At the root level, all configuration files must define the **type**, **group**, and **properties** objects. This schema allows for the addition of the **subtab** object as an option.

The following example depicts the general structure and order of a JSON configuration file.

```
//...
{
  "type": "object",
  "group": {
    //...
  },
  "properties": {
    //...
  }
}
//...
```

Type

The **type** object specifies the type of object defined by the configuration file. The value of this object must always be set to **object** and it must be specified in the first line of the configuration file. This is required to be compliant with the JSON schema specification.

```
//...
{
  "type": "object",
  //...
}
```

Group

The SCA Configuration record uses this metadata to determine the tab in which to display configuration properties. The **group** object defines the following:

- **id** – defines an internal identifier for the tab in the SuiteCommerce Configuration record. The id must only contain alphanumeric characters and periods (no other special characters or spaces), and must be unique across all configuration files.
- **title** – defines the label of the tab displayed in the SuiteCommerce Configuration record.
- **description** – provides a description of the **group** object.

The following example, taken from the `checkoutApplications.json` file, initiates the Checkout tab using the group **id** (`checkoutApp`). Any subtabs or properties appearing in this tab of the UI must declare this group id.

```
//...
"group": {
  "id": "checkoutApp",
  "title": "Checkout",
  "description": "Checkout Application configuration"
},
//...
```

Subtab

The SCA Configuration record uses this metadata to determine the subtab in which to display any nested properties. The **subtab** object, if used, defines the following:

- **id** – defines an internal identifier for the subtab in the SuiteCommerce Configuration record. The id must only contain alphanumeric characters and periods (no other special characters or spaces), and must be unique across all configuration files.
- **title** – defines the label of the subtab displayed in the SuiteCommerce Configuration record.
- **description** – provides a description of the **subtab** object.
- **group** – defines the **group** object to which the subtab is assigned. This is the tab where the subtab appears in the UI.

The following example, taken from the checkoutApplications.json file, initiates the Forms subtab using the **subtab** id (checkoutForms) and **group** id (checkoutApp). This locates the **Forms** subtab within the **Checkout** tab, as shown in the following image.

```
//...
{
  "subtab": {
    "id": "checkoutForms",
    "title": "Forms",
    "description": "Checkout configuration related to web forms.",
    "group": "checkoutApp"
  },
//...
```

The screenshot shows two configuration sections:

- Checkout:**
 - See Help
 - SKIP CHECKOUT LOGIN
 - ENABLE MULTIPLE SHIPPING?
 - CHECKOUT STEPS: Standard (dropdown menu)
 - PAYPAL LOGO URL: https://www.paypalobjects.com/webstatic/mktg/logo/pp_cc_mark_11
- Forms:**
 - See Help
 - AUTO POPULATE NAME AND EMAIL
 - LOGIN AS GUEST, SHOW NAME

Resource

The **resource** object specifies data available as a resource within the configurationManifest.json file. You can later refer to this resource using the **source** of a **properties** object to determine the possible choices of a configurable property.

In the following example from categories.json, the resource object defines the commerce category fields to make available as a resource.

```
//...
"resource": {
  "fields": {
    "commercecategory": [
      "internalid", "name", "description", "pagetitle", "pageheading", "pagebannerurl" //...
    ]
  }
}
//...
```

Properties

The **properties** object defines the configuration properties for a module. This object can define one or more properties as object literals. These object literals define the associated property's members/attributes and their values, delimited by commas.

Each **properties** object can define values for the following:

- Id
- Group

- Subtab
- Type
- Title
- Description
- Items
- Enum
- Multiselect
- Default
- Mandatory
- Translate
- Source
- Hidden
- Nstype

Id

The id is the key used in the properties object as the property object declaration. This forms the location of the property in the final configuration object structure. The id is required. The id must only contain alphanumeric characters and periods (no other special characters or spaces), and must be unique across all configuration files. In the following example, the property declaration, `productReviews.maxFlagCount": {type: "string", ...}`, results in the following configuration object:

```
//...
{
  productReviews: {
    maxFlagCount: 1
  }
}
//...
```

Group

The **group** attribute specifies the **id** of the corresponding **group** object (tab) where the property appears in the UI. This attribute is required.

The following example from checkoutApp.json defines the group value of the checkoutApp.skipLogin property as **checkoutApp**. As a result, this property appears on the **Checkout** tab in the UI:

```
//...
,
  "properties": {
    "checkoutApp.skipLogin": {
      "group": "checkoutApp",
      "type": "boolean",
      "title": "Skip Checkout Login",
      "description": "Check this box to enable anonymous users to skip the login/register page...",
      "default": false
    },
}
//...
```

Subtab

The **subtab** attribute specifies the **id** of the corresponding **subtab** object (subtab) where the configurable property appears in the UI.

The following example from checkoutApp.json defines the subtab value of the **autoPopulateNameAndEmail** property as **checkoutForms**, defined earlier. Note that this property also declares the **group** associated with the Checkout tab. As a result, this property appears on the **Checkout** tab and the **Forms** subtab in the UI:

```
//...
  "autoPopulateNameAndEmail": {
    "group": "checkoutApp",
    "subtab": "checkoutForms",
    "type": "boolean",
    "title": "Auto populate name and email",
    "description": "Check this box to enable auto-population of a guest shopper's name...",
    "default": true
  },
//...
```

Type

The **type** attribute specifies the data type of the configurable property. If specified, the user must enter a value of this type in the SuiteCommerce Configuration record. This must be a valid JSON-supported type: **integer**, **string**, **boolean**, **array**, or **object**. This attribute is required.

The following example from checkoutApp.json defines the **checkoutApp.skipLogin** property is defined as a boolean:

```
//...
, "properties": {
  "checkoutApp.skipLogin": {
    "group": "checkoutApp",
    "type": "boolean",
    "title": "Skip Checkout Login",
    "description": "Check this box to enable anonymous users to skip the login/register page...",
    "default": false
  },
//...
```

Title

The **title** attribute defines the label that appears in the SuiteCommerce Configuration record. In the preceding example, the SuiteCommerce Configuration record displays the **checkoutApp.skipLogin** property as **Skip Checkout Login**. This attribute is required.

Description

The **description** attribute specifies a string that displays as help text when a user points to a label in the SuiteCommerce configuration record.

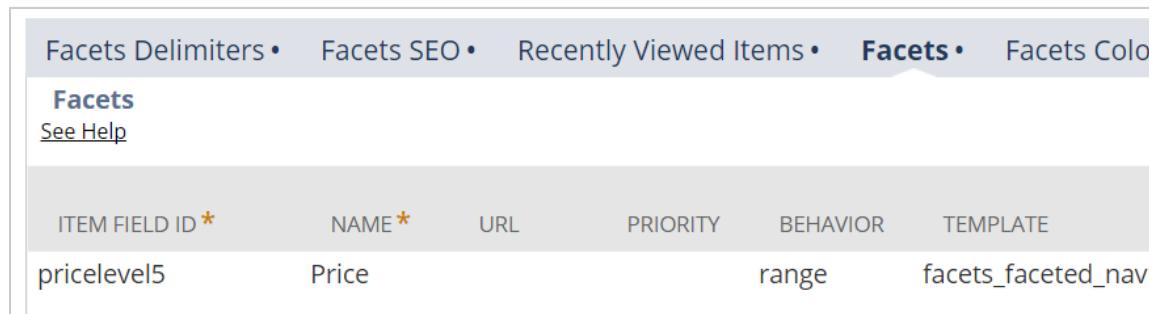
Any **property**, **subtab**, or **group** can declare a description. The SuiteCommerce Configuration record displays the description on simple properties only.

Items

If a property's **type** value is set to an array, **items** is required to define the different properties of the table in the SuiteCommerce Configuration record.

The following example from facets.json initializes Facets as “**type**” : “**array**”. This displays as an array of properties in the SuiteCommerce Configuration record. The **items** attribute defines metadata to further define each configurable property (column), as shown in the image below.

```
//...
"facets": {
    "group": "catalog",
    "type": "array",
    "title": "Facets",
    "docRef": "bridgehead_4393383668",
    "description": "Facets editor declarations",
    "items": [
        {
            "type": "object",
            "properties": {
                "id": {
                    "type": "string",
                    "title": "item field id",
                    "description": "Netsuite item field id, something like 'custitem31'",
                    "mandatory": true
                },
                "name": {
                    "type": "string",
                    "title": "Name",
                    "translate": true,
                    "description": "Label for this facet in the UI",
                    "mandatory": true
                },
                "url": {
                    "type": "string",
                    "title": "Url",
                    "description": "Url fragment for identifying the facet in the url."
                }
            }
        }
    ]
}
```



The screenshot shows the 'Facets' tab selected in the navigation bar. Below it, there is a table with one row. The columns are labeled: ITEM FIELD ID*, NAME*, URL, PRIORITY, BEHAVIOR, and TEMPLATE. The first column contains 'pricelevel5', the second contains 'Price', the fourth contains 'range', and the fifth contains 'facets_faceted_navi'. The table has a light gray header and white body rows.

ITEM FIELD ID*	NAME*	URL	PRIORITY	BEHAVIOR	TEMPLATE
pricelevel5	Price			range	facets_faceted_navi

Enum

The **enum** attribute specifies the set of available values for a configurable property. By default, when **enum** specifies multiple values, the user can only select one of the possible values. This generates a list of possible selections in the UI.



Note: Adding the `enum` attribute to a property renders that property as a **select** or **multi-select** field in the SuiteCommerce Configuration record UI. To render a property as a multi-select field, set the `multiselect` attribute to `true`.

The following example from facets.json specifies five possible values for the `facetsSeoLimits.options` property (order, page, show, display, or keywords):

```
//...
"facetsSeoLimits.options": {
  "type": "string",
  "group": "catalog",
  "subtab": "facetsSeoLimits",
  "title": "Options",
  "description": "Description of this property",
  "enum": ["order", "page", "show", "display", "keywords"],
  "default": ["page", "keywords"],
  "multiselect": true
},
//...
```

Multiselect

The `multiselect` attribute specifies that a configurable property can contain multiple values in the UI. If set to `true`, users can select multiple values as defined in `enum`. This generates a multi-select list in the UI.

In the previous example, the `facetsSeoLimits.options` property is set to accept multiple values with the following code:

```
"multiselect": true
```

As a result, users can choose multiple options from the SuiteCommerce Configuration record UI. The property can also accept multiple default values as defined in the `default` attribute.

Default

The `default` attribute specifies default values based on the property type, defined earlier. These values automatically populate fields in the SuiteCommerce Configuration record UI. Defaults load in the application at runtime. These defaults are only superseded by changes saved in the SuiteCommerce Configuration record. If no user saves a record, the application uses the defaults specified here.

In the previous example, the default values for the `facetsSeoLimits.options` property are `page` and `keywords`.

Mandatory

The `mandatory` attribute specifies that a property is required. In the following example from facets.json, the `name` property is set to `true`. As a result, the attribute is required.

```
//...
"name": {
  "type": "string",
  "title": "Name",
  "translate": true,
  "description": "Label for this facet in the UI",
```

```

    "mandatory": true
},
//...

```

Translate

The **translate** attribute specifies that the **title** and **description** attribute values must be translated. If this is set to true, its default values are translated. In the preceding example, the name property's **title** and **description** fields are set to be translated.

Source

The **source** attribute declares the data source to be used as possible choices when configuring the property in the UI. This can include a variable that refers to the **resource** object declared within the JSON file (or within another JSON file that makes up the configurationManifest.json).

Note: Adding the **source** attribute to a property renders that property as a **select** or **multi-select** field in the SuiteCommerce Configuration record. To render a property as a multi-select field, set the **multiselect** attribute's value to **true**.

The following example from categories.json uses the **source** attribute to refer to a local **resource** object (**fields.commercecategory**). In this example, **\$resource** references the **fields.commercecategory** array:

```

//...
"categories.sideMenu.sortBy": {
  "group": "integrations",
  "subtab": "categories",
  "type": "string",
  "title": "Side menu > sort by",
  "description": "Enter the Category record field to act as the primary sort field in the Categories sidebar.",
  "source": "$resource.fields.commercecategory",
  "default": "sequencenumber"
}
//...

```

The **source** attribute can also refer to data within some custom records in NetSuite via SuiteScript. In this case, **source** references elements of a specific record type. The variable used must match the Internal ID of a applicable custom record types.

The following fictitious example uses the **source** attribute to provide a list of all support issues within NetSuite (record type: Issue):

```

//...
"issue": {
  "type": "string",
  "title": "Support Issues",
  "source": "issue"
}
//...

```

Hidden

The **hidden** attribute specifies that a property is hidden and not shown in the user interface. If set to true, the property's default values are still present in configurationManifest.json, but the user will not be able to see or edit the property using the SuiteCommerce Configuration record.

Nstype

The **nsType** attribute specifies a concrete NetSuite widget for editing a type parameter. If a text field is too long for normal text entry, the user can declare "nsType": "textarea" and the user interface will show a text area instead of a normal text entry.

The following example from checkoutApp.json declares the nstype:

```
//...
"checkoutApp.invoiceTermsAndConditions": {
  "group": "checkoutApp",
  "type": "string",
  "nsType": "textarea",
  "title": "Invoice Terms and Conditions",
  "description": "Invoice payment method terms and conditions text",
  "default": "<h4>Invoice Terms and Conditions</h4.><p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.<p>"
},
//...
```

Configuration Modification Schema

This section describes the configuration modification schema. Follow these steps to create a JSON modification file.

Step 1: Add the Type Object

The type specifies the type of object defined by the configuration file. The value of this object must always be set to **object**. This is required to be compliant with the JSON schema specification.

```
{
  "type": "object",
}
```

Step 2: Add the Modifications Array

When you deploy to NetSuite, the developer tools use the modifications listed in this array to create the configurationManifest.json. Each **modifications** object requires the target and action components. Some actions require an addition value component.

This declares the modifications to include when you deploy to NetSuite.

```
{
  "type": "object",
  "modifications": [
    ]
}
```

Step 3: Add the Target Attribute

The target attribute contains a string representing a JSONPath query. The elements returned are the targets to be affected by the action. In this example, when you deploy your customizations, the JSONPath query searches for the **enum** property of the target **addToCartBehavior** property.

```
{
  "type": "object",
  "modifications": [
    {
      "target": "$.properties.addToCartBehavior.enum",
    }
  ]
}
```



Note: For more information on how to build a target JSONPath query, see <https://github.com/s3u/JSONPath>.

Step 4: Add the Action Attribute

The action attribute indicates the operation to apply over each element of the target returned by the JSONPath query. Possible actions are:

- **add** – adds a new element to the user interface, such as a value, a property, or an option. Only object, array, or string elements can be returned by the target query when using the **add** action, and each type results in a different behavior.
- **replace** – overwrites values in a configuration file. The target query can only return a number, string, boolean, or null.
- **remove** – deletes an element from an array. The target query only can return a number, string, boolean, or null inside an array.

This example includes the **add** action.

```
{
  "type": "object",
  "modifications": [
    {
      "target": "$.properties.addToCartBehavior.enum",
      "action": "add",
    }
  ]
}
```

Step 5: Add the Value Attribute

The value attribute specifies the value to be added or replaced. Only the **add** and **replace** actions require this attribute.

When using the **add** action, the value attribute behaves differently depending on the target type:

- **Target is an Object** – The value attribute merges into the target. If the value has a property with the same name as a value in the target, the target value gets overwritten.
- **Target is an Array** – This value attribute is pushed into the array. This value can be any valid JSON element.
- **Target is a String** – This value attribute gets concatenated to the end of the target string. This value must be a number or a string.

The following example adds the **newOption** configuration option to the **addToCartBehavior** list in the user interface.

```
{
  "type": "object",
  "modifications": [
    {
      "target": "$.properties.addToCartBehavior.enum",
      "action": "add",
      "value": "newOption"
    }
  ]
}
```

Use Case Examples

Add a New Default Property to an Array

This example uses the **add** action to introduce a new property to the object describing a `resultsPerPage` option. When you deploy this customization, the JSONPath query searches for the `resultsPerPage.items` property whose value is 12 per page. This example modification then adds `newProperty` to the array.

```
{
  "type": "object",
  "modifications": [
    {
      "target": "$.properties.resultsPerPage.default[?(@.items == 12)]",
      "action": "add",
      "value": {"newProperty" : "new property value"}
    }
  ]
}
```

After deploying this modification, the code in the `configurationManifest.json` looks like this:

```
...
"default": [
  {
    "items": 12,
    "name": "Show ${0} products per page",
    "newProperty" : "new property value"
  },
  ...
]
```



Note: This example assume you have created a custom `newProperty` using the JSON configuration schema. For details, see [JSON Configuration Files](#).

Add Text to an Existing String

This example uses the **add** action to append text to a string, such as a label of a property within an array. When you deploy this customization, the JSONPath query searches for the `resultsPerPage.default` property whose `item` value is 12. This example modification then appends `!!!` to the end of the string in the default `name` property.

```
{
```

```

    "type": "object",
    "modifications" : [
        {
            "target": "$.properties.resultsPerPage.default[?(@.items == 12)].name",
            "action": "add",
            "value": "!!!"
        }
    ]
}

```

After deploying this modification, the code in the configurationManifest.json looks like this:

```

...
"default": [
    {
        "items": 12,
        "name": "Show ${0} products per page!!!",
    },
...

```

After deploying this modification, the SuiteCommerce Configuration record looks like this:

ITEMS*	NAME*
12	Show \${0} products per page!!!
24	Show \${0} products per page
48	Show \${0} products per page

Add a New Default Option to an Array

This example uses the **add** action to introduce a new configuration option within an array. When you deploy this customization, the JSONPath query searches for the **resultsPerPage.default** property. This modification then adds a new row to the table for a default configuration of 5 items per page.

```

{
    "type": "object",
    "modifications" : [
        {
            "target": "$.properties.resultsPerPage.default",
            "action": "add",
            "value": {
                "items": 5,
                "name": "Show 5 products per page"
            }
        }
    ]
}

```

After deploying this modification, the code in the configurationManifest.json looks like this:

```

...
"default": [
  {
    "items": 12,
    "name": "Show ${0} products per page"
  },
  {
    "items": 24,
    "name": "Show ${0} products per page",
    "isDefault": true
  },
  {
    "items": 48,
    "name": "Show ${0} products per page"
  },
  {
    "items": 5,
    "name": "Show 5 products per page"
  }
],
...

```

After deploying this modification, the SuiteCommerce Configuration record looks like this:

Type Ahead • Result Sorting Search Results per page Search Results • Result Display Options	
Search Results per page	
See Help	
ITEMS*	NAME*
12	Show \${0} products per page
24	Show \${0} products per page
48	Show \${0} products per page
5	Show 5 products per page

Change the Default Value of a Property

This example uses the **replace** action to overwrite the default value of a property. When you deploy this customization, the JSONPath query searches for the `newsletter.genericFirstName` and `newsletter.genericLastName` properties. This modification replaces the current value, `unknown`, with `FirstName` and `LastName`, consecutively.

```

{
  "type": "object",
  "modifications": [
    {
      "target": "$.properties[newsletter.genericFirstName].default",
      "action": "replace",
      "value": "FirstName"
    },
    {
      "target": "$.properties[newsletter.genericLastName].default",
      "action": "replace",
      "value": "LastName"
    }
  ]
}

```

```

        "action": "replace",
        "value": "LastName"
    },
]
}

```

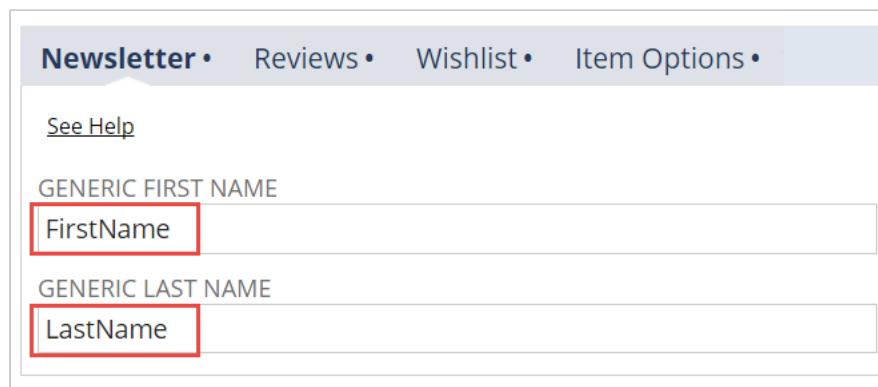
After deploying this modification, the code in the configurationManifest.json looks like this:

```

...
"properties": {
    "newsletter.genericFirstName": {
        "group": "shoppingApplication",
        "subtab": "newsletter",
        "type": "string",
        "title": "Generic first name",
        "description": "Enter the generic first name to populate...",
        "default": "FirstName",
        "id": "newsletter.genericFirstName"
    },
    "newsletter.genericLastName": {
        "group": "shoppingApplication",
        "subtab": "newsletter",
        "type": "string",
        "title": "Generic last name",
        "description": "Enter the generic last name to populate...",
        "default": "LastName",
        "id": "newsletter.genericLastName"
    },
...

```

After deploying this modification, the SuiteCommerce Configuration record looks like this:



Change the Label of a Subtab

Note: This procedure is similar for making changes to a tab. To replace the label of a tab, use the **group** property.

This example uses the **replace** action to overwrite the title of a subtab. When you deploy this customization, the JSONPath query searches for the subtab with the id **newsletter**. This modification then changes the title from **Newsletter** to **Email Newsletter**.

```
{
  "type": "object",
  "modifications": [
    {
      "target": "$[?(@property == 'subtab' && @.id == 'newsletter')].title",
      "action": "replace",
      "value": "Email Newsletter"
    }
  ]
}
```

After deploying this modification, the code in the configurationManifest.json looks like this:

```
...
{
  "type": "object",
  "subtab": {
    "id": "newsletter",
    "group": "shoppingApplication",
    "title": "Email Newsletter",
    "docRef": "bridgehead_4685031554",
    "description": "Configuration of the Newsletter subscription"
  },
  ...
}
```

After deploying this modification, the SuiteCommerce Configuration record looks like this:



Remove a Configuration Option

This example uses the **remove** action to remove an option for a property. When you deploy this customization, the JSONPath query searches for the **addToCartBehavior** property's enum array. This modification removes the **showMiniCart** option in the user interface.

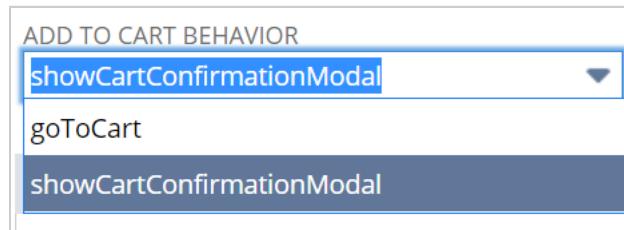
```
{
  "type": "object",
  "modifications": [
    {
      "target": "$.properties.addToCartBehavior.enum[?(@ == 'showMiniCart')]",
      "action": "remove"
    }
  ]
}
```

```
}
```

After deploying this modification, the code in the configurationManifest.json looks like this:

```
...
"properties": {
    "addToCartBehavior": {
        "group": "catalog",
        "subtab": "cart",
        "type": "string",
        "title": "Add to cart behavior",
        "description": "Choose the action that occurs when the user adds an item to the cart.",
        "default": "showCartConfirmationModal",
        "enum": [
            "goToCart",
            "showCartConfirmationModal"
        ],
        "id": "addToCartBehavior"
    },
...
}
```

After deploying this modification, the SuiteCommerce Configuration record looks like this:



Hide a Property from the User Interface

This example uses the **add** action to introduce the **hidden** property and set it to true.



Important: Do not customize the source code to remove any configurable properties included with SCA. Removing configurable properties will result in errors and can break your site. To prevent a property from appearing in the SuiteCommerce Configuration record's user interface, use the add action to introduce the **hidden** element to the desired property. This maintains the required code in the configurationManifest.json, but removes the property from the user interface.

When you deploy this customization, the JSONPath query searches for the **newsletter.companyName**. This modification introduces the **hidden** property and sets it to true. This action prevents the property from appearing in the user interface, but maintains the code in configurationManifest.json.

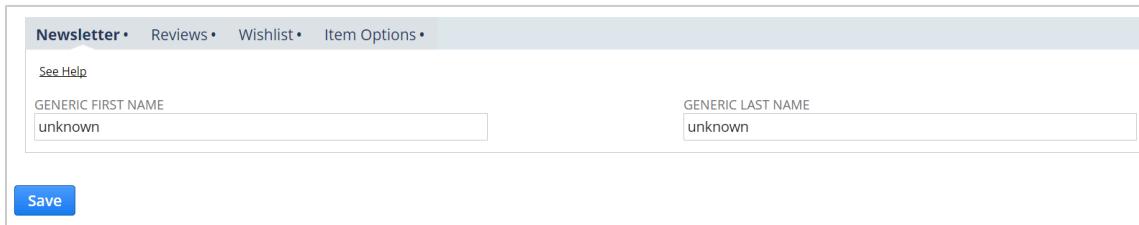
```
{
    "type": "object",
    "modifications": [
        {
            "target": "$.properties[newsletter.companyName]",
            "action": "add",
            "value": {"hidden": "true"}
        }
    ]
}
```

```
        ]
    }
```

After deploying this modification, the code in the configurationManifest.json looks like this:

```
...
"newsletter.companyName": {
    "group": "shoppingApplication",
    "subtab": "newsletter",
    "type": "string",
    "title": "Company Name",
    "description": "Enter the generic Company name to populate...",
    "default": "unknown",
    "hidden": "true",
    "id": "newsletter.companyName"
}
...
```

After deploying this modification, the SuiteCommerce Configuration record looks like this:



JavaScript Configuration Files

ⓘ Applies to: SuiteCommerce Advanced | Mont Blanc | Denali

For implementations prior to the Vinson release of SuiteCommerce Advanced (SCA), configuration files are stored as JavaScript files for each application module. Each of the configuration JavaScript files are contained within an application module. Each file is compiled and deployed as part of the application. To edit a configuration file, you must create a custom version of its application module.

ⓘ Note: For information on configurable properties, see [Configuration Properties Reference](#).

There are several files used to configure frontend behavior and a single file backend configuration file to modify server-side behavior.

Frontend Configuration

The SC.Configuration.js file contains general configuration properties used by SCA. Each of the application-specific configuration files contain this file as a dependency. Also, any JavaScript file within a module that needs to access configuration properties directly includes this file as a dependency.

SCA uses the following configuration files to configure the behavior of the frontend application. These all return an object called Configuration that is accessed by other modules.

- **SC.Configuration** – defines the configuration objects and properties that are used by all SCA applications globally. This is part of the SCA application module.

- **SC.Checkout.Configuration** – defines configuration objects and properties that are used by the Checkout application. This is defined in the CheckoutApplication application module.
- **SC.MyAccount.Configuration** – defines configuration objects and properties that are used by the My Account application. This is defined in the MyAccountApplication.SCA application module.
- **SC.Shopping.Configuration** – defines configuration objects and properties that are used by the Shopping application. This is defined in the ShoppingApplication application module.

For a detailed example on how to customize frontend configuration files in a pre-Vinson implementation, see [Extend Frontend Configuration Files](#).

Backend Configuration

The Configuration.js file within the SspLibraries module defines the backend NetSuite configuration properties for SCA. By modifying this file, you can configure a variety of objects server-side. Amongst other things, modifying these objects can often improve performance of your website by restricting search results. There is a single Configuration.js file used for all applications, although some objects are applicable for only certain applications (Checkout).

For a detailed example on how to customize the backend configuration file in a pre-Vinson implementation, see [Extend the Backend Configuration File](#).

Site Management Tools Configuration

ⓘ Applies to: SuiteCommerce Web Stores | Site Management Tools

Site Management Tools (SMT) let you manage content on your site by dragging and dropping new content, editing or removing existing content, and rearranging content by dragging it from one location to another. This section explains how to configure SMT for your site.

See the following help topics for additional information on SMT related configuration:

- [Upgrade from SMT Version 2 to SMT Version 3](#)
- [SMT Templates and Areas](#)
- [SMT Custom Preview Screen Sizes](#)
- [Working with SMT Landing Pages in a Sandbox Account](#)
- [Changing SMT to Use a Different Hosting Root](#)
- [Configuring Escape to Log In](#)
- [Internationalization of SMT Administration](#)

Upgrade from SMT Version 2 to SMT Version 3

ⓘ Applies to: SuiteCommerce Web Stores | Site Management Tools | Kilimanjaro | Aconcagua

To upgrade from SMT version 2 to SMT version 3, follow these steps:

1. [Install the SMT Core Content Types Bundle](#)
2. [Migrate SMT Content required only when upgrading from version 2 to version 3 of SMT](#)
3. [Deploy the Supported SCA Version to a Site](#)

Some important benefits of upgrading from SMT version 2 to SMT version 3 include:

- Version 3 provides you with some features that are not available in version 2. See the help topic [SMT Versions](#) for a list of features and version availability.
- Content for SMT version 3 is stored as NetSuite Records. See [CMS Records for SMT](#).
- Content records for version 3 content are exposed to SuiteScript. See the help topic [Website](#).



Important: It is best practice to run only one version of Site Management Tools on a single site. For example, if you have a single site record with multiple domains, do not run SMT version 2 on one domain and version 3 of SMT on another domain. Each domain should run the same version of SMT.

Install the SMT Core Content Types Bundle

SMT Version 3 requires installation of the SMT Core Content Types Bundle. This bundle creates the four SMT Core Content Types and their corresponding custom records in your NetSuite account. These content types include:

- Text
- Image
- Merchandising Zone
- HTML

The content types and their associated custom records are locked and cannot be edited or deleted. For information on the custom records and CMS Content Records, see [CMS Records for SMT](#).

Install the SMT Core Content Types bundle as the first step in implementing or migrating to SMT version 3. The bundle ID is 190323.

See the help topic [Installing a Bundle](#).

Migrate SMT Content

Content that you add to a site running SMT version 2 is stored differently from content for version 3 of SMT. When you upgrade your site, such as migrating from a pre-Kilimanjaro release of SCA to Kilimanjaro or greater, the content on the site must be migrated from version 2 to version 3. Without this migration, content created with version 2 of SMT does not display on the site.



Important: This process migrates only the published content on your site. If your SMT version 2 site has content that has not yet been published, you must publish the content before the content migration. If you have content that is not ready to be published, you must manually add it to your site after the migration.

Multiple Migrations

You can run the migration process multiple times. Each time you run the migration, previously migrated content is cleared and re-created from the version 2 content. You receive a warning about this during the migration. An example of when you could encounter this situation is if you run the migration, but then realize you have content that was not yet published. Since unpublished content is not migrated, you must publish the content, and then run the migration again.



Important: If you have upgraded your site to version 3, downgraded to version 2, then upgraded to version 3, it is important to re-run the migration to migrate any new content that was added to your version 2 site.

To migrate content from SMT version 2 to version 3:

1. Go to Lists > Web Site > CMS Contents.
2. Click **CMS Content Migration**. This displays the CMS Content Migration page.
3. Select the site you want to migrate from the **Site** dropdown list.
4. Click **Begin Migration**.



Important: This displays a message that the migration deletes any SMT version 3 content. If you are migrating from SMT version 2 to SMT version 3 then there is no need to be alarmed by the message. You can run the migration multiple times and each time it will clear the previously migrated content and migrate it again from the content on the version 2 site.

5. Click **OK** to begin the migration.

It may take several minutes to complete the migration. You can click **Refresh** to monitor the status of the migration.

Deploy the Supported SCA Version to a Site

Site Management Tools version 3 is supported by SuiteCommerce and the Kilimanjaro release of SuiteCommerce Advanced and greater. If you are setting up a new site with the SuiteCommerce or the Kilimanjaro or greater release of SuiteCommerce Advanced, then follow the normal implementation

procedure. See the help topic [Install Your SuiteCommerce Application](#). If your site currently runs on a pre-Kilimanjaro release of SCA, then you must migrate to SuiteCommerce or a supported release of SuiteCommerce Advanced. See [Update SuiteCommerce Advanced](#).

SMT Templates and Areas

The structure of your website pages is controlled by template files. The template files contain HTML markup and place holders for data and scripts. Site Management Tools uses areas defined within the template files to determine the locations on a page where you can add and manage website content. Each of the following SuiteCommerce Advanced (SCA) template files includes pre-defined areas for use with Site Management Tools:

- Home page template
- Page header template
- Page footer template
- Breadcrumb template
- Item detail template
- Facet search template
- Landing page template

For information on using Site Management Tools, see the help topic [SMT Overview](#).

For more information on Site Management Tools areas and templates, see the following help topics:

- [Site Management Tools Areas](#)
- [Templates and Areas for SCA](#)

Site Management Tools Areas

Areas are defined by adding `div` tags to the template. Site Management Tools uses the area `div` tag to know where the areas are located on each page and also to determine the scope of any content placed within that area. The **scope** of an area determines on which website pages any assigned content displays. When you add content to a page, each area is labeled with its scope so you know how and when the content is displayed. The three scope types are:

- [All Pages Areas](#)
- [Page Type Areas](#)
- [This Page Areas](#)

Each `div` tag has the two following attributes that name the area and set its scope:

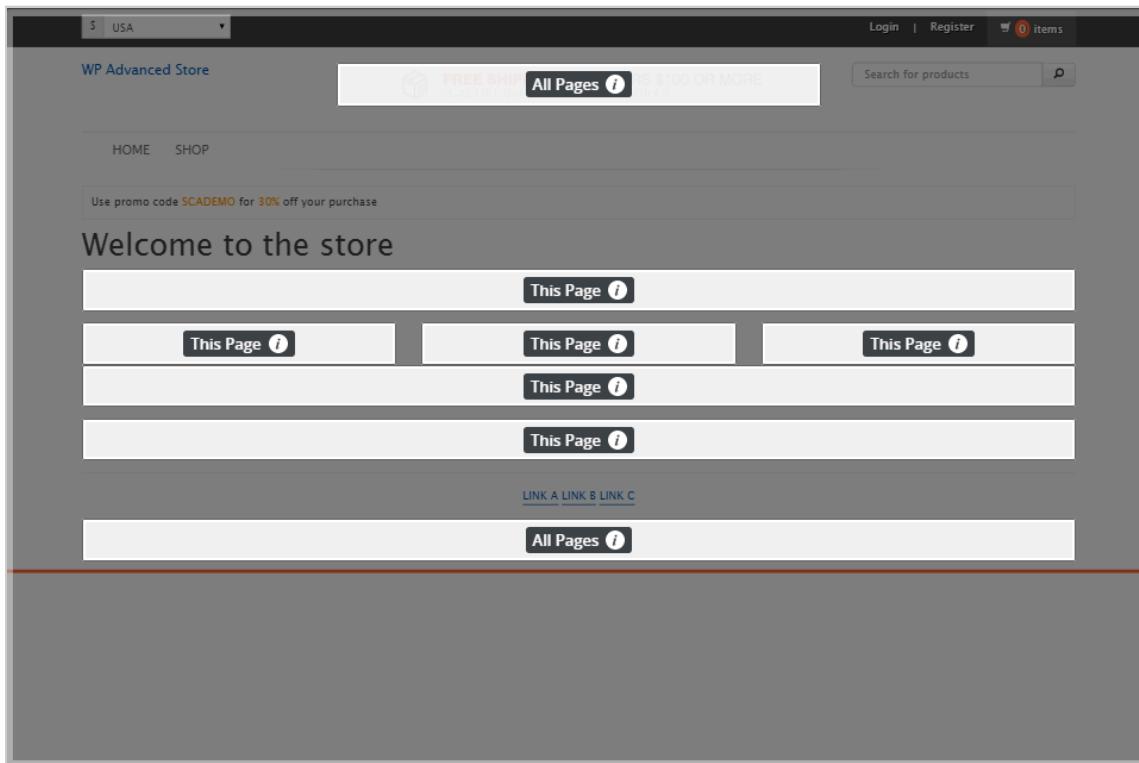
- **data-cms-area** — The value of the `data-cms-area` attribute is a user-defined name given to the area. For an area you want to add to the main part of the page, you might set the value of the `data-cms-area` attribute to `main`, `main-header`, or `main-footer`. For an area that you want to place in a sidebar, you might set the value of the `data-cms-area` to `sidebar`, `sidebar-right`, or `sidebar-left`, and so on. Remember, this is a user-defined value that names the area, so use the naming convention that works best for you. For example, name the area by its intended purpose or page location.

 **Note:** The `data-cms-area` name is used only to define the area in template files. The name is not displayed on the website.

- **data-cms-area-filters** — This attribute determines the scope of an area, meaning on which pages content in that area displays. Scope types are defined as follows:

Area Scope	data-cms-area-filters	Description
All Pages	global	The All Pages area displays content on any page, provided the page has an area with the same data-cms-area .
Page Type	page_type	The Page Type area displays content on any page of the designated type, for example, product list or product detail.
This Page	path	The This Page area is the most specific area with regard to when content displays. Content displays only on the page specified by the path in the page URL. This is the type of area to use if you want to display content for a specific product, facet, or home page.

Each page template can include multiple areas. For example, you may want to put one or more **This Page** areas and a **All Pages** area on your home page. You may want to include the same **All Pages** area on the Item Detail and Facet Browse pages. You may also want to include one or more **This Page** and **Page Type** areas on the **Facet Browse** and **Item Detail** pages. Pages can have multiple areas.



Note: Areas are displayed visually in edit mode when you add content to the page. The label for each area indicates the scope of that area.

All Pages Areas

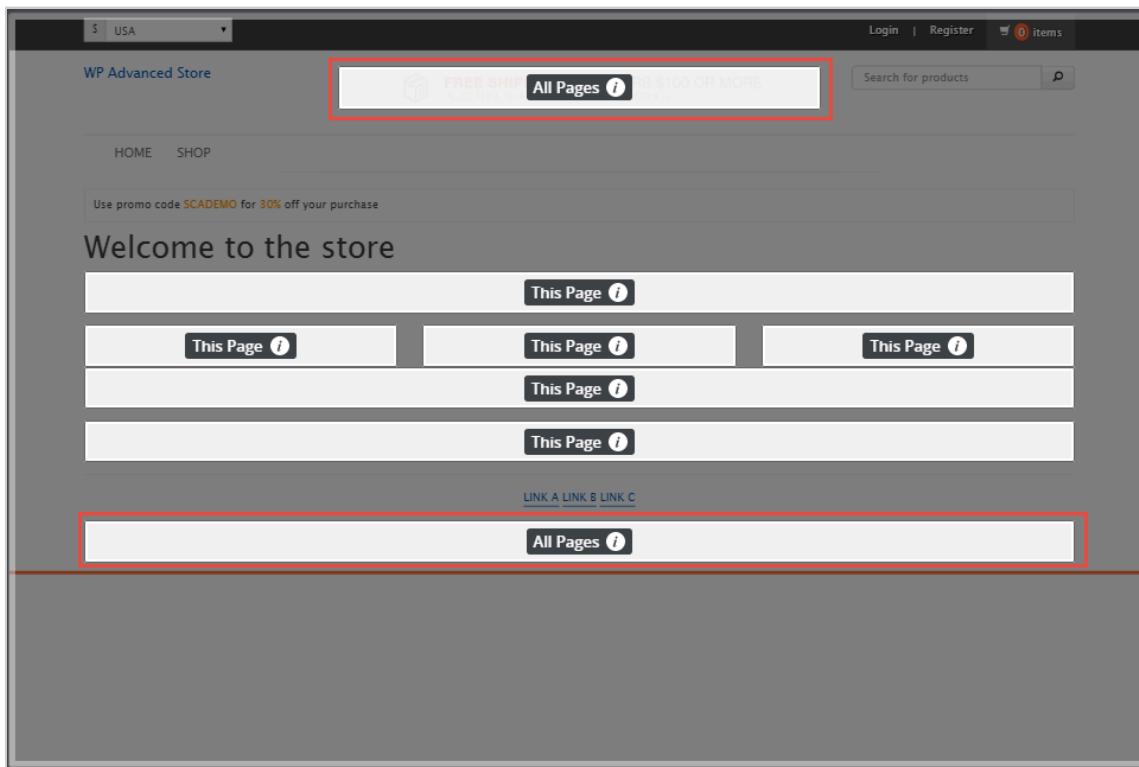
You create an area with **All Pages** scope by setting the **data-cms-area-filters** attribute in the area **div** to **global**. **All Pages** areas display on any page on the site that contains a **global** area with the same **data-cms-area** name. For example, if you define an area with a **data-cms-area** equal to **main** in the, **home.tpl**, **facets_facet_browse.tpl**, and **item_details.tpl** template file, any content you place in that area is displayed on the following pages:

- The Home Page
- All Facet Browse Pages
- All Item Detail Pages

The content is displayed on those pages because each of the pages contains a global area named **main**. The following code sample illustrates this area:

```
<div data-cms-area="main" data-cms-area-filters="global">
</div>
```

By adding this area **div** to one or more templates, you create a new **All Pages** area on the template. In the following screen shot, you see two **All Pages** areas.



These two area **divs** are defined as follows:

```
<div data-cms-area="header_banner" data-cms-area-filters="global"></div>
<div data-cms-area="global_banner_footer" data-cms-area-filters="global"></div>
```

In this example, the **divs** are added to the header and footer template files. By including them in those templates, the areas are available on all shopping pages without having to add them to each template file.

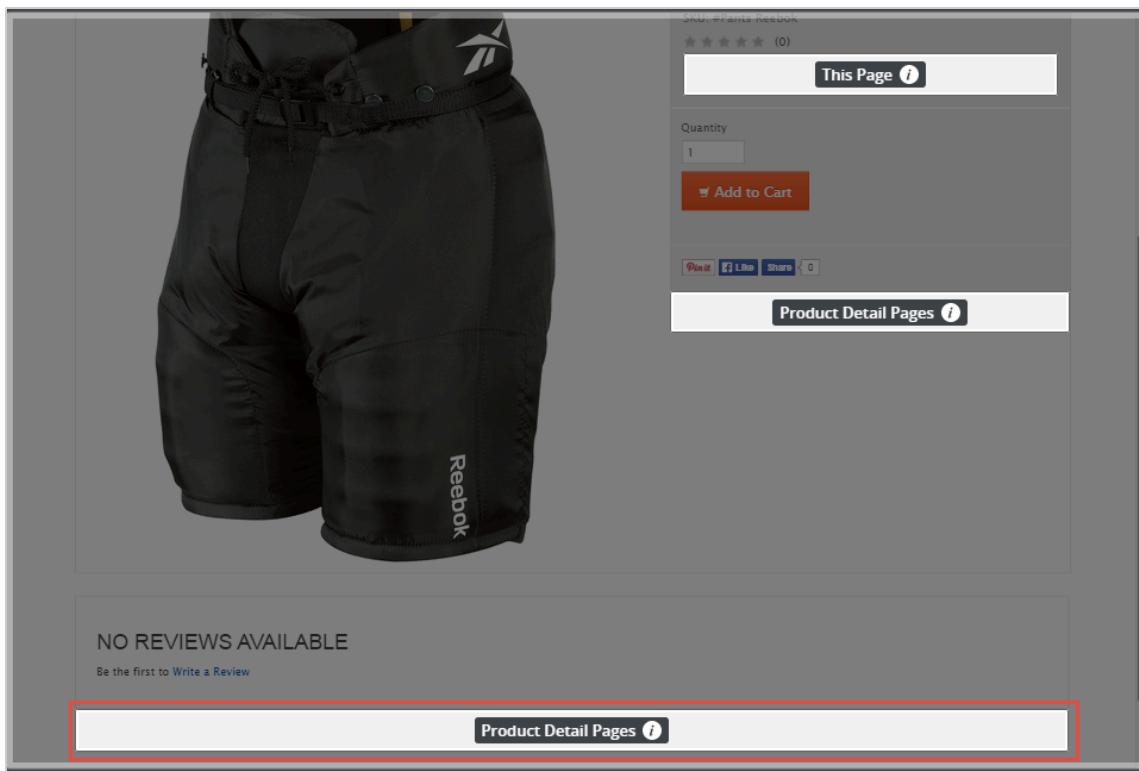
Page Type Areas

Areas with page type scope are identified by the type of page, for example **Product Detail** pages or **Facet Browse** pages . You create an area with page type scope by setting the value of the **data-cms-area-**

filters attribute to `page_type`. The page type can be the home page, **Product Detail** page, or **Facet Browse** page. For example, you may want an area at the bottom of a product detail page that you can use for displaying text, merchandising zones, banners, etc. To do this, you modify the product details template file to add an area. Set the value of `data-cms-area` to `item_info_bottom`, and set the value of `data-cms-area-filters` to `page_type`. The `div` code for this area is as follows:

```
<div data-cms-area="item_info_bottom" data-cms-area-filters="page_type">
</div>
```

By adding this area `div` to the product detail page template, the area will be available on every product detail page. Content placed in the area displays for every product. The area on the page appears similar to this:



Notice that in this example, the area is labeled with the type of page: **Product Detail Pages**. This lets you know that any content you add to that area displays on every product detail page.

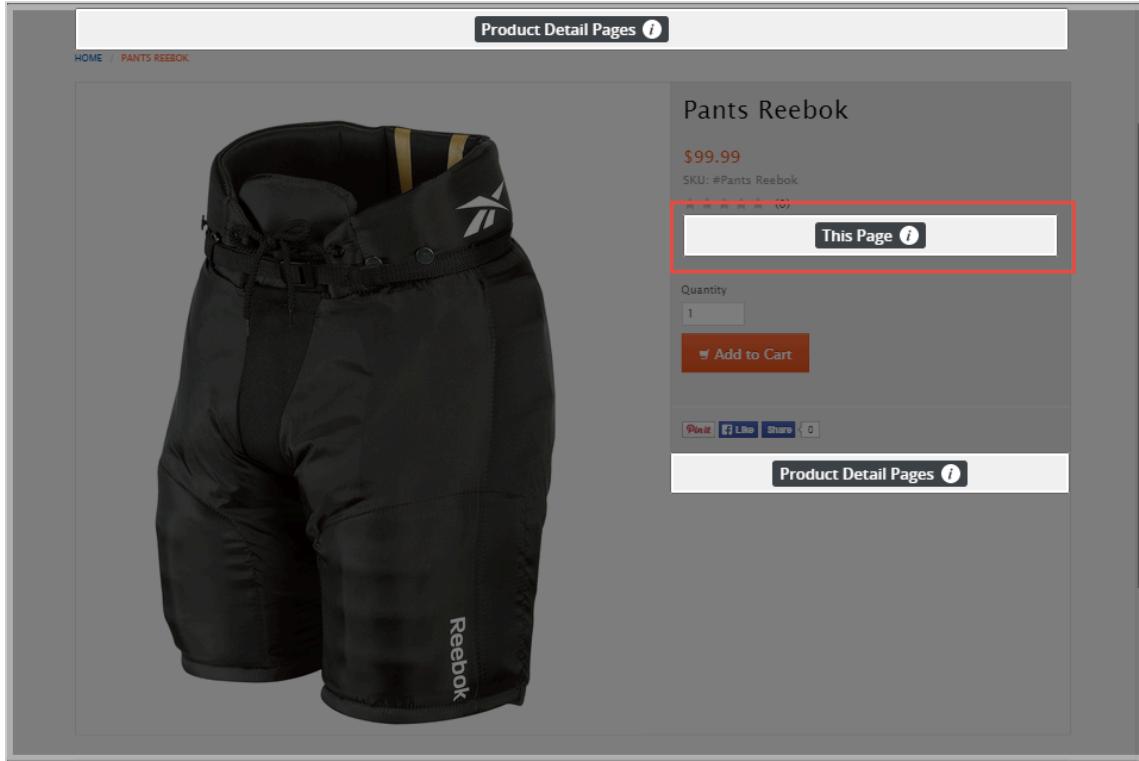
This Page Areas

You create **This Page** areas by setting the value of the `data-cms-area-filters` to `path`. The **This Page** area is the most specific of the three areas because the condition for its display is specific to the path of the page. The path is the URL, excluding host and query string parameters. For example, you modify the product detail template and include an area with the `data-cms-area` equal to `item_info` and `data-cms-area-filters` equal to `path`.

When you add content to the area it is tied to the path. For example, the URL to Reebok Pants is `http://website.name.com/pantsreebok`. The path to this product is `/pantsreebok`. Even though the page is a product detail page, when you add content to that area it displays only when a visitor goes to `/`

pantsreebok. The content does not display for any other product. The **div** segment for this area can be defined as follows:

```
<div data-cms-area="item_info" data-cms-area-filters="path">
```



A more sophisticated use of a **This Page** area is to apply it to a facet search page. This makes it possible to display content specific to the facet the visitor is viewing. For example, the path to a facet search of black items is **/color/black**. When you place content in a **This Page** area, the content displays only when a visitor views items with a facet filter of black.

To take this example a step further, the path to a facet filter of black pants is **/category/pants/color/black**. When you place content in a **This Page** area, the content displays only when a visitor views the pants page with a facet filter of black.

Facet Order in URLs

When URLs for a search page that includes facet filters are constructed, the facets in the URL are always listed in the same order, regardless of the sequence in which the visitor selects the facets. For example if one visitor filters first on **pants** and then on **black** and another visitor filters first on **black** and then on **pants**, the URL for both visitors is identical. This means that the path for black pants is always **/category/pants/color/black** regardless of the order the visitor applies the facet filters.

Templates and Areas for SCA

Template files for SCA pages are part of the corresponding module. These templates contain pre-defined Site Management Tools areas. This enables you to quickly add content and achieve a variety of page layouts without the need for creating custom page templates.

See the following topics for template specific areas:

- Header and Footer Module Default Areas
- Home Module Default Areas
- Breadcrumb Default Areas
- Facets Module Default Areas
- Item Details Module Default Areas
- Landing Page Default Areas
- Customizing Template Files

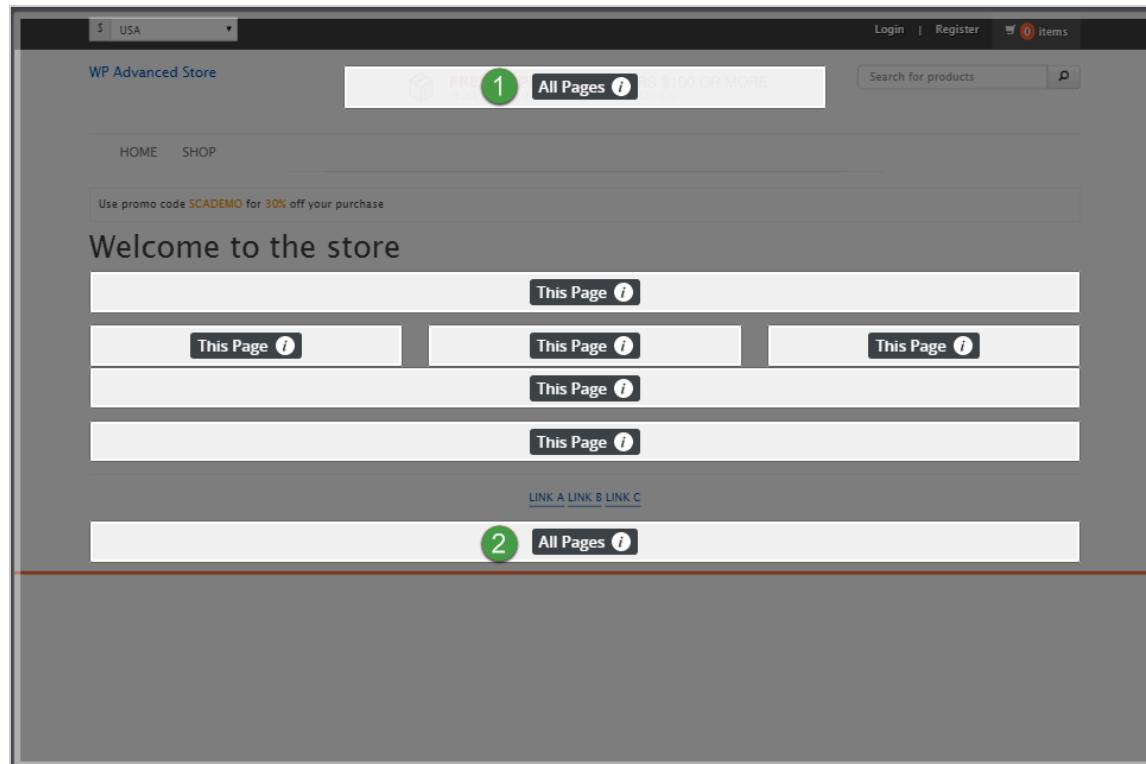
If you determine that the default page layouts do not fully meet your needs, you can customize the modules to create your own areas or make changes to the existing areas. For example, you may want to change the scope of an existing area to **Page Type** or to add a new **All Pages** area across all shopping pages.

For more information on adding areas to pages, see [Customizing Template Files](#).

Header and Footer Module Default Areas

The layout of the header and footer areas of your site pages is determined by the Header and Footer modules. Area **divs** placed in the template files for these modules adds areas to your page header and footer. The scope of these pre-defined areas is **All Pages**, which means any content you add to these areas displays across all of your shopping pages.

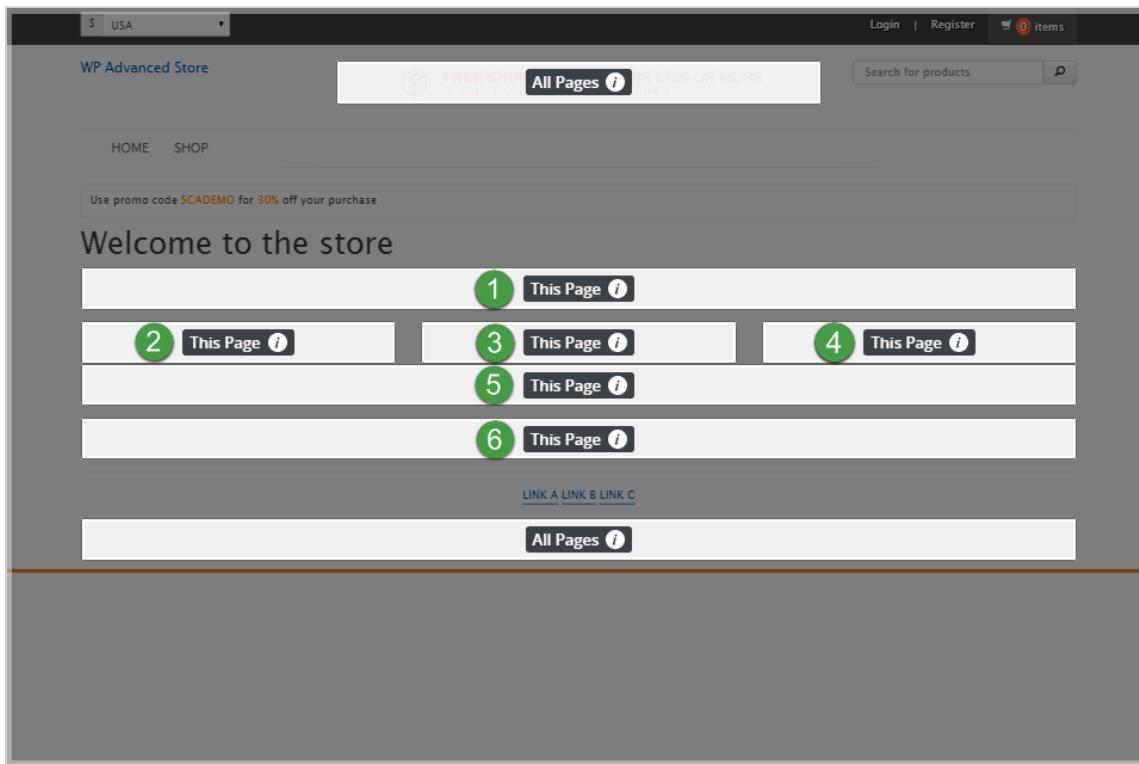
When a web page on your site is displayed, the page contains not only the areas defined in the header and footer modules but also those areas defined by the module that creates that page. In the following screen shot, you see the header and footer areas as they are displayed on the home page when adding content. These same areas also display on the search page and item detail pages.



1. The header template file is **header.tpl**, and it contains one area named **header_banner_top** with **All Pages** scope.
2. The footer template file is **footer.tpl**, and it contains one area named **global_banner_footer** with **All Pages** scope.

Home Module Default Areas

The layout of your site's home page is determined by the **home.tpl** template file in the Home@x.x.x module. In the following screen shot you see the defined areas. Note that you also see the **All Pages** header and footer module areas at the top and bottom of the page.



The **home.tpl** template file contains the following **This Page** areas.

1. **home_main**
2. **home_banner_1**
3. **home_banner_2**
4. **home_banner_3**
5. **item_list_banner_bottom**
6. **home_bottom**

Breadcrumb Default Areas

The breadcrumb navigation section of web pages on your site is created by the **global_viewsBreadcrumb.tpl** template file in the GlobalViews@x.x.x module. There are two default

areas. One area is located directly above the breadcrumb navigation and the other area is located directly below:

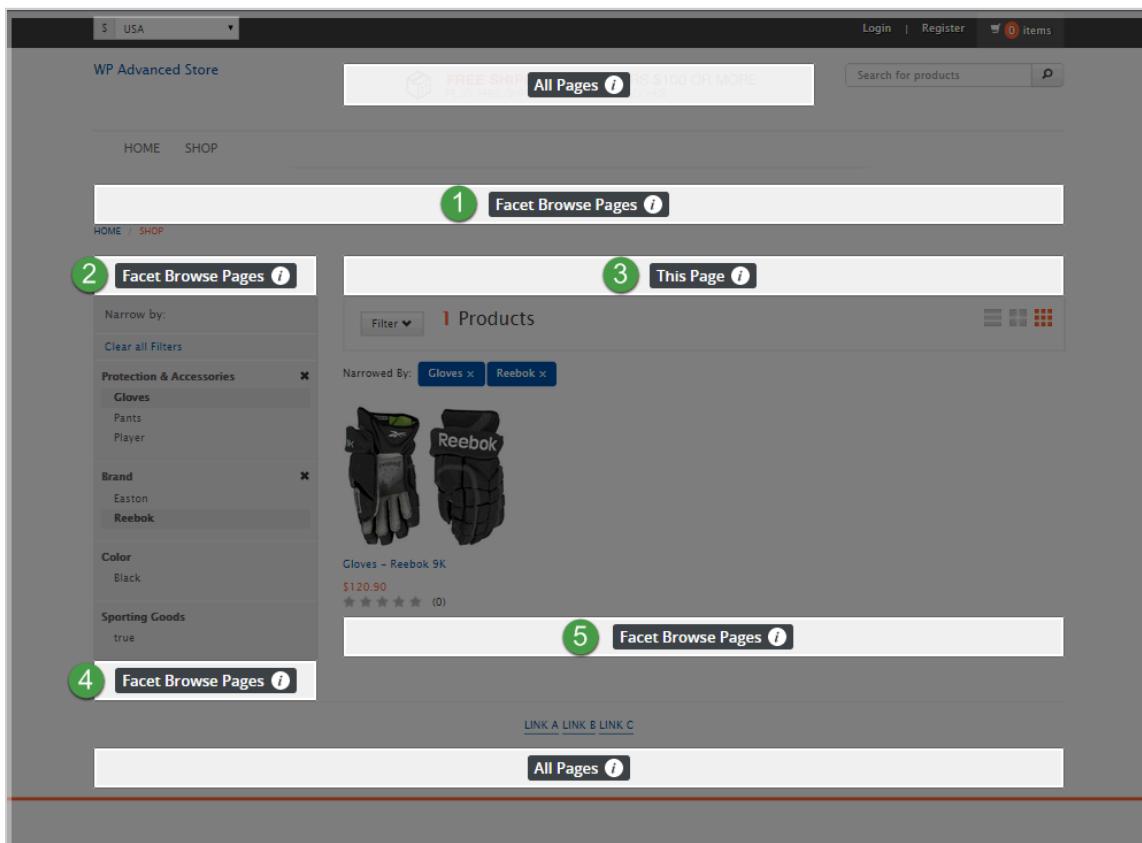


The `global_viewsBreadcrumb.tpl` template file contains the two following **All Pages** areas.

- `breadcrumb_top`
- `breadcrumb_top`

Facets Module Default Areas

The layout of your site's search page is determined by the `facets_facetBrowse.tpl` template file in the Facets@x.x.x module.



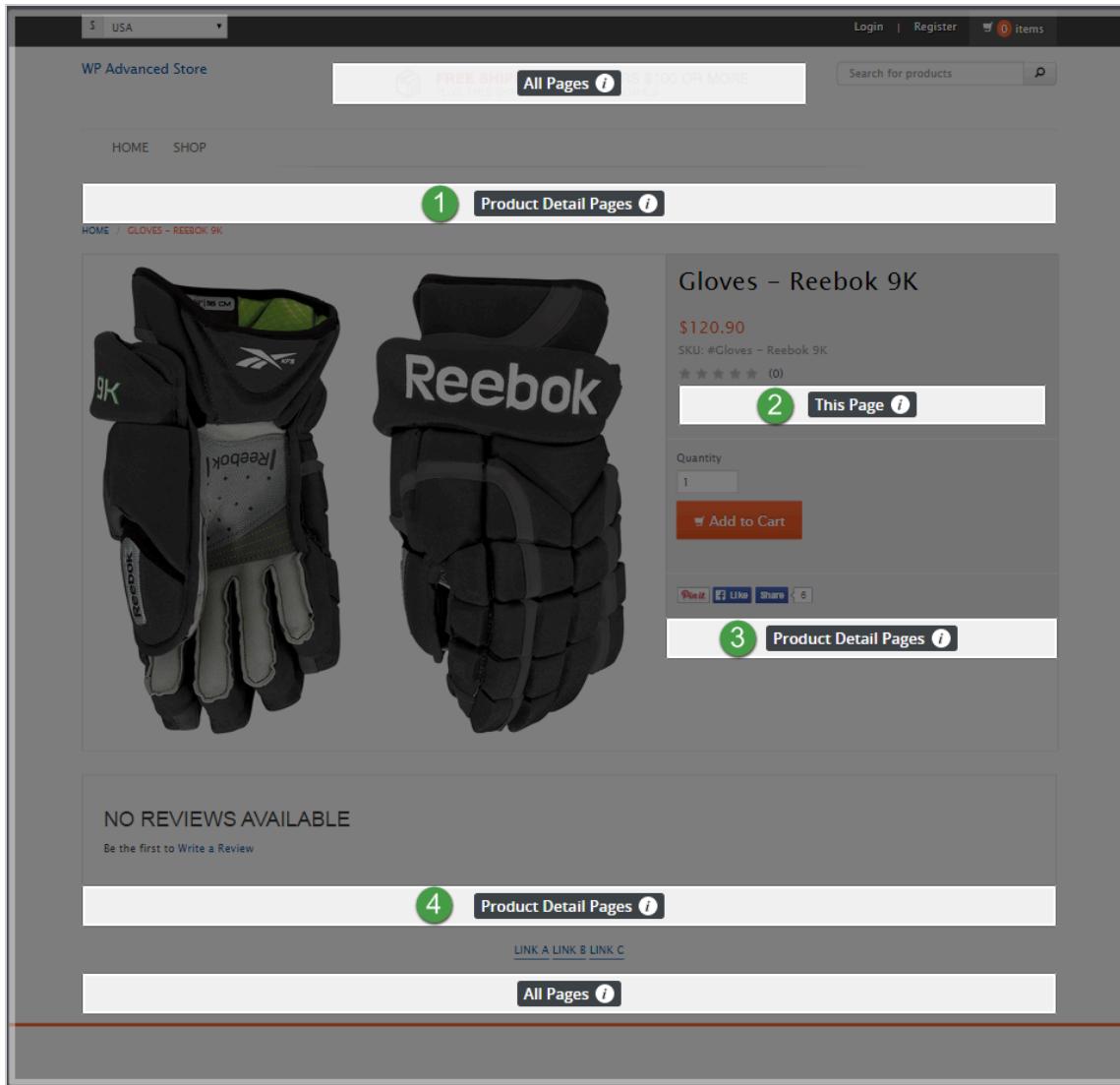
The `facets_facetBrowse.tpl` template file has four **Page Type** areas and one **This Page** area.

1. Page Type —`itemList_banner`
2. Page Type —`facetNavigation_top`

3. This Page —`item_list_banner_top`
4. Page Type —`facet_navigation_bottom`
5. Page Type —`item_list_banner_bottom`

Item Details Module Default Areas

The layout of the item detail pages on your site is determined by the `item_details.tpl` template file in the `ItemDetails@x.x.x` module.

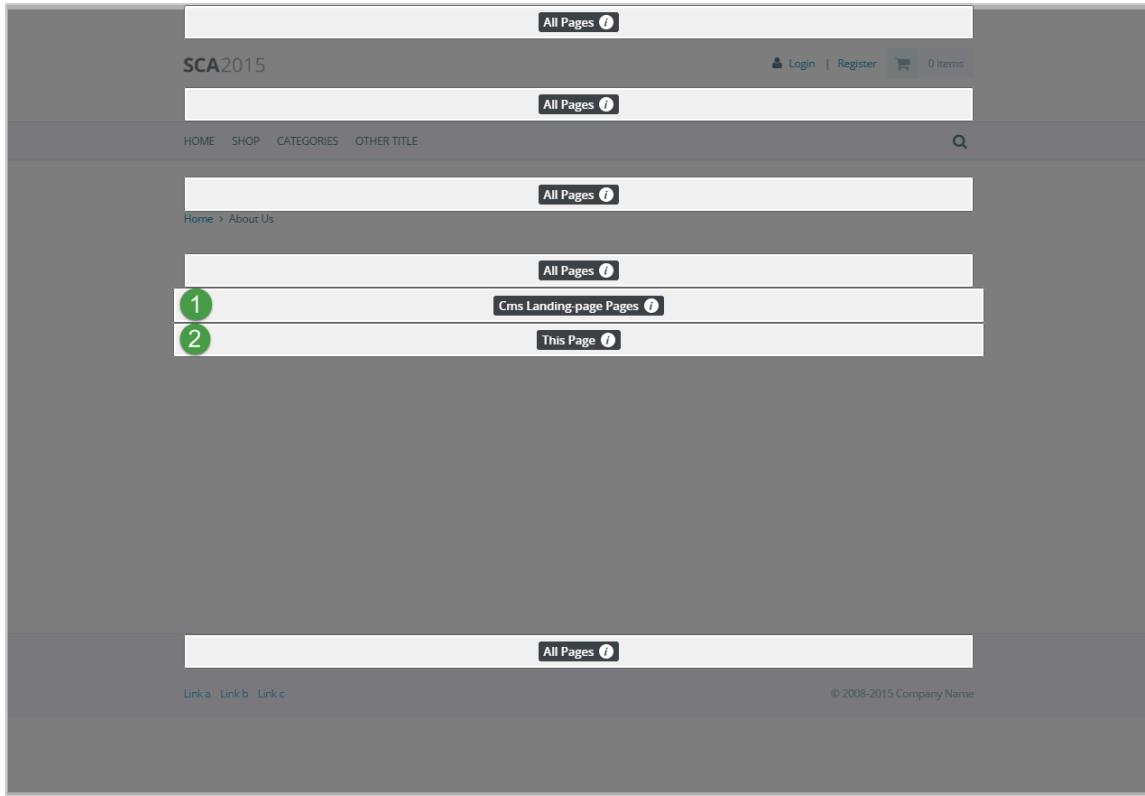


The `item_details.tpl` file contains three **Page Type** areas and one **This Page** area.

1. Page Type —`item_details_banner`
2. This Page —`item_info`
3. Page Type —`item_info_bottom`
4. Page Type —`item_details_banner_bottom`

Landing Page Default Areas

The layout of landing pages is determined by the `cms_landing_page.tpl` file in the CMSadapter@x.x.x module. In the following screen shot you see the defined areas. Note that you also see the header, footer, and breadcrumb areas that are defined in those modules.



The `cms_landing_page.tpl` file contains one **Page Type** area and one **This Page** area.

1. Page Type — `cms-landing-page-placeholder-page-type`
2. This Page — `cms-landing-page-placeholder-path`

Customizing Template Files



If you want to create a new area, remove an area, or change an area, you must customize the corresponding module and apply your changes to the relevant template file. For example, if you want to add more CMS areas to the item detail page, you must create a custom ItemDetails module that overrides the default `item_details.tpl` template file with a customized `item_details.tpl` template file. See [Core SuiteCommerce Advanced Developer Tools](#).

Module	Template File	Local Source File Module Template Location
home@x.x.x	home.tpl	Modules\suitecommerce\home@x.x.x\Templates
ItemDetails@x.x.x	item_details.tpl	Modules\suitecommerce\ItemDetails@x.x.x\Templates
Facets@x.x.x	facets_facet_browse.tpl	Modules\suitecommerce\Facets@x.x.x\Templates

Module	Template File	Local Source File Module Template Location
Header@x.x.x	header.tpl	Modules\suitecommerce\Header@x.x.x\Templates
Footer@x.x.x	footer.tpl	Modules\suitecommerce\Footer@x.x.x\Templates
GlobalViews@x.x.x	global_viewsBreadcrumb.tpl	Modules\suitecommerce\GlobalViews@x.x.x\templates

SMT Custom Preview Screen Sizes

If you want to preview your site for dimensions that are not already defined, you can create your own preview sizes by adding the dimensions to the adapter file. Additionally, you can override the default preview sizes so that only your custom preview sizes are available for selection.

The following sample code illustrates several custom preview screen sizes. Notice that each preview option is designated as either desktop, tablet, or phone. The attributes for each preview size includes the name, width dimension, and height dimension.

```
var setup = { // Config values the adapter can give the cms on startup.
    // Screen size preview override/extension
    screen_preview: {
        override_defaults: false,
        sizes: [
            {
                desktop: [
                    {
                        name: 'XL',
                        width: 2000,
                        height: 3000
                    },
                    {
                        name: 'XXL',
                        width: 3000,
                        height: 4000
                    },
                    {
                        name: 'Portrait',
                        width: 768,
                        height: 1024
                    }
                ],
                tablet: [
                    {
                        name: 'Huge',
                        width: 1300,
                        height: 2000
                    }
                ],
                phone: [
                    {
                        name: 'Massive',
                        width: 1400,
                        height: 2400
                    }
                ]
            }
        ]
    };
};
```

When you define custom dimensions for tablets and phones, Site Management Tools forces the width dimension to be greater than the height dimensions. If you define a dimension where the height is greater than the width, Site Management Tools swaps the width and height dimensions so that width is greater than height. This rule does not apply to desktop dimensions.

Override Defaults

If you want to make only your custom preview sizes available, in the CMS adapter, set the value of the `override_defaults` flag to `true`. Only custom dimensions will be available. When set to false, both default and custom preview dimensions are listed on the dimensions dropdown list.

Enabling Custom Preview Sizes and Overriding Defaults

To enable custom preview screen sizes or override default preview sizes, you must customize the CMS adapter file to include the code for the custom screen sizes and to set the value of the `override_defaults` flag. For information on customizing a module, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Working with SMT Landing Pages in a Sandbox Account

When working with Site Management Tools in a sandbox account, you must configure the URL for landing pages to reflect the domain of the sandbox environment. If you fail to set this configuration property, you receive a **Page Not Found** error when attempting to access an SMT landing page in the sandbox environment.

For configuration information, see [Configure Properties](#).

To set the Landing Pages URL

1. Select the domain to configure at Setup > SuiteCommerce Advance > Configuration.
2. In the SuiteCommerce Configuration record, navigate to the **Integrations** subtab and then the **Site Management Tools** subtab.
3. In the **Landing Pages URL** field, enter the URL for your sandbox account.
The format of the sandbox domain is your account specific domain appended with the unique identifier for the sandbox, for example, https://345678_SB1.app.netsuite.com.
4. Click **Save**.

See the help topics [NetSuite Sandbox](#) and [URLs for Account-Specific Domains](#) for additional information.

To set the Landing Pages URL (pre-Elbrus)

1. Create a custom CMSAdapter module to extend `CMSadapter.model.js`
2. Add a custom property to set `cmsPagesUrl` is as follows:

```
'https://345678_SB1.app.netsuite.com/api/cms/pages?site_id=' + siteSettings.siteid + '&c=' + nlapiGetCont  
ext().getCompany() + '&{}'
```



Note: This sample URL uses the account specific domain and the unique identifier for the sandbox.

For more information on customizing a module, see [Customize and Extend Core SuiteCommerce Advanced Modules](#). For information on deploying customizations to a sandbox account, see [Deploy to a NetSuite Sandbox](#).



Note: When the sandbox account is refreshed, your customization will be overwritten and will need to be redeployed to the sandbox account.



Important: If you make customizations in your sandbox account and then move those customizations to production, you must edit the **cmsPagesUrl** in the **CMSadapter.model.js** file to change the sandbox URL to the production URL.

Changing SMT to Use a Different Hosting Root

When you create a website, the Website Setup record specifies the HTML Hosting Root. When you implement Site Management Tools on that site, it utilizes the same HTML Hosting Root as the website. If you move the site to a different HTML hosting root, you will also need to create a new CMS SSP application. This application will utilize the new HTML Hosting Root so that SMT can continue to function and your content continue to display. This is required because Site Management Tools and the website must utilize the same HTML hosting root.

To configure SMT to use a different HTML hosting root:



Note: For the purpose of example only, the following procedure uses the sample folder name, **Custom Hosting Root**. Your custom hosting root can be named differently.

1. Publish any unpublished changes to your site. See the help topic [Publish Content](#) for more information.
2. Make a copy of your SSP application in the new HTML hosting root folder. This process includes:
 - Creating the directories in the file cabinet, for example, **Custom Hosting Root/SSP Applications**.
 - Copying the SSP application to the new **SSP Applications** folder.
 - Creating the new SSP application record and pointing it to the copy of the SSP application. See the help topic [Create a SuiteScript 1.0 SSP Application Record](#)
3. Copy the **NetSuite Inc. - CMS** folder to the **Custom Hosting Files/SSP Applications** folder in the file cabinet.



Important: The folder must be named **NetSuite Inc. - CMS**. Do not rename this folder or any of its subfolders.

4. Create a new SSP application record for SMT. See the help topic [Create a SuiteScript 1.0 SSP Application Record](#). Use the original SSP application record as your model:

SSP Application

		Edit	Back	Link to Site	Link to Domain	Actions ▾
NAME	CMS for Custom Hosting Root					
ID	webapp1100					
APPLICATION FOLDER	Custom Hosting Files: /SSP Applications/NetSuite Inc. - CMS/CMS					
URL ROOT	/CMS					
DESCRIPTION	SMT Implementation used by sites in the Custom Hosting Root.					

- a. Give the application a unique, CMS-related name.
 - b. Give the application a unique ID or leave the ID empty to use a system-generated ID.
 - c. Set the **Application Folder** to **Custom Hosting Root: /SSP Applications/NetSuite Inc. — CMS/CMS**.
 - d. Set the **URL Root** to **/CMS**.
 - e. Enter a description of the CMS SSP application. The description should provide enough information to prevent confusion.
 - f. Save the new application.
5. Edit the Website Setup record to change the **Default Hosting Root** to **Web Site Hosting Files : Custom Hosting Files**.
 6. Link the new SSP application record to your site.
 7. Clear the website cache.
 8. Go to your site and log in to Site Management Tools.



Important: This is a crucial step so don't forget it.

9. Add simple text content to any page on your site and publish the change.
10. After you confirm that the text content published correctly, delete the text content and publish the change to remove the text from your site.

Configuring Escape to Log In

Applies to: SuiteCommerce Web Stores | Site Management Tools | Elbrus | Kilimanjaro | Aconcagua

Available in the Elbrus release of SuiteCommerce Advanced and greater.

Use the following procedure to enable or disable the Site Management Tools Escape to Login feature, see the help topic [Disable Esc Key to Log In](#) for more information.

To enable or disable Escape to Login:

1. Select the domain to configure at Setup > SuiteCommerce Advance > Configuration.

2. In the SuiteCommerce Configuration record, navigate to the **Integrations** subtab and then the **Site Management Tools** subtab.
3. Select the **Disable Esc Key to Login** box to disable the Escape key for logging in or clear the box to enable the Escape key for logging in.

Internationalization of SMT Administration

Internationalization enables the website administrator to set the language for the Site Management Tools user interface. SMT uses the language preference selected in your NetSuite account. To select your language preference, see the help topic [Choosing a Language for Your NetSuite User Interface](#). This enables administrators to set their own language preferences when using SMT. For example, one administrator can use SMT in English and one administrator can use SMT in Spanish. SMT uses the language preference selected in the administrator's NetSuite account.

Internationalization does not affect the language for visitors to the website. Visitor language is controlled by language preference in SuiteCommerce Advanced. See the help topic [Web Site Language Preferences](#).

Currently, internationalization is only supported in SMT V3. To check your version of SMT, see the help topic [Determine Your Version of SMT](#).

All foreign languages available in NetSuite are available to select for internationalization.



Note: Custom Content Types are not currently supported for internationalization.

CMS Records for SMT

Applies to: Site Management Tools

When you add content to your website with Site Management Tools and publish it, the content and the information regarding how it is displayed on the site is saved as NetSuite records. These include the following:

- [Custom Record for CMS Content](#)
- [CMS Contents Record](#)
- [CMS Page Record](#)
- [CMS Page Type Record](#)

The custom record contains the actual content that you added to the site. For example, if you add a banner image to your site, there is a custom CMS_IMAGE record that includes the source for the image file, the alternate text for the image, and the link for the image. For each custom record, there is a corresponding CMS Content record. The CMS Content record details specific information about how and where that content from the custom record is displayed on the site.



Note: Only published content is available as NetSuite records. Unpublished content is not accessible in the NetSuite Admin.

Example 1. Example: Records Created when Adding Content

You add a banner image to the header area of your site. When you add the image, you specify the following information:

- Visibility Options
- Image File
- Alternate Text
- Link

The area where you add the image is defined in the page template file and the area attributes provide a couple of important pieces of information. These are the scope of the content which can be global, current page, or page type. In this example, the area is in a page header and it is defined with global scope. The area also has a unique name. In this example, the name is `global_banner_top`.

When you publish the content, a new CMS_IMAGE custom record is created. This record stores the content which is the image file source, alternate text, and link. A CMS Content record is also created. This CMS Content record links to the custom record with the custom record id. The CMS Content record includes the visibility date and times, area scope, area name, and several other data elements regarding the contents. See [Custom Record for CMS Content](#) and [CMS Contents Record](#).

Custom Record for CMS Content

The SMT Core Content Types bundle creates the custom records for the SMT core content types. These content types include:

- Text
- Image
- Merchandising Zone
- HTML



Note: The SMT core content type custom records are locked and cannot be edited or deleted.

The custom records are defined as follows:

Content Type	Custom Record Name	Custom Fields
Text	CMS_TEXT	clob_html
Image	CMS_IMAGE	string_link string_alt string_src
Merchandising Zone	CMS_MERCHZONE	exclude_cart exclude_current merch_rule_url merch_rule_count

		merch_rule_id merch_rule_template
HTML	CMS_HTML	clob_html

When you add content to your site and publish it, the content is saved as a custom record of one of these types. You can view or edit the record, and any edits you make to the custom record are immediately visible on the site. An example of when to edit the custom record is if you notice a misspelled word in a piece of text content. You can edit the custom record and correct the misspelling in the text. This makes the correction available immediately.

For each content custom record, there is also a CMS Content record that defines information specific to how and when the content is displayed on the site. See [CMS Contents Record](#).

 **Note:** If you create custom content types, a custom record is created for each custom content type. See [Custom Content Type](#).

To view the custom record for SMT content:

1. Go to Customization > List, Records, & Fields > Record Types.
2. In the **Record Types** list, locate the CMS Content Type record for the content you want to view. The core content type records are CMS_HTML, CMS_IMAGE, CMS_MERCHZONE, and CMS_TEXT. You may also have a content type record for any custom content type you enable.
3. Click **List** for the content type record. This displays the list of content for that record type.
4. Click **View**.

CMS Contents Record

The CMS Content record defines specific information about how each instance of SMT content is displayed. Each CMS Content record links to a corresponding custom record for content. For example for each CMS_IMAGE custom record, there is one corresponding CMS Content record. The CMS Content record specifies how and where content is displayed. The CMS Content record includes the following fields:

Name	The name is an internal identifier for the content. The name is not displayed to the visitor on the website. The name field is left empty for content created through SMT.
Description	This is a user defined description of the content. The description field is left empty for content created through SMT.
Site	This specifies the site for the content.
Content Type	<p>Identifies the content type. The core content types include:</p> <ul style="list-style-type: none"> ■ CMS_IMAGE ■ CMS_HTML ■ CMS_MERCHZONE ■ CMS_TEXT <p>Additional type values may also be available for custom content types.</p>
Settings	Identifies the ID for the custom record that stores the content. This ID links the custom record for the content instance to the corresponding CMS Content record.
Template	Determines the template used when rendering content. This should be set to default .

Match Type	The Match Type field is not used by Site Management Tools or SuiteCommerce at this time.
Change URL	Specifies the URL used when the content was created or changed. This URL does not necessarily reflect the only URL where the content is displayed. For example, content with page type and global context, can display on many pages. The URL in this field reflects only the URL of the page where the content was added or last edited.
Page Type Context	This corresponds with the SMT This Page area scope and is used for content that you want to place on a specific type of page such as a product detail page or a facet browse page. Available options are: <ul style="list-style-type: none"> ■ ProductDetails — denotes a product detail page. ■ facet-browse — denotes a facet browse page or a category page. ■ cms-landing-page — denotes a landing page. See This Page Areas .
Global Context	This corresponds with the SMT All Pages area type. When content is configured to display in all pages, this field contains the *. See All Pages Areas .
Path Context	The corresponds with the This Page area type and denotes the URL to the page. See This Page Areas .
Area Name	Specifies the name of the area on the page where this content is placed. Area name is the value of the data-cms-area attribute from the page template file.
Start Date	The visibility start date and time for the content.
End Date	The visibility end date and time for the content. This identifies the date and time when the content expires. When this field is empty the content never expires.

Edit CMS Content

Even though you can make changes to content by editing the CMS Content record, best practice for editing content is to make your edits in SMT. Editing the CMS content record directly is not recommended because, if done incorrectly, you can cause content to disappear from your site.

Add CMS Content

CMS Content records are created automatically when you add content to your site and publish it. The CMS Content record lets you add a new record directly from within NetSuite. Remember, each CMS Content record links to a corresponding CMS custom record. Although you can manually add CMS Content and CMS custom records to add content to your site, NetSuite recommends using SMT for adding or editing content.

Delete CMS Content

The CMS Content record lets you delete SMT content from your site. When you delete a CMS Content record, the corresponding CMS custom record is also deleted, and the content is immediately and permanently removed from the site.

To delete a CMS Content record:

1. Go to Lists > Web Site > CMS Contents.

2. Click **Edit** for the record you want to delete.
3. Select **Delete** from the **Actions** menu.
4. When asked if you are sure you want to delete the record, click **OK**.

CMS Page Record

The CMS page record gives you access to the NetSuite records for Site Management Tools landing and enhanced pages. These include landing and enhanced pages for both SMT version 2 and SMT version 3. These records include landing and enhanced pages that were created directly within SMT and also those created with the CMS Page record. The CMS Page record lets you:

- [Delete a CMS Page Record](#)

The CMS Page record gives you access to only the page, not to the SMT content on that page. You can add content to the page by logging in to SMT. The page attributes that you can set for a landing or enhanced page are the same attributes that you set for a page with SMT and include the following:

Name	The page name is an internal identifier for the page. This name is not displayed to the visitor on the website, but it does identify the page in NetSuite and in SMT.
Site	Determines the site for this page.
Type	Specifies if this record is for a landing page or an enhanced page. <ul style="list-style-type: none"> ■ 1 = landing page ■ 2 = enhanced page
Addition to Head	This field lets you specify additional information such as JavaScript or CSS to include in the <head> area of the page.
URL	Identifies the relative path to use for accessing this page on the website. The URL should contain only the relative URL with no slashes or spaces.
Template	Determines the template for the page.
Page Title	The page title populates the HTML title element. For SEO, titles should be descriptive and include target keywords to help improve the page's ranking on search results pages. Additionally, search engines often display this title on search results pages and link it to the page. Follow SEO best practice when creating page titles.
Page Heading	Specifies the heading for the page. If the page template file supports it, this heading is displayed on the page.
Meta Description	The Meta Description populates the HTML meta description element. The description is an important SEO tool and is often displayed on search results pages as a description of the page. Setting a meaningful and accurate description that contains your target keywords can help improve your ranking in search results pages. Use SEO best practice when writing descriptions.
Meta Keywords	The Meta Keywords populate the HTML meta keywords element for the page. It is important to follow SEO best practice when defining Meta Keywords for the page, because using them incorrectly can have a negative impact on the page's ranking on search results pages.
Start Date	The visibility start date and time for the content.
End Date	The visibility end date and time for the content. This identifies the date and time when the content expires. <p>When this field is empty, the content never expires.</p>

Delete a CMS Page Record

When you delete a CMS Page record, that page is removed and is no longer available. It can no longer be accessed by visitors, and it is no longer available for editing in SMT. Be sure to consider the SEO consequences of removing a page. If the page had been indexed by search engines, then you should create a 301 redirect so that search engines know the page has been permanently removed and the address of the page that replaces it. This helps prevent a negative hit on your search engine ranking. The 301 redirect also ensures that visitors who try to access the page directly are taken to an alternative page rather than just receiving a 404 Page Not Found error. For more information on redirects, see the help topic [Setting Up a URL Redirect](#).

If you delete a page that has not yet been published, then the record is removed and the unpublished changes list in SMT no longer lists the unpublished page. Since the page is unpublished there are no SEO considerations when you delete it.

To delete a CMS Page record:

1. Go to Lists > Web Sites > CMS Pages.
2. Click **Edit** next to the page link.
3. Point to the **Actions** menu and select **Delete** from the dropdown list.
4. When asked if you are sure you want to delete the record, click **OK**.

Landing Pages and Sitemap Generator

When using a new NetSuite account, there are two circumstances where new landing pages may not be included in the site map:

- You create CMS Landing Page records in NetSuite instead of with SMT
- You use the CSV import to create CMS Landing Page records

To ensure Landing Pages are included in the sitemap, you must log in to Site Management Tools. Logging in to SMT creates the necessary version records for the system to create the sitemap. There are no additional actions needed while logged in to SMT.

To avoid potential issues, you should use SMT to manage CMS content, CMS Landing Pages, and Commerce Categories.

For more information and best practices, see the help topic [Sitemap Generator](#).

For more information on CSV Import Files, see the help topic [Guidelines for CSV Import Files](#).

CMS Page Type Record

The CMS Page Type record gives you access to the NetSuite records for Site Management Tools page types. The CMS Page Type record gives you access to only the page type, not to the SMT content associated with that page type. A new page type record is also created when a user registers an extension.

You create a new CMS Page using your desired CMS Page Type, then add any desired SMT content to your new page.

The 19.1 release has one standard page type, landing page, that you cannot delete.

You can set the following the CMS Page Type Record attributes in NetSuite:

Name	The page name is an internal identifier for the page. This name is not displayed to the visitor on the website, but it does identify the page in NetSuite and in SMT. This field is also populated when an extension is registered.
Display Name	The display name is the external identifier for the page. This the name displays on the dropdown list when the creating a new page. This field is also populated when an extension is registered.
Description	This is a user defined description of the page type.
Base URL Path	The base URL path lets you customize the URL for all the pages of that type. For example, if you a have a page type of blog and a base URL path of blog, all blog pages are accessed by mysite.com/blog/[page url].
Custom Record Type	This lets you create a Custom Content Type (CCT) and associate with a page type. You create the new CCT and choose your customizations, this can be fields, Javascript code, CSS elements, and other custom module resource. This must be implemented through SuiteCommerce.
CMS Creatable	When CMS Creatable is set to yes, then the page type is available on the page type list when creating a new page. When CMS Creatable is set to No, then the page type is not available as an available type when creating a new page.
Inactive	When a page type is checked Inactive, it cannot be used to create a new page. Existing pages using this page type continue to work. Inactive page types remain visible in SMT for use in sorting and filtering operations.

Add a new CMS Page Type Record

The CMS Page Type record lets you create a new page type.

The CMS Page Type record enables you to only create the page type. You must then create a new page in SMT using the new page type. After you create a new page with the desired CMS Page Type, you can add content using SMT.

To add a CMS Page Type Record:

1. Go to Lists > Web Site > CMS Pages > New.
2. Enter the following information.
 - **Name** — Enter a name to identify this page in NetSuite. The name is not displayed to website visitors and only identifies the record in NetSuite. This field is also populated when an extension is registered.
 - **Display Name** — This is used to indicate the page type to the SMT admin. This field is also populated when an extension is registered.
 - **Description** — This is a user-defined description of the page type.
 - **Base URL Path** — The base URL path lets you customize the URL for all the pages of that type. For example, if you a have a page type of blog and a base URL path of blog, all blog pages are accessed by mysite.com/blog/[page url].
 - **Custom Record Type** — This lets you create a Custom Content Type and associate with a page type. You create the new CCT and choose your customizations. These can be fields, Javascript code, CSS elements, and other custom module resource. This must be implemented through SuiteCommerce.
 - **CMS Creatable** — Select **Yes** if the SMT admin can create pages of this type. Select **No** if the SMT admin cannot create pages of this type.
 - **Inactive** — Check the **Inactive** box to prevent users from creating pages of this type. Any existing pages of this type continue to display on the site.

3. Click **Save**.

Edit a CMS Page Type Record

The CMS Page Type record lets you edit an existing CMS Page Type. You can edit the attributes of your page type, but not any of the content on any page associated with the page type.

To edit a CMS Page Type Record:

1. Go to Lists > Web Site > CMS Page Type.
2. Click **Edit** next to the page type link.
3. Make your changes.
4. Click **Save**.

Delete a CMS Page Type Record

You can only delete a CMS Page Type record if no pages of that type exist. If you want to delete a CMS Page Type record from your NetSuite account, you must first delete all CMS Page records associated with that page type.

To delete a CMS Page Type record:

1. Go to Lists > Web Sites > CMS Page Type.
2. Click **Edit** next to the page type link.
3. Point to the **Actions** menu and select **Delete** from the dropdown list.
4. When asked if you are sure you want to delete the record, click **OK**.

Custom Content Type

 **Applies to:** Site Management Tools | Kilimanjaro | Aconcagua

Site Management Tools custom content type provides a platform for SuiteCommerce developers to create custom website applications or features that are enabled and managed through the SMT user interface. This enables SuiteCommerce users to implement custom content and expand functionality of their SuiteCommerce website with Site Management Tools. The process for implementing CCTs on your SuiteCommerce site can be broken down into two major activities. The first activity is to create your custom SuiteCommerce module. This consists of creating your JavaScript code, CSS, and other supporting custom module resources. These must all be implemented in SuiteCommerce following SuiteCommerce best practice.

See [Create a CCT Module](#) for best practice on creating a custom CCT module.

After you create your custom module, the next step is to setup the CCT for use in SMT. This process includes the following:

- Create the custom records and fields for your custom content type. See [Custom Record for Custom Content Type](#).
- Create the Content Type Record for your custom content type. See [CMS Content Type Record](#).
- Enable the Custom Content Type in the SMT Content Manager. See the help topic [Settings](#).

Custom Record for Custom Content Type

SMT custom content types use NetSuite's Custom Records and Fields for defining the settings and values required for the custom content type. For example, if you have a custom content type that displays an image with a text overlay, you must create a custom record for the content type and then create the custom fields for the values required by your content. In the following screen shot you see the custom record for the SC CCT ImageViewer. As part of implementing a CCT you associate this custom record with a corresponding CMS Content Type Record. See [CMS Content Type Record](#) for more information.

Custom Record Type

SC CCT ImageViewer

Save Cancel Reset | Change ID | Actions ▾

NAME *	SC CCT ImageViewer	SHOW OWNER: ON RECORD . ON LIST . ALLOW CHANGE	. ALLOW QUICK ADD																								
ID	customrecord_sc_cct_imageviewer	ACCESS TYPE	No Permission Required																								
INTERNAL ID	425		. ALLOW UI ACCESS																								
OWNER			. ALLOW MOBILE ACCESS																								
DESCRIPTION			. ALLOW ATTACHMENTS																								
			. SHOW NOTES																								
			. ENABLE MAIL MERGE																								
			. RECORDS ARE ORDERED																								
			. SHOW REMOVE LINK . ALLOW CHILD RECORD EDITING . ALLOW DELETE																								
			. ALLOW QUICK SEARCH																								
<ul style="list-style-type: none"> . INCLUDE NAME FIELD . SHOW ID <p>SHOW CREATION DATE: ON RECORD . ON LIST SHOW LAST MODIFIED: ON RECORD . ON LIST</p>																											
Fields • Subtabs Syblists Icon • Numbering • Fgrms • Online Forms Permissions Links Managers Translation • Child Records Parent Records History • System Notes •																											
<input type="checkbox"/> SHOW INACTIVES																											
<input type="button" value="New Field"/> <input type="button" value="Move To Top"/> <input type="button" value="Move To Bottom"/> <table border="1"> <thead> <tr> <th>DESCRIPTION</th> <th>ID</th> <th>TYPE</th> <th>LIST/RECORD</th> <th>TAB</th> <th>SHOW IN LIST</th> </tr> </thead> <tbody> <tr> <td>:: Message</td> <td>custrecord_sc_cct_iv_text</td> <td>Text Area</td> <td></td> <td></td> <td>No</td> </tr> <tr> <td>:: Vertical Align</td> <td>custrecord_sc_cct_iv_align</td> <td>List/Record</td> <td>SC CCT ImageViewer V Align</td> <td></td> <td>No</td> </tr> <tr> <td>:: Url</td> <td>custrecord_sc_cct_iv_imageurl</td> <td>Free-Form Text</td> <td></td> <td></td> <td>No</td> </tr> </tbody> </table>				DESCRIPTION	ID	TYPE	LIST/RECORD	TAB	SHOW IN LIST	:: Message	custrecord_sc_cct_iv_text	Text Area			No	:: Vertical Align	custrecord_sc_cct_iv_align	List/Record	SC CCT ImageViewer V Align		No	:: Url	custrecord_sc_cct_iv_imageurl	Free-Form Text			No
DESCRIPTION	ID	TYPE	LIST/RECORD	TAB	SHOW IN LIST																						
:: Message	custrecord_sc_cct_iv_text	Text Area			No																						
:: Vertical Align	custrecord_sc_cct_iv_align	List/Record	SC CCT ImageViewer V Align		No																						
:: Url	custrecord_sc_cct_iv_imageurl	Free-Form Text			No																						
Save	Cancel	Reset	Change ID Actions ▾																								



Important: The Access Type for the custom record must be set to **No Permission Required**.

Custom Fields for Custom Content Type

After you create the custom record, the next step is to define the custom fields. The custom fields you define here are displayed in the side panel in Site Management Tools when you add the custom content type to a page. In this example, the custom fields are used to define the information that is displayed by the custom content type. These fields include:

- **Message** — A text area that lets you enter text to overlay on the image.
- **Vertical Align** — Lets you select the vertical alignment for the text. Notice that in this example, the **Vertical Align** field is defined as a list/record type, and is associated with a custom list.
- **URL** — Identifies the source for the image.

To reference the custom fields in your CCT module, use the ID assigned to each custom field. In this example, the IDs for custom fields are:

- **custrecord_sc_cct_iv_text** — Message field

- **`custrecord_sc_cct_iv_valign`** — Vertical Align field
- **`custrecord_sc_cct_iv_imageurl`** — URL field



Note: Custom content types do not support the Password field type.

After your custom content type is fully implemented, it is displayed in SMT. The following screenshot illustrates the SC CCT ImageViewer custom content type.



Important: A Custom Content Type is available in SMT only after you have completed all implementation steps.

Tabs for Custom Content Type

Depending upon your custom content type and the number, type, or function of fields, you may want to create tabs to organize the fields. You can create tabs in the custom record for your custom content type and assign fields to the tabs. The tabs and field organization are reflected in the custom content types settings in the side panel in SMT.

For more information on custom records and fields, see the help topic [Custom Records](#).

Custom Content Type Records

Each time you use SMT to add an instance of a Custom Content Type to your site, a CMS content record and a custom record for the Custom Content Type is created. The CMS content record determines how and where the custom content is displayed, see [CMS Contents Record](#). The custom record for the content type stores the values for the custom content elements. In the example of the ImageViewer custom content type, these values include:

- Message

- Vertical Align
- URL

Although you can view or edit instances of the Custom Content Types in NetSuite, best practice is to make your changes to the Custom Content Type in SMT.

To view a Custom Content Type Record in NetSuite:

1. Go to Customizations > Lists, Records, & Fields > Record Types.
2. In the Record Types list, locate the record for your Custom Content Type and click **List**.
3. This displays a listing of all instances of that Custom Content Type.

CMS Content Type Record

The CMS Content Type record defines the different types of content for SMT. When you install the SMT Core Content Types bundle, the bundle creates the CMS Content Type records for the following core content types:

- CMS_HTML
- CMS_IMAGE
- CMS_MERCHZONE
- CMS_TEXT

CMS Content Type Record for CCT

When you create custom content types, you must manually create a CMS Content Type record for each custom content type. The CMS Content Type Record links to the custom record you created for your CCT. You must create the CMS Content Type Record before your custom content type is available in SMT.

To create a Content Type Record

1. Go to Lists > Website > CMS Content Types > New.
2. Click **New**.
3. In the **Name** field, enter a name for this content type. The name must be all lowercase and best practice is to use no spaces. The name identifies this record in NetSuite so make it descriptive of the custom content type. The name you specify here must be set as the value of the **id** property within the `registerCustomContentType()` method used to initialize your CCT module.
See [Create a CCT Module](#).
4. In the **Description** field, enter more details about this custom content type to explain its purpose.
5. In the **Icon Image Path** field, enter the path in the file cabinet for the icon you want to use for this content type. Only the scalable vector graphics (*.svg) format is supported. If you do not specify an icon, a default icon is used for the content type.
6. Specify the **Label** to use in SMT for this content type. The label identifies the content type to the user.
7. In the **Custom Record** field, enter the name of the custom record you created for this custom content type.
8. Click **Save**.

CCT Icon Requirements

Icons for custom content type should be a single-color SVG image. From a design perspective, the image should be simple but distinctive enough for users to recognize the icon at a glance and associate it with the content type.

Images that are comprised of SVG shape elements that can be filled, rather than SVG strokes, work best. Shapes should have no defined color. The icon can utilize strokes, but the element's fill attribute must be set to "none", and its stroke attribute must be set to "currentColor".

Icon styling should be done using presentational attributes, not CSS. Style elements present in the SVG image may be removed.

Required Settings

- Valid SVG
- One root element (<svg>) with valid value for xmlns attribute
- Root element's viewBox attribute is set to "0 0 48 48"
- Color
- Elements without a stroke must have fill set to "currentColor" or no fill attribute.
- Elements with a stroke must have fill set to "none" and stroke set to "currentColor".

Recommended Settings

- Desc element
- Should describe what the image contains or looks like
- Should be the first child of the root element
- Should generally only contain shape elements (circle, ellipse, line, path, polygon, polyline, rect), but may also contain structural elements (g, defs, use, symbol)

Not Permitted

- text, except when integral to a logo/branding where a non-text version is not available or would violate brand guidelines. Convert any included text to outlines.
- raster images
- style elements
- script elements
- animation elements
- gradient elements
- font elements
- filter primitive elements
- animation attributes
- event attributes
- conditional processing attributes

Overview

 **Applies to:** SuiteCommerce Web Stores

SuiteCommerce provides a fully functional application that you can use to implement your ecommerce solutions. Theme developers create HTML templates and Sass files as themes. The SuiteCommerce Base Theme is an excellent place to start building a theme and is available as a SuiteApp. Extension developers create extensions to introduce functionality to a site.

If you are developing a SuiteCommerce Advanced (SCA) site, you have access to the core SCA source code as well, although this comes with a different set of instructions and best practices you must follow.

This topic explains how to customize the following:

- [Themes](#)

Themes contain any number of HTML templates, Sass files, and assets, such as images and fonts. These are organized into modules and act as a single package that can be later applied to any domain linked to a SuiteCommerce site. Read this section if you are a theme developer (SuiteCommerce or the Aconcagua release of SCA or later).

- [Extensions](#)

Extensions introduce added functionality to your website through any number of JavaScript, SuiteScript, configuration JSON, and other files. You use the extension developer tools to build a baseline extension and build from there. Extensions can also include HTML and Sass. Read this section if you are an extension developer (SuiteCommerce or the Aconcagua release of SCA or later).



Note: The Extensibility API is only available to extension developers. This means that you must be building extensions for SuiteCommerce sites or sites implementing the Aconcagua release of SCA or later to access this API.

- [Theme and Extension SuiteApps](#)

This section explains how to bundle themes and extensions as SuiteApps, making them available for distribution to NetSuite accounts.

- [Core SCA Source Code](#)

This section is for SuiteCommerce Advanced developers only. Read this section if you are customizing objects that are not accessible using the Extensibility API. This requires using the core SCA developer tools and customization procedures to extend SuiteCommerce Advanced Source code. You must also read this section if you are implementing the Kilimanjaro release of SuiteCommerce Advanced or earlier.

SuiteCommerce Advanced Developers

As a SuiteCommerce Advanced developer, you have different options when customizing the application. The tools, practices, and procedures available all depend on your implementation.

Aconcagua Release of SCA and Later – If you are implementing the Aconcagua Release of SuiteCommerce Advanced have access to themes and extensions. You can alter a site's appearance using Sass and HTML files deployed or bundled as themes. To introduce new functionality to a site through JavaScript and SuiteScript, you can use the Extensibility API to build extensions using the SuiteCommerce Extension Framework.

However, if you are developing JavaScript, SuiteScript, or configuration objects that are not accessible using the Extensibility API, you must use the core SCA developer tools and follow SCA module customization practices.

Kilimanjaro Release of SCA and Earlier – If you are customizing the Kilimanjaro Release of SuiteCommerce Advanced or earlier, you must either migrate to the latest release of SCA or use the core SCA developer tools and follow specific procedures to customize SuiteCommerce Advanced modules. Themes and extensions are not available for these earlier releases of SCA.

For a list of components exposed using the Extensibility API, see the help topic [Extensibility Component Classes](#).

For information on using the core SCA developer tools and customizing SCA modules, see [Core SCA Source Code](#).

Themes

 **Applies to:** SuiteCommerce Web Stores

A theme is a special type of extension that affects a domain's design and appearance (layouts and styles). Themes contain any number of HTML templates, Sass files, and other assets that are available as published themes (bundled into a single SuiteApp) or deployed to a NetSuite account by on-site theme developers. After downloading the theme developer tools and creating your developer environment, you are ready to create your own themes.

This topic includes the following topics:

- [Themes Overview](#)
- [Develop Your Theme](#)
- [Best Practices for Creating Themes](#)
- [Theme Manifest](#)

Themes Overview

Benefits of Using Themes

The following list describes some of the benefits of using themes:

- Themes let non-technical users change the layout and styles of their web store by installing and activating any number of pre-developed themes from a marketplace.
- Themes allow any developer working with SuiteCommerce or SuiteCommerce Advanced (SCA) to create and manage their own themes and activate them for any domains associated with their site. Partners can also publish and distribute themes as bundled SuiteApps.
- Themes leverage the features and functionality of Site Management Tools (SMT). You can expose variables to the SMT Theme Customizer. This allows SMT administrators to further customize any theme using the SMT user interface.

Before You Begin

Be aware of the following important information:

- These procedures assume that you have already created your developer environment, fetched an active theme to use as a baseline, and are ready to build your theme. For more information, see [Theme Developer Tools](#).
- To develop a theme, you must have experience working with HTML, Sass/CSS, and Handlebars.js. Before customizing any theme or extension source code, read and understand the [Best Practices for Creating Themes](#).
- As a best practice use the SuiteCommerce Base Theme as a baseline for theme development. This is available as a SuiteApp. See the help topic [Install Your SuiteCommerce Application](#) for details on this SuiteApp.
- The examples presented in this section describe customizing Sass files. However, you can customize HTML files as well to suit your needs. You can also introduce new images and fonts as assets.
- After fetching the active theme using the developer tools, you can customize theme-related files directly within your Workspace/<THEME_DIRECTORY> folder.

- If you are implementing the Aconcagua release of SuiteCommerce Advanced, you must implement themes to customize Sass or HTML templates.

Develop Your Theme

As a theme developer, you start with a baseline theme and develop your own from there. Best practice is to start development using the SuiteCommerce Base Theme. This is available as a SuiteApp in NetSuite. Simply install this SuiteApp and activate it using the Manage Extensions Wizard. Use the theme developer tools to fetch the theme files into a theme development workspace and start developing.

Theme Development Tutorials

The following tutorials assume that you have already activated the SuiteCommerce Base Theme using the Manage Extensions Wizard and fetched theme files using the theme developer tools. If you are planning to override extension-related HTML and Sass to accommodate your theme, you must activate any related extensions as well. You should also familiarize yourself with the [Best Practices for Creating Themes](#).

Refer to the following topics for assistance before creating a theme:

- See [Fetch Active Theme Files](#) for details on fetching files for development.
- See the help topic [Themes and Extensions](#) for details on activating themes and extensions.

Step	Description	More Information
1	Create a baseline theme using the SuiteCommerce Base Theme. Follow the best practices related to the Base Theme when using this theme.	SuiteCommerce Base Theme
2	Edit existing theme Sass and HTML files.	Customize Pre-Existing Theme Files
3	Create new HTML, Sass, or asset files to meet your needs.	Add a New File to a Theme
4	If you want elements of your theme (HTML and Sass) to impact an active extension (use a new color, for example), you can create overrides that implement your changes to the active extension. These procedures explain how to introduce HTML and Sass changes to a deployed extension using the override method.	Override Active Extension Files
5	If you want your theme to be customizable using the Site Management Tools Theme Customizer, follow these instructions to expose variables and organize how those variables look in the SMT user interface. You can also create skins for your theme. Skins are predefined settings that change the appearance of a theme in a specific way. Follow these instructions to create skin preset files, which define new values for any number of exposed variables.	Set Up Your Theme for Customization in SMT
6	Understand the theme manifest. This is a JSON file that includes all the information required to compile resources for an active theme.	Theme Manifest
7	Throughout development, you may want to test your theme on a local server or on a test domain. You have the following options: <ul style="list-style-type: none"> When you are ready to test your theme, you can use the theme development tools to test on a local server. To see your theme on a development or production domain, you must deploy to your NetSuite account and activate the theme (and any extensions that include overrides) for the domain. 	Test a Theme on a Local Server Deploy a Theme to NetSuite

Step	Description	More Information
8	Activate your theme to view your theme to your NetSuite account.	Manage Themes and Extensions

Customize Pre-Existing Theme Files

When you download a theme's source files, you have access to the HTML, Sass, and assets used by that theme. You can customize the HTML and Sass files directly within your Workspace directory. Best practice, when making changes to existing Sass variables, is to edit the existing Sass files directly.

To customize a pre-existing theme file:

1. Open the theme directory in your top-level development directory.

For example:

`Workspace/<THEME_DIRECTORY>/`

2. Locate the subdirectory containing the file or files you want to customize.

For example, you want to customize the top margin of the **Add to Cart** button for this theme. You locate the `_cart-add-to-cart-button.scss` file in the theme's Cart module:

`Workspace/<THEME_DIRECTORY>/Modules/Cart@1.0.0/Sass/_cart-add-to-cart-button.scss`

3. Edit the HTML or Sass file within the associated module.

You can edit this file directly. See [Best Practices for Creating Themes](#) for details. In this example, you edit the `.cart-add-to-cart-button-button` class and change the `margin-top` value from `lv3` to `lv5`.

```
//...
// New code with new margin-top value:
.cart-add-to-cart-button-button{
    @extend .button-primary;
    @extend .button-large;
    width: 100%;
    font-weight: $sc-font-weight-semibold;
    margin-top: $sc-margin-lv5;
}

/* Previous Code:
.cart-add-to-cart-button-button{
    @extend .button-primary;
    @extend .button-large;
    width: 100%;
    font-weight: $sc-font-weight-semibold;
    margin-top: $sc-margin-lv3;
} */
//...
```

4. If you want to expose Sass variables to the Site Management Tools Theme Customizer, add the `editable()` function to expose any variables. See [Set Up Your Theme for Customization in SMT](#) for details.
5. Save the file.
6. Repeat this procedure in this fashion for any other theme-related HTML or Sass files you intend to customize.

When you are ready, use the theme developer tools to test or deploy your theme. See [Theme Developer Tools](#) for procedures on how to use these tools.

Add a New File to a Theme

When you customize a theme, you can also create new HTML, Sass, or asset files to meet your needs. This procedure includes editing a manifest file to ensure that your new file compiles at activation.

 **Note:** To reference assets within your code, use the HTML and Sass Helpers provided. See [Asset Best Practices](#) for more information.

To create a new file for a theme:

1. Create your new HTML, Sass, or asset file in the correct location, as required.

File Type	Location Within Workspace/<THEME_DIRECTORY>/
HTML	Modules/<MODULE>/Templates
Sass	Modules/BaseSassStyles/Sass/reference-theme
Assets	assets

2. Create each file as required. See [Best Practices for Creating Themes](#) for details.



Important: To avoid file name collisions, do not create any new files or folders that share the same name, even if they reside in different locations.

3. Save this file in the correct location within your directory structure.

For example, you want to add a new Sass file titled **_newSass.scss**. As a best practice, save this file in the following location:

Workspace/<THEME_DIRECTORY>/Modules/BaseSassStyles/Sass/reference-theme/_newSass.scss

4. Open the theme's manifest.json file.

For example:

Workspace/<THEME_DIRECTORY>/manifest.json

5. Include your new file in the appropriate location as required:

- If adding a template, list the file by application. Include the .tpl file in the **files** array of the appropriate application object (Shopping, MyAccount, or Checkout). When declaring your new file, include the path to the correct module. The order of your declaration does not matter.

For example:

```
//...
"templates": {
  "application": {
    "shopping": {
      "files": [
        //...
      ]
    }
  }
//...
```

- If adding a Sass file, list the .scss file in the **files** array of the **sass** object. You set up the Sass entry point in a later step. When declaring your new file, include the path to the correct module. Add this line in a location that makes the most semantic sense within the Sass hierarchy. For example:

```
//...
"sass": {
  "entry_points": {
    "shopping": "Modules/Shopping/shopping.scss",
    "myaccount": "Modules/MyAccount/myaccount.scss",
    "checkout": "Modules/Checkout/checkout.scss"
  }
  "files": [
    //...
  ]
}
//...
```

- If adding an asset, add your asset as part of the **files** array of the **img**, **font-awesome**, or **font** object, as appropriate. When declaring your new file, include the path to the correct folder (img/, font-awesome/, or font/). The order of your declaration does not matter. For example:

```
//...
"assets": {
  "img": {
    "files": [
      //...
    ]
  }
  "font-awesome": {
    "files": [
      //...
    ]
  }
}
//...
```

This ensures that the compiler includes your customizations. In this example, you are adding a Sass file. Therefore, you add **newSass.scss** as a dependency to the **files** array of the **sass** object.

Your manifest file might look similar to the following example:

```
"sass": {
  "entry_points": {
    "shopping": "Modules/Shopping/shopping.scss",
    "myaccount": "Modules/MyAccount/myaccount.scss",
    "checkout": "Modules/Checkout/checkout.scss"
  }
  "files": [
    //...
    "Modules/BaseSassStyles@x.y.z/Sass/reference-theme/_newSass.scss",
  ]
}
//...
```

- Save the manifest.json file.
- If your new file is an asset or template, you have no further action required. However, if your new file is a Sass file, follow these additional steps:

- a. Identify the application or applications impacted by your new Sass file.
- b. Edit the application entry point file to include the same dependency you introduced in the manifest file.

Application Impacted	Application Entry Point
Shopping	Modules/Shopping/shopping.scss
My Account	Modules/MyAccount/myaccount.scss
Checkout	Modules/Checkout/checkout.scss

In this example, the Cart module impacts the Shopping application. Therefore, you open the Shopping entry point file and include the dependency. Your Shopping.scss file should look similar to the following example:

```
//...
@import "../BaseSassStyles@x.y.z/Sass/reference-theme/newSass";
//...
```

Just as you did in the manifest.json file, place this file in such a manner that makes semantic sense within the Sass hierarchy you are customizing.

- c. Save the application Sass file.

When you are ready, use the theme developer tools to test or deploy your theme. See [Theme Developer Tools](#) for procedures on how to use these tools.

Override Active Extension Files

When creating a theme, you can customize the HTML and Sass of any active extensions for a domain and deploy them with your theme customizations.

When you run the `gulp theme:fetch` command, you download the theme Sass, HTML, and asset files. You also download all extension-related HTML and Sass files. The developer tools place these in the appropriate folder within the Workspace/Extras/extension directory.

This section explains how to use the **Override** method to customize HTML and Sass files related to extensions.



Important: Extension-related files placed in the Extras directory are provided as a reference only. Never edit these files directly. Instead, use the Override method described in this section.

Example

You activate a theme for your domain. You also activate an extension for the same domain. You create your theme to include a new color variable. You want the extension that is active with your theme to use this same variable. However, the extension's Sass does not include the new variable. You must customize the extension's Sass files to include this color variable using the Override method. This ensures that the active extension includes your Sass variable, without actually changing the extension's code. Instead, you override it.

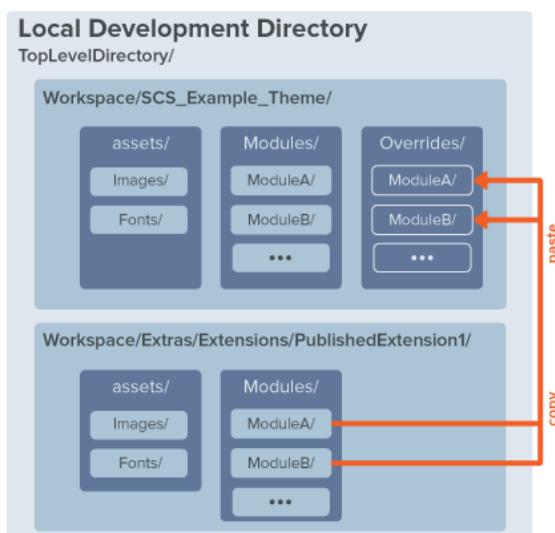
All of your theme customizations and extension overrides are maintained as part of your theme. Therefore, when you later activate your theme and the extension for your domain, the active extension

includes your new color variable. This does not change the extension. Instead, this change belongs to the theme.

The Override Method

To customize extension-related HTML and Sass files, you must use the Override method. This involves placing **copies** of source HTML and Sass from your Extras directory into your theme directory and making your customizations there. When you deploy your changes, the theme development tools detect the new files and initiate an override within the manifest.

Note: When you downloaded an extension's HTML and Sass files, the theme developer tools placed them in your Extras/Extensions directory, sorted by extension. These are for reference only. At the same time, the developer tools created an identical directory structure in your theme's Overrides directory.



Important: You cannot override asset files. To introduce new assets for your template customizations or extension overrides, add them as you would for any new files within the template directory, then reference the assets using the HTML and Sass Helpers. See [Add a New File to a Theme](#). See [Asset Best Practices](#) for details on using helpers.

To customize an extension using the Override method:

1. Locate the source file you want to override.

For example, you want to override the _Pub-Extend-Error.scss file of the **Published Extension 1** extension. Therefore, the file you want to override is in the following location:

`Workspace/Extras/Extensions/PublishedExtension1/Modules/
PubExtendModule@1.0.0/Sass/_Pub-Extend-Error.scss`

2. Copy the source file to your operating system's clip board.
3. Paste a copy of the file in the corresponding location within the theme's Overrides directory.

For example:

`Workspace/<THEME_DIRECTORY>/Overrides/Modules/PubExtendModule@1.0.0/Sass/
_Pub-Extend-Error.scss`



Important: Do not rename these files. These files must share the same name for the Override method to function correctly.

4. Open your new file in your Overrides directory.
5. Follow best practices to customize your new file. See [Best Practices for Creating Themes](#) for important details on customizing these files.
6. Repeat this procedure for all extension-related files you intend to customize.

When you are ready, use the theme developer tools to test or deploy your theme. See [Theme Developer Tools](#) for procedures on how to use these tools.

Set Up Your Theme for Customization in SMT

As the theme developer, you edit your theme's Sass files and expose variables for customization. This can include, at your option, preset files (JSON) that define skins for each theme. Within your Sass structure, you can also establish a schema for displaying each variable in the SMT user interface.

Before reading the topics in this section, familiarize yourself with the best practices outlined in [Sass Best Practices](#)



Note: Site Management Tools Version 3 includes the Theme Customizer feature. This lets SMT administrators customize styles and layouts of an active theme based on the settings that you define during theme development. SMT administrators can apply changes to individual settings or apply skins to change multiple settings at one time.

Expose Sass Variables for Customization

You control the variables that your theme exposes for customization by introducing metadata into your Sass files. You expose variables using the `editable()` function as described in this section.

To expose a variable for customization:

1. Open the Sass file containing the variable you want exposed.
2. Create an inline comment (`//`) or a block comment (`/* ... */`) immediately following the variable declaration.

You can use either comment method. For example:

```
$sc-primary-color: red; //
```

```
$sc-primary-color: red; /* */
```

3. Use the `editable()` function within your comment tags to declare the variable as `editable`. See [The editable\(\) Function](#) for details. This function is required.
 4. Save the file.
- If you are creating a new Sass file, you must declare the file within the manifest file and the appropriate application entry point. See [Add a New File to a Theme](#) for details.
5. Repeat this procedure for every variable you want to expose.

For exposed variables to appear in Site Management Tools, you must declare the Sass file containing the variable and its metadata in the manifest file and entry points for each application using that variable.

For example, you can define and declare a Sass variable that works in Checkout only. You add the containing Sass file to the theme manifest and the Checkout entry point. Later, when viewing your theme in the SMT Theme Customizer, your variable only appears in the SMT user interface when navigating to Checkout. If you want your variable to appear in multiple applications, you must declare the new file within each. Limiting global definitions to the BaseSassStyles module helps ensure functionality of global variables.

When you are ready, use the theme developer tools to test or deploy your theme. See [Theme Developer Tools](#) for procedures on how to use these tools.

The `editable()` Function

The metadata to expose a variable takes the form of an `editable()` function call wrapped inside a comment in your Sass files. This function call accepts a JSON object as its single argument with the following properties:

- **`type`** – declares the variable type. This property is required.

Possible values include:

- **`color`** – displays a color picker in SMT.
- **`string`** – declares a value as a string (font-weight, font-style, thumbnail size, etc.).



Important: SMT only supports `color` and `string` as values for the `type` property. If you wish to declare a number value or a font, you must do so as string.

- **`label`** – provides a formatted string to reference the variable in the SMT user interface. If not defined, SMT displays the Sass variable name as the label.
- **`description`** – describes the variable in a long-formatted string. This property is optional.
- **`options`** – provides a list of options as an array for the variable. Use this property to limit possible values using a dropdown list. See [Limit Selections](#). This property is optional.

The following code snippet is an example of how to expose a color variable using inline notation:

```
$sc-primary-color: red; // editable({ "type": "color", "label": "Primary Color" })
```

You can also use the `editable()` function with block notation:

```
$sc-primary-color: red; /* editable({
    "type": "color",
    "label": "Primary Color",
    "description": "Used mainly for 'call to action' elements."
}) */
```



Important: The comment must start immediately after the variable declaration. This is essential for the Sass preprocessor to relate the variable name with the declared metadata in the comment.

Limit Selections

You can use the `options` property of the `editable()` function to specify one or more values within a dropdown list in SMT.

For example, you can declare a list of available font weights as selections within a list instead of requiring SMT administrators to manually enter a numeric value as a string.

In this example, each option's **value** property is the value to be processed for **\$base-font-weight**, and the **text** property is the corresponding option as viewed in the dropdown list:

```
$base-font-weight: 200; /* editable({
  "type": "string",
  "label": "Font Weight",
  "options": [
    {"value": 200, "text": "light"},
    {"value": 400, "text": "normal"},
    {"value": 800, "text": "bold"}]
}) */
```

You can also apply this option to colors. Using the **option** property on a color requires the SMT administrator to choose from a list of possible colors instead of using a color picker. Note that the option value must be a string in this case.

The following example displays a list of color options:

```
$feedback-color: white ; /* editable({
  "type": "color",
  "label": "Feedback Color",
  "options": [
    {"value": "white", "text": "normal"},
    {"value": "black", "text": 'contrast'},
    {"value": "#eddeded", "text": 'low contrast'}
  ]
}) */
```

Organize Variables for Display in SMT

When you expose Sass variables for customization, SMT displays each exposed variable in the side panel. Without any organizational structure, these variables do not display in any meaningful, intuitive way. To aid SMT administrators when customizing their theme, you can define how each exposed variable displays in the SMT side panel. You do this by creating group metadata using the **group()** function as described in this section.

You can create group metadata in any Sass files within your theme.

To group variables in the SMT side panel:

1. Open the Sass file within your theme development files that contains the exposed variable you want to group.

You can place this in any location within the Sass file.

2. Use the **group()** function within comment tags to create an organization scheme.

Build your group schema here. See [The group\(\) Function](#) for details.

3. Save the file.

If you are creating a new Sass file, you must declare the file within the manifest file and the appropriate application entry point. See [Add a New File to a Theme](#) for details.

4. Repeat this for every new variable you want to expose for your theme.

When you are ready, use the theme developer tools to test or deploy your theme. See [Theme Developer Tools](#) for procedures on how to use these tools. Note that you must also activate your theme to make these changes available for SMT. See the help topic [Manage Themes and Extensions](#) for details.

The group() Function

You introduce the `group()` function within comments, as you do with the `editable()` function. However, `group()` does not need to be located immediately after a variable declaration within a Sass file.



Note: Functionally, you can add the `group()` function at any point within your theme's Sass files. However, as a best practice, introduce `group()` in the same file that contains the associated variable.

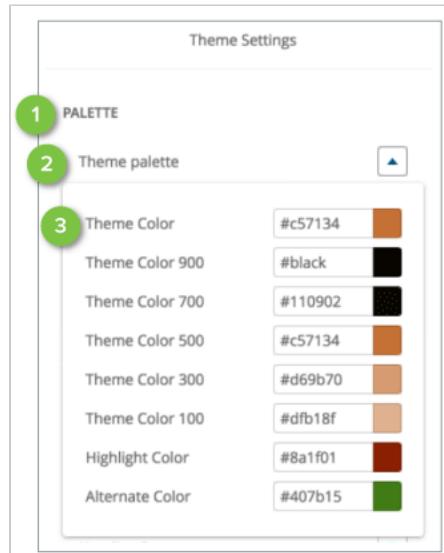
Each `group()` function call accepts a JSON object as its single argument with the following properties:

- `id` – declares the group ID. This is used to define and reference the group.
- `label` – provides a formatted string for the group to be shown in the SMT Theme Customizer.
- `description` – describes the group in a long-formatted string. This property is optional.
- `children` – provides an array of variable names or group IDs (for a nested subgroup). If declaring a variable, you must precede the variable with `$`. The SMT user interface mirrors the order of any children declared here.



Note: Any variables or subgroups listed here do not need to be defined before the group definition to be referenced as a group children.

This function lets you declare groups and organize your Sass variables in an intuitive way within the SMT side panel.



1. Parent (Group)
2. Child (Subgroup)

3. Child (Variable)

A top-level (parent) group displays as a heading in the SMT side panel, such **PALETTE**. Should a further subgroup be necessary, nested children appear as expandable/collapsible subgroups, such as **Theme palette**. Children of a subgroup are variables exposed for editing. SMT only supports two group levels.

Example

The following code declares one parent group called **Pallette-color** and four children to create subgroups. Each of these children is defined later in the code. Any children of the subgroups are declared variables.

```
/*
group({
  "id": "palette-color",
  "label": "Palette",
  "description": "",
  "children": [
    "theme-palette",
    "neutral-shades",
    "text-colors",
    "container-colors"
  ]
})
*/


/*
group({
  "id": "theme-palette",
  "label": "Theme palette",
  "description": "",
  "children": [
    "$sc-color-theme",
    "$sc-color-theme-900",
    "$sc-color-theme-700",
    "$sc-color-theme-500",
    "$sc-color-theme-300",
    "$sc-color-theme-100",
    "$sc-color-primary",
    "$sc-color-secondary"
  ]
})
*/

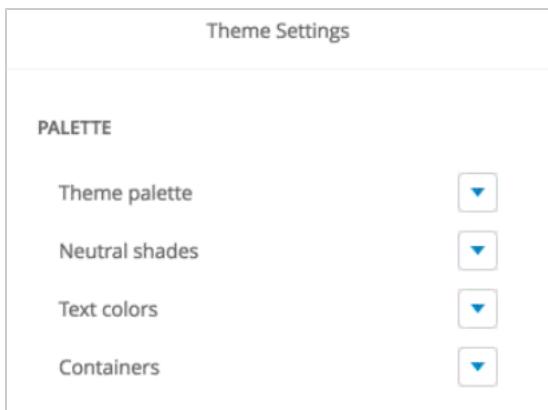

/*
group({
  "id": "neutral-shades",
  "label": "Neutral palette",
  "description": "",
  "children": [
    "$sc-neutral-shade",
    "$sc-neutral-shade-700",
    "$sc-neutral-shade-500",
    "$sc-neutral-shade-300",
    "$sc-neutral-shade-0"
  ]
})
*/
```

```

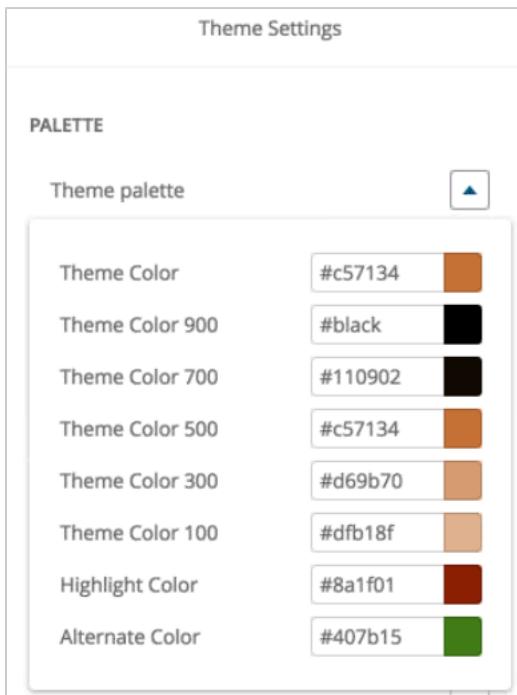
        ]
    })
/*
group({
    "id": "text-colors",
    "label": "Text",
    "description": "",
    "children": [
        "$sc-color-copy",
        "$sc-color-copy-dark",
        "$sc-color-copy-light",
        "$sc-color-link",
        "$sc-color-link-hover",
        "$sc-color-link-active"
    ]
})
*/
/*
group({
    "id": "container-colors",
    "label": "Containers",
    "description": "",
    "children": [
        "$sc-color-body-background",
        "$sc-color-theme-background",
        "$sc-color-theme-border",
        "$sc-color-theme-border-light"
    ]
})
*/

```

The preceding code, after it is deployed and activated for a domain, displays in the SMT side panel as depicted in the following images. The first image shows the parent group, **Palette**, and the four nested subgroups: **Theme palette**, **Neutral shades**, **Text colors**, and **Containers**.



Children of the parent group are the nested subgroups. Children of each subgroup must be variable names because SMT does not support more than two group levels. These appear as expandable/collapsible groups. SMT administrators can now edit the settings as they please.



Note: The design architecture style definitions used to maintain the Sass structures use derived/calculated values. By exposing derived colors, as in this example, any UI selections of the base value results in changes of any derived values automatically. SMT admins can further edit each derived value within the UI as well, assuming those variables are exposed for editing. See [Style Definitions](#) for more information on how these Sass structures work.

Create Skins

What is a Skin?

Generally speaking, a skin is a selectable set of predefined settings that change the appearance of a theme. One theme can include any number of skins that SMT administrators select to change that theme's appearance. For example, a theme might include multiple skins with a different color scheme for specific seasons.

- **Theme** – a ready-made package that can be activated for any number of domains. A theme can include any number of skins.
- **Skin** – a set of predefined variables that lets SMT administrators alter a theme in a specific way.

From a development perspective, you define a skin using a JSON file. Each file contains the variables and corresponding values that change when the skin is applied using SMT. You create one file in the theme's Skins directory for each skin you want available. When you run the `gulp theme:local` or `gulp theme:deploy` command, the developer tools update the theme's manifest file, automatically adding a skin for each file located in Skins directory.

When you deploy and later activate the theme for a domain, your presets appear as selectable skins in the SMT user interface. When SMT administrators apply your skin, the application compiles the values, and the theme incorporates the changes.



Important: If you apply a skin, then make changes to one or more variables, you will see your changes in the theme preview. However, if you then select a new skin, your changes to the individual variables will be lost.

Important Information

Be aware of the following information when creating skins:

- When developing your skin, include only variables that are exposed for customization. If your skin includes variables that are not editable, they will not change when the skin is applied. [Expose Sass Variables for Customization](#) for details.
- A theme can include any number of skins or no skins at all. If you do not create any skin preset files, SMT does not provide the option to select any.
- Any Sass variables exposed to SMT are available for customization at any time using the SMT Theme Customizer. A theme does not need to include skins for exposed variables to be editable. Likewise, after an SMT administrator selects a skin, they can make their own changes to individual variables.
- When SMT administrators apply changes to any exposed variables, those changes apply to the domain, but they do not overwrite the predefined skin or the theme's Sass files. The only way to permanently change a Sass variable or save changes to a skin is to edit the theme's development files or create a new theme using the developer tools.

The Skin Preset File

Creating a skin involves building a JSON preset file. This file contains the variables you want your skin to change. Each variable listed includes the new value to use when the skin is applied.

To create a skin preset file:

1. Expose all Sass variables you want your skin to override. See [Expose Sass Variables for Customization](#) for instructions.
2. Navigate to your theme's Skins directory.

This directory is located in your top-level theme development directory. For example:

```
<Top-LevelDevelopmentDirectory>/
  Workspace/
    Skins/
```

3. Create a new JSON file.

Name this file in an intuitive manner. Do not include spaces or special characters.

For example:

```
.. /Workspace/Skins/winter_skin.json
```

4. Open this file and define the Sass variables you want your skin to change and include the values your skin changes.

For example, your winter skin might look like the following:

```
{
  "$sc-color-body-background": "white"
  , "$sc-button-primary-text-color": "white"
  , "$sc-button-primary-hover-text-color": "white"
  , "$sc-button-primary-active-text-color": "white"
  , "$sc-button-primary-text-transform": "none"
```

```

    "$sc-button-secondary-text-color": "white"
    "$sc-button-secondary-hover-text-color": "white"
    "$sc-button-secondary-active-text-color": "white"
    "$sc-button-secondary-text-transform": "none"
    "$sc-button-tertiary-hover-text-color": "white"
    "$sc-button-tertiary-active-text-color": "white"
    "$sc-button-tertiary-text-transform": "none"
    "$sc-button-large-line-height": "1"
    "$sc-button-large-letter-spacing": "1px"
    "$sc-button-large-mobile-width": "100%"
    "$sc-button-large-desktop-width": "auto"
    "$sc-button-medium-line-height": "1"
    "$sc-button-medium-letter-spacing": "0.5px"
    "$sc-button-medium-mobile-width": "100%"
    "$sc-button-medium-desktop-width": "auto"
    "$sc-button-small-line-height": "1"
    "$sc-button-small-letter-spacing": "normal"
    "$sc-button-small-mobile-width": "auto"
    "$sc-button-small-desktop-width": "auto"
    "$sc-body-line-height": "1.6"
    "$sc-h1-line-height": "1.2"
    "$sc-h2-line-height": "1.2"
    "$sc-h3-letter-spacing": "0"
    "$sc-h4-line-height": "1.4"
    "$sc-h5-line-height": "1.4"
    "$sc-h6-line-height": "1.4"
    "$sc-blockquote-line-height": "1.6"
    "$sc-color-primary": "#e23200"
    "$sc-color-secondary": "#15607b"
    "$sc-neutral-shade": "#4D5256"
    "$sc-color-theme": "#97CCDF"
    "$sc-color-link": "#0067b9"
    "$sc-color-link-active": "#0067b9"
}

```



Important: When creating a skin preset file, include only variables that are exposed for editing within your theme's Sass files. If you include a variable that is not exposed, that value does not change when the skin is applied to the theme. See [Expose Sass Variables for Customization](#).

5. Save the JSON file.
6. Include your new skin in the extension manifest file. See [Add a Skin to the Theme Manifest](#).

Add a Skin to the Theme Manifest

For your skin to appear in the SMT Theme Customizer, it must be declared in the theme's manifest. The developer tools do this automatically when you run either `gulp theme:local` or `gulp theme:deploy`.



Note: You can also edit the manifest file manually, following the example in this section. The theme's `manifest.json` file is located at `../Workspace/<Theme_Directory>/manifest.json`.

To update the theme manifest file:

1. Perform one of the following Gulp commands:

- `gulp theme:local` – the developer tools update the manifest and start the local server.
- `gulp theme:deploy` – the developer tools update the manifest and deploy the theme to NetSuite.



Note: Running `gulp theme:local` updates the manifest, but it does not deploy anything to NetSuite servers. You must deploy your theme files to NetSuite to view your skins in the SMT Theme Customizer.

2. If necessary, you can edit the `manifest.json` file to provide a more visually appealing value for the `name` property.

Any edits you make to the `manifest.json` file do not apply to your domain until you redeploy your theme.

When you are ready, use the theme developer tools to deploy your theme. See [Theme Developer Tools](#) for procedures on how to use these tools. Note that you must also activate your theme to make these changes available for SMT. See the help topic [Manage Themes and Extensions](#) for details.

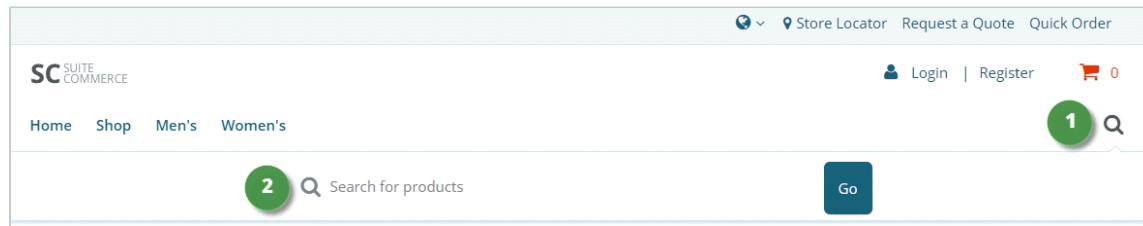
See [Theme Manifest](#) for more information on this file.

Customize Site Search Elements

Applies to: SuiteCommerce Web Stores

This topic applies to SuiteCommerce and the **2018.2** release of SuiteCommerce Advanced only.

Any themes created using the SuiteCommerce and the **2018.2** release of SuiteCommerce Advanced and later include the ability to customize the site search button and input bar. You can customize your theme to locate the site search button (magnifying glass button) and input bar in any location within any template or independently within different templates.



1. Site Search Button
2. Site Search Input Bar and Go Button

Site Search Button

The site search button is visible by default in the SuiteCommerce Base Theme. Site users must click this button to activate the site search input bar. You can locate this button in any template for desktop or mobile use with the following code:

Note: The code displayed in this topic is backward compatible with SuiteCommerce themes created prior to the 2018.2 release.

Desktop

```
<div class="header-menu-search" data-view="SiteSearch.Button">
</div>
```

- **Base Theme Module:** Header
- **Base Theme Template:** header_menu.tpl

Mobile

```
<div class="header-menu-searchmobile" data-view="SiteSearch.Button">
</div>
```

- **Base Theme Module:** Header
- **Base Theme Template:** header.tpl

Site Search Input Bar

Site users rely on this input bar to search a site. You can locate the site search input bar and **Go** button in any template for desktop or mobile use with the following code:

Site Search

```
<div class="site-search" data-type="site-search">
<div class="site-search-content">
<form class="site-search-content-form" method="GET" action="/search" data-action="search">
<div class="site-search-content-input">
<div data-view="ItemsSeacher"></div>
<i class="site-search-input-icon"></i>
<a class="site-search-input-reset" data-type="search-reset"><i class="site-search-input-reset-icon"></i></a>
</div>
<button class="site-search-button-submit" type="submit">{{translate 'Go'}}</button>
</form>
</div>
</div>
```

- **Base Theme Module:** SiteSearch
- **Base Theme Template:** site_search.tpl

The site search input bar is hidden by default in the SuiteCommerce Base Theme. However, you can customize your templates to make this input bar visible by default. To do so, add the **active** class to **site-search** as shown in the following example:

```
<div class="site-search active" data-type="site-search">
```

Go Button

```
<button class="site-search-button-link" data-action="show-itemsearcher" title="{{translate 'Search'}}">
```

```
<i class="site-search-button-icon"></i>
</button>
```

- **Base Theme Module:** SiteSearch
- **Base Theme Template:** site_search_button.tpl

Best Practices for Creating Themes

Before creating a theme or overriding extension-related files, read and understand these best practices. The information in this section provides important steps you must take to ensure that your themes and extension overrides deploy without error.

SuiteCommerce Base Theme

The SuiteCommerce Base Theme provides a basic theme developers can activate and use as a baseline for further theme development on a SuiteCommerce site. This is a managed bundle. This theme includes all best practices as described in [Best Practices for Creating Themes](#). Use this theme as a starting point for theme development to ensure that your theme is compatible with your SuiteCommerce site.

To use the SuiteCommerce Base Theme:

1. Install the SuiteCommerce Base Theme SuiteApp. See the help topic [Install Your SuiteCommerce Application](#).
2. Activate the SuiteCommerce Base Theme on a domain. See the help topic [Manage Themes and Extensions](#).
3. Use the Theme Developer Tools to fetch the active theme files for the domain. See [Fetch Active Theme Files](#).
4. Customize your theme. See [Develop Your Theme](#) and [Best Practices for Creating Themes](#).

Important Information About the SuiteCommerce Base Theme

- When you deploy a theme based on the **SuiteCommerce Base Theme**, the Theme Developer Tools prompt you to create a unique name. You cannot overwrite the Base Theme but you can create a new theme based upon it.
- The SuiteCommerce Base Theme is a managed bundle. Updates to this theme coincide with major releases of SuiteCommerce and SuiteCommerce Advanced (SCA).
- SuiteCommerce updates are all backward compatible with the SuiteCommerce Base Theme. Therefore, when your implementation of SuiteCommerce updates to the latest (managed) update, any themes created with older versions of SuiteCommerce Base Theme are compatible with your SuiteCommerce site.
- SCA customers must take care when creating themes using later releases of the SuiteCommerce Base Theme. Each managed update of the SuiteCommerce Base Theme is intended to be used with the corresponding core releases of SCA or later. Therefore, a theme created using the SuiteCommerce Base Theme 2019.1 release is compatible with sites implementing the same major release (SCA 2019.1) and later. That same theme might not be compatible on sites implementing an earlier SCA release (2018.2, for example).

General Best Practices for Themes

The following list provides some general knowledge and practices to remember when customizing themes for SuiteCommerce sites:

- Use themes to customize HTML templates and Sass for your site.
- If you are using a published theme as a baseline, the developer tools force you to create a new theme. This new theme includes your changes as a custom theme.
- Whenever possible, use the existing folder structure as created when you downloaded themes and extensions to your Workspace directory. If you must add new subdirectories, make sure the paths to any files are included in the theme manifest and any required entry points.
- Do not move, delete, or edit any files located in your Workspace/Extras directory. Any files located here are for reference and overrides only.
- When you fetch a theme, you also get the HTML and Sass files for any active extensions. You can customize these files to suit your theme using the Override method. See [Override Active Extension Files](#) for details.
- Place any new assets (images or fonts) in the appropriate location within the theme's assets directory.
- Use helpers when referencing any assets within your HTML or Sass customizations and overrides. See [Asset Best Practices](#) for details.
- Follow the template context when editing HTML template files or creating overrides. See [HTML Best Practices](#) for details.
- To avoid file name collisions, do not create any new files or folders that share the same name, even if they reside in different locations. The exception to this practice is when working with extension overrides.
- Limit the number of files and folders at the same level to 100 when developing themes and extensions. Some options include introducing images across multiple folders and placing custom modules in a different top-level directory. For more information on file and folder limiting, see [Web Services Governance Overview](#).

HTML Best Practices

SuiteCommerce templates use the Handlebars.js templating engine to keep the HTML (presentation layer) separate from any deeper code logic. These logic-less templates help you build your themes without requiring access to deeper code. Templates define the HTML for specific areas of the user interface, such as a button or Home banner.

When an associated theme is activated, the template engine combines all of these files into a single, finished web page. To achieve this, templates use an easy-to-understand syntax, which is a combination of HTML and Handlebars.js expressions (helpers). These helpers tell the templating engine to include the value of variables (properties and objects) when rendering your site or executing functions. These variables are specific to each template, depending on what the template is designed to render. This information is called the template's **context**.

Template Context

Each template relies on its context to know what variables are available (what data the template can render). You cannot customize a template to render information it cannot provide. Therefore, you must operate within this context when making any customizations to pre-existing templates or introducing your own.

SuiteCommerce templates include a list of context variables within the file itself. This lists the context objects, types, or any properties as part of an array. This information is nested within comment blocks at the end of each file.

Note: If you are an extension developer making a template for an extension, the context is defined by your own custom views. See the help topic [Extensibility API Tutorials](#) for details.

Note: SuiteCommerce Advanced users have access to the context within the associated view files.

The following example from case_list.tpl depicts this context notation.

```
...
{!----}
Use the following context variables when customizing this template:

pageHeader (String)
hasCases (Number)
isLoading (Boolean)
showPagination (Boolean)
showCurrentPage (Boolean)
showBackToAccount (Boolean)

----}}
```

The following code snippet depicts the **hasCases** number and **isLoading** boolean variables as used in the case_list.tpl template code. This code either renders a support case list or renders a **Loading** or **No Cases Found** string, depending on the query results.

```
...
{{#if hasCases}}
<table class="case-list-recordviews-table">
<thead class="case-list-content-table">
    ...
</thead>
<tbody data-view="Case.List.Items"></tbody>
</table>
{{else}}
{{#if isLoading}}
<p class="case-list-empty">{{translate 'Loading...'}}</p>
{{else}}
<p class="case-list-empty">{{translate 'No cases were found'}}</p>
{{/if}}
{{/if}}
...
```

To find the template context in a template file:

1. In an editor, open the source .tpl file for the template you intend to customize.
2. At the end of the file, look for the following string, located within comment tags:

```
{!--
```

Use the following context variables when customizing this template:

...

3. Note the context variables available to the template and customize accordingly.

To find the context variables using the browser developer tools:

At this time, some template files do not include this context as described above. In these cases, you can use the `{{log this}}` helper as described below. This procedure assumes that you have already begun customizing a template. Do not edit source files directly.



Note: The examples in this section uses Google's Chrome browser developer tools.

1. Open the template file you are customizing in an editor.
2. Include the following helper as the first line in the template for which you want to reveal the context.

```
{{log this}}
```

3. Deploy your customization to your local server. See [Test a Theme on a Local Server](#).
4. Navigate to the page rendering your custom template.
5. Using your browser's developer tools, inspect and refresh the page.
6. The developer tools **Console** tab lists the variables available to the template as the output of the `{{log this}}` helper:

```
▼ Object [ ]
  ► bottomBannerImages: Array[3]
  ► carouselImages: Array[3]
    imageHomeSize: "main"
    imageHomeSizeBottom: "main"
  ► __proto__: Object
  undefined
  ▶
```

Sass Best Practices

SuiteCommerce themes let you customize the design elements of your web store experience using Sass, or Syntactically Awesome StyleSheets. Sass is a scripting language that is transformed into CSS when you use the theme development tools to compile and deploy your customizations to the application. The SuiteCommerce Designer's Guide provides more information on customizable Sass styles. See [Design Architecture](#) for details.



Note: All SuiteCommerce Sass files are named as partials (with a preceding underscore), such as `_BaseSassStyles.scss`.

Creating Sass Variables

The SuiteCommerce Base Theme SuiteApp includes the `BaseSassStyles` module. This module contains all Sass variables and metadata (for exposing and organizing variables in SMT). The best practice when creating a theme is to use the Base Theme as a baseline. See [SuiteCommerce Base Theme](#) for details on installing this theme.

You can create your own Sass variables. You can introduce variables within new Sass files, as part of a new module, or within the BaseSassStyles module. Observe the following best practices regarding new variables and their metadata:

- If you are defining a new module as part of a theme, define all new variables and any metadata within that module.
- If you are defining a variable for a particular module only, define that variable within the associated module. Avoid introducing global definitions within one module.
- If you are defining a variable that controls multiple properties in different modules (such as a new theme color `$sc-color-theme-250`), define that variable within the BaseSassStyles module.
- If you are adding a new Sass file to your theme, you must declare each file in the theme's manifest.json and the applicable entry point file. See [Add a New File to a Theme](#) for details.
- Always test any new variables in your domain, across all applications. This includes checking to ensure that any exposed variables display as expected in the Site Management Tools Theme Customizer.

Formatting Sass for SMT

Some Sass development practices can affect the Site Management Tools Theme Customizer user experience.

As you build Sass for your theme, you can expose any variables to the SMT Theme Customizer. When an SMT administrator changes an exposed variable, the application compiles any changes and renders them for preview. This action includes two separate compilations. The first compilation occurs on the frontend within 2 seconds. However, after 5 seconds of user idle time, the application triggers a second compilation on the backend.

As a result, the following Sass practices result in variables that are only accessible in the backend. As a best practice, avoid using the following for any Sass variables exposed to SMT:

- Sass variables that receive user-defined function calls
- If conditional statements
- Mixins

User-defined function calls

Avoid user-defined function calls.

For example, consider the following example, assuming that `$sc-primary-color` is also :

```
$sc-color-secondary: myfunc($sc-primary-color) /*editable(.)
```

In this example, you set the result of an exposed variable, such as `$sc-color-secondary`, to a custom function, `myfunc($sc-primary-color)`. Assuming `$sc-primary-color` is also declared as editable, when the SMT admin changes the primary color value, the primary color value compiles quickly in the frontend. The problem with this approach is that `myfunc($sc-primary-color)` only exists in the backend and takes a longer time to preview.

If Statements and Mixins

Avoid if constructions using exposed Sass variables. For example:

```
@if $sc-color-secondary == $12345 { background-color: $sc-color-primary; }
@else { background-color: $56789; }
```

The problem with this example is that the frontend does not know the else condition. Passing values within or grouping variables as mixins causes a similar issue. All of these issues are solved during the backend compilation, but this can take a considerable amount of time.

Sass Entry Points

Every theme relies on an **entry point** file to load Sass into the correct application (Shopping, My Account, or Checkout). Each entry point declares the Sass files that are part of the application. For example, if a Sass file affects the shopping application only, it needs to load through the shopping.scss entry point. If it affects all three applications, it needs to load through all three entry points.

For an example of how to edit an entry point, see [Add a New File to a Theme](#).



Important: All Sass files must also be declared in the theme's manifest.json file. See [Theme Manifest](#) for more information.

Each theme includes the following Sass entry points:

Application Impacted	Application Sass File
Shopping	Modules/Shopping/shopping.scss
My Account	Modules/MyAccount/myaccount.scss
Checkout	Modules/Checkout/checkout.scss

Asset Best Practices

An asset is an image or font that is not managed by a NetSuite account but still part of a theme or extension. An example of an asset is a carousel image, an icon, or a new font. This can be either a pre-existing asset or one that you introduce as part of a new theme.

When you run the `gulp theme:fetch` command:

- Assets for the active theme download to your Workspace/<THEME_DIRECTORY>/assets folder. This is the location where you manage all of your assets (new and pre-existing).
- Assets for any active extensions download to your Workspace/Extras/Extensions/<EXTENSION_DIRECTORY>/assets folder. Do not move, delete, edit, or add files in this location.

When you activate your theme, all assets are placed in specific locations in your NetSuite File Cabinet based on parameters you specify when you deployed your theme (vendor name, theme name, and theme version). Later, when you activate a newer version of the same theme, your assets are now located in a different location in your File Cabinet. Your code must adapt to the change in the path. If you use absolute paths, the links to these assets will break.



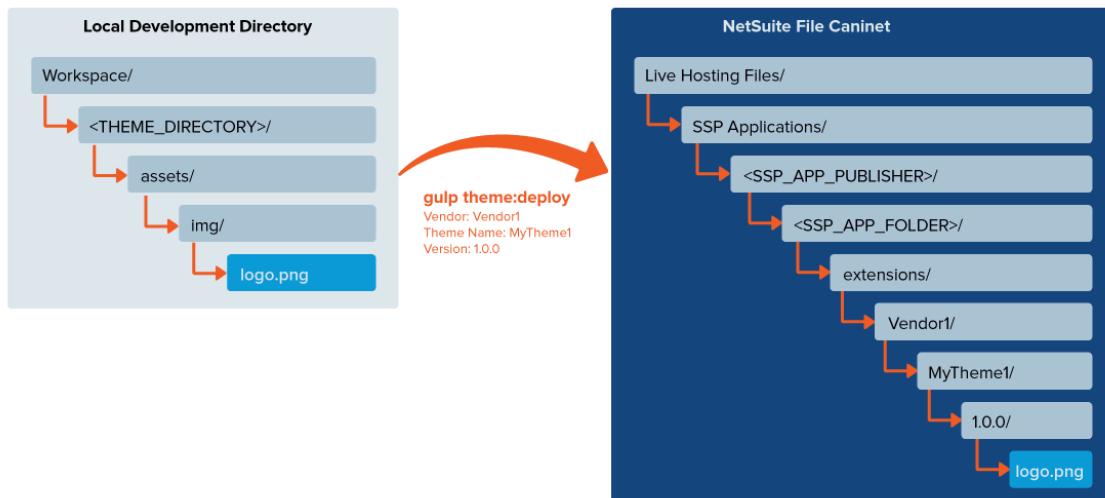
Important: Do not use absolute paths when referring to assets in your code customizations and overrides. Asset files are managed in different locations in NetSuite based on the theme's vendor, extension name, version, etc. As a best practice, always use the HTML and Sass helpers when referencing assets to ensure that you maintain dynamic paths without unnecessary code changes.

SuiteCommerce provides a few [HTML Helpers](#) and [Sass Helpers](#) to maintain dynamic paths to your assets, regardless of the vendor, theme, version, etc.

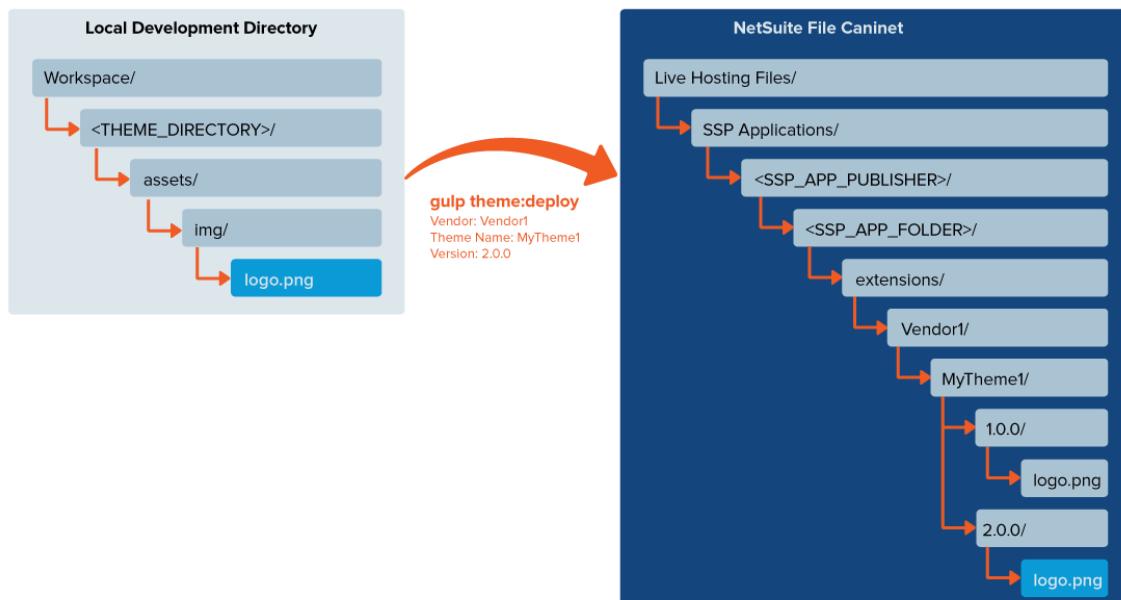
Example:

Using the theme development tools, you deploy a new theme (**MyTheme1**, Version: **1.0.0**). You then activate that theme for a specific site and domain. The compiler places all assets in your NetSuite File Cabinet. The exact location is specific to the SSP Application, vendor, extension, and version that you specified when you deployed your theme.

This location might be similar to the following:



Later, you decide to update your theme, giving it a new version, **2.0.0**. When you activate your latest version, the compiler places all assets into a new location:



HTML Helpers

The following four Handlebars.js helpers provide a means to maintain dynamic paths to your assets when customizing HTML templates.

getExtensionAssetsPath(default_value)

Use this HTML template helper:

- To access pre-existing assets included with an active extension
- In your extension overrides (templates)

 **Note:** The `default_value` argument is the relative path to the asset.

Example Syntax in a Template File:

The following is an example of how to use this helper in an HTML template:

```

```

Example Result:

This helper returns the active extension's asset path. If the first argument is defined, this helper returns the path as configured in NetSuite:

```
<MY_PATH>/img/other_logo.png
```

If the first argument is undefined, this helper uses the second argument to return the correct path of the active extension:

```
<FULL_PATH>/extensions/<VENDOR>/<EXTENSION>/<VERSION>/img/logo.png
```

getThemeAssetsPath(default_value)

Use this HTML template helper:

- To access new and pre-existing assets included in your theme directory
- In your extension overrides (templates)
- In your theme customizations (templates)

Using this helper, **default_value** is the relative path to the asset.

Example Syntax in Template File:

```

```

Example Result:

This helper returns the active theme's asset path. If the first argument is defined, this helper returns the path as configured in NetSuite:

```
<MY_PATH>/img/other_logo.png
```

If the first argument is undefined, this helper uses the second argument to return the correct path of the active theme:

```
<FULL_PATH>/extensions/<VENDOR>/<THEME_NAME>/<VERSION>/img/logo.png
```

Sass Helpers

The following two helpers provide a means to maintain dynamic paths to your assets when customizing Sass files.

getExtensionAssetsPath(\$asset)

Use this Sass helper:

- To access pre-existing assets included with an active extension
- In your extension overrides (Sass)

Example Syntax in a Sass File:

```
body {
    background-image: url(getExtensionAssetsPath('img/background-image.png'));
}
```

Example Result:

This helper returns the active extension's asset path.

```
<FULL_PATH>/extensions/<VENDOR>/<EXTENSION>/<VERSION>/img/logo.png
```

getThemeAssetsPath(\$asset)

Use this Sass helper:

- To access new and pre-existing assets included in your theme directory
- In your extension overrides (Sass)
- In your theme customizations (Sass)

Example Syntax in a Sass File:

```
body {
    background-image: url(getThemeAssetsPath('font-awesome'));
}
```

Example Result:

This helper returns the active theme's asset path.

```
<FULL_PATH>/extensions/<VENDOR>/<THEME_NAME>/<VERSION>/font-awesome
```

Design Architecture

The SuiteCommerce design architecture provides an intuitive, robust, and flexible solution that lets you customize the design elements of your web store experience with ease. SuiteCommerce Sites use the

Sass CSS extension language. With Sass, CSS syntax is fully supported in addition to features such as variables and imports.

Sass source code also uses KSS notation to assist in building a Style Guide. This requires the Kilimanjaro release of SuiteCommerce Advanced or later.

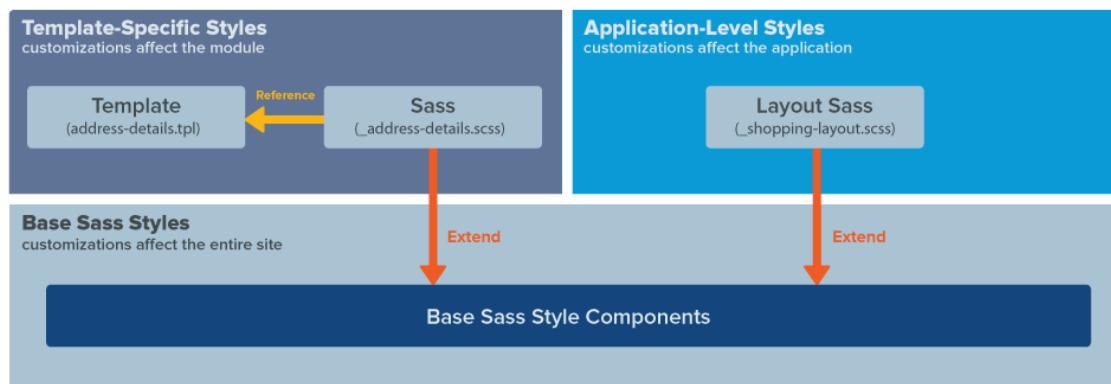


Important: This section provides details on the design architecture. For a complete catalog of all design components and the user experience workflows see the Design section of the SuiteCommerce Developer's Portal at developer.suitecommerce.com.

Design Hierarchy

Sass styles are organized in a hierarchy as follows:

- **Base Sass Styles:** These are the basic components designed to be combined and reused across the site.
- **Template Specific Styles:** Sass styles specific to a page or template are defined here. These are extensions and combinations of the base components.
- **Application Level Styles:** Within the Base Sass Styles, you can define components that are referenced from a specific application: ShopFlow, My Account, or Checkout. This lets you define unique styling for each application experience.



Base Sass Styles

In SuiteCommerce Advanced core styles are defined as reusable components. Core reusable components can then be iteratively combined to create increasingly complex structures such as templates and pages. All basic components are defined within the **BaseSassStyles** folder at Modules > suitecommerce > BaseSassStyles. Components are categorized into three different types:

- **atoms:** basic building blocks to be used within other structures. For example, buttons, messages, and forms.
- **molecules:** simple combinations of atoms. Molecules are built for reuse but serve different purposes depending on the page context.
- **variables:** basic styling rules such as spacing and typography, that are then referenced and extended in atoms and molecules.

Template Specific Styles

Each module contains a Sass folder that defines styling rules specific to the templates used for that module. Within these Sass files, the Base Sass Style components are used as the starting point and then

extended as needed. There is generally a single Sass file corresponding to each template file within a module.

For example, the Address module has a **address-details.tpl** file in the Templates folder of the module with a corresponding **_address-details.scss** Sass file in the Sass folder of the module. Within the **_address-details.tpl** file the **address-details-remove-address** class is referenced. This class is defined in the **_address-details.scss** file where base classes are extended to add styles specific for this template.

Class Referenced in Template File:

```
{#{if showRemoveButton}
  <button class="address-details-remove-address" data-action="remove" data-id="{{internalid}}" {{#if
  isInvalidAddressToRemove}}disabled{{/if}}>
    {{translate 'Remove'}}
  </button>
{#{/if}}
```

Class Defined in Sass File:

```
.address-details-remove-address {
  @extend .button-edit;
  margin-right: $sc-base-margin-lvl2;
  cursor: pointer;
}
```

Application Level Styles

In addition to defining styles specific to a module using template-specific classes, you can define styles to be specific to each application. The ShopFlow, My Account, and Checkout experiences can all have a unique design.

By default, for each application module, there is a single **layout** Sass file with several application specific style extensions defined.

- **_shopping-layout.scss**
- **_myaccount-layout.scss**
- **_myaccount-layout-sb.scss**
- **_checkout-layout.scss**

Also, you can define application specific dependencies for base styles or customizations in the theme manifest file. Each file must also be added as a dependency to the appropriate application in the **sass** object before styles defined in those files can be included when you deploy. See [Theme Manifest](#) for details.

Style Definitions

i Applies to: Kilimanjaro

To help you maintain and develop Sass variables, SuiteCommerce Advanced uses the following Sass style definitions. These provide flexible, comprehensible, and easy-to-maintain variables and their values.

This section explains how SuiteCommerce Advanced uses the following style definitions:

- Colors
- Typography
- Spacing



Note: To ensure inheritance and to define variables in a readable way, SuiteCommerce Advanced uses the KSS notation to define them. See [Style Guide](#) for details.

Colors

Colors represent a large number of variables throughout SuiteCommerce Advanced. These are represented as primary colors, secondary colors, themes, messaging, links, and neutral shades.

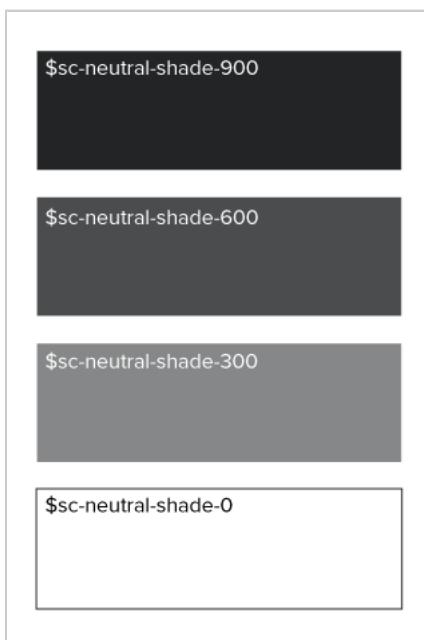
Neutral shades and theme colors are handled with a slight difference to other color definitions. Both neutral shades and theme color palettes use a shading scale from 1000 (dark) to 0 (light).

For example, the BaseSassStyle module defines neutral shades in the following variables:

```
//...
$sc-neutral-shade-base: #222426;
$sc-neutral-shade-900: $sc-neutral-shade-base;
$sc-neutral-shade-600: lighten($sc-neutral-shade-base, 18);
$sc-neutral-shade-300: lighten($sc-neutral-shade-base, 55);
$sc-neutral-shade-0: lighten($sc-neutral-shade-base, 100);

//...
```

In this example, **sc-neutral-shade-base** defines the base neutral value **#222426**, which is a very dark gray. SuiteCommerce Advanced declares other neutral shades as calculated percentages of the base. Therefore, **sc-neutral-shade-900** has no calculation and matches the base shade. However, **sc-neutral-shade-600** renders as the base shade lightened by 18%. As we progress through each neutral shade, shades becomes lighter. You can extend the class definitions in the `_colors.scss` file to introduce intermediate shades in this manner.



The theme color palette handles colors in a similar way. Instead of defining a hexadecimal value, the theme palette defines a previously declared color value. For example, `$sc-color-secondary`.

Typography

Like color variables, SuiteCommerce Advanced declares typography variables in an intuitive, human-readable manner. This relies on the familiar sizing method of extra-small (xs) through triple-extra-large (xxxl). Each of these sizes is relative to a base font size (`$sc-font-size-base`). SuiteCommerce Advanced expresses the base font size in pixels and all related font sizes in rem units (relative em). `1rem` equals the font size of the base element. Therefore, `1.067rem` is a calculation meaning the value for `$sc-font-size-base` times 1.067.

In the following example from `_typography.scss`, `$sc-font-size-m` is equal to 15 pixels ($1 \times 15\text{px}$), and `$sc-font-size-l` is equal to about 16 pixels ($1.067 \times 15\text{px}$). Likewise, `$sc-font-size-xxxl` is equal to about 26 pixels ($1.73 \times 15\text{px}$).

```
// ...

$sc-font-size-base: 15px;

$sc-font-size-xxs: 0.75rem;
$sc-font-size-xs: 0.87rem;
$sc-font-size-s: 0.93rem;
$sc-font-size-m: 1rem;
$sc-font-size-l: 1.067rem;
$sc-font-size-xl: 1.2rem;
$sc-font-size-xxl: 1.47rem;
$sc-font-size-xxxl: 1.73rem;

// ...
```

Spacing

Spacing variables affect the structural height, width, and spacing of elements. These are most commonly referred to as the padding and margins. Like color and typographic variables, spacing variables are declared in an intuitive, human-readable manner. SuiteCommerce Advanced uses **levels** to indicate the added or reduced space for a class relative to a base style. Each level designation in the named variable equals the multiplier. That is, `1v2` bares a multiplier of 2.

In the following example from `_spacing.scss`, `$sc-padding-base` is equal to 5 pixels, and `$sc-padding-1v2` is equal to 10 pixels ($2 \times 5\text{px}$). Likewise, `$sc-padding-1v8` equals 40 pixels ($8 \times 5\text{px}$).

```
// ...

$sc-padding-base: 5px;
$sc-padding-lv1: $sc-padding-base;
$sc-padding-lv2: $sc-padding-base * 2;
$sc-padding-lv3: $sc-padding-base * 3;
$sc-padding-lv4: $sc-padding-base * 4;
$sc-padding-lv5: $sc-padding-base * 5;
$sc-padding-lv6: $sc-padding-base * 6;
$sc-padding-lv7: $sc-padding-base * 7;
$sc-padding-lv8: $sc-padding-base * 8;

// ...
```

Style Guide

i Applies to: Kilimanjaro

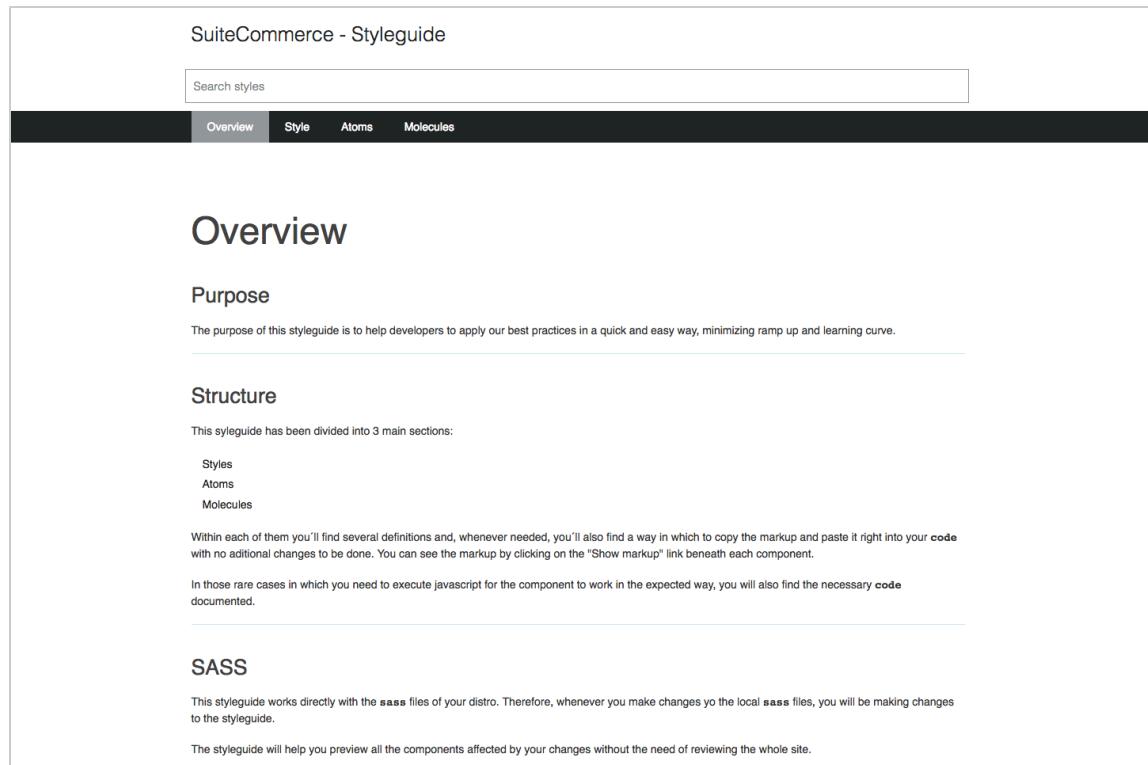
A style guide helps developers and designers use the various style elements defined for a site. Contributors can refer to a style guide to create new pages and elements or customize existing ones. A style guide can also ensure that your best practices for site design are being met by multiple contributors.

You can use the developer tools to create a style guide automatically. To do so, the source `BaseSassStyles` module uses KSS notation to document and define Sass elements. As a best practice, build any Sass customizations using KSS notation as defined in this section.

To create a style guide using the developer tools:

1. In your local developer environment, open a command line or terminal and access the top-level development directory.
2. Run the following command:
`gulp theme:local styleguide`
3. Point your browser to:
`localhost:3000`

You should see your style guide appear, looking similar to the following example:



The screenshot shows the 'SuiteCommerce - Styleguide' interface. At the top, there's a search bar labeled 'Search styles'. Below it is a navigation bar with tabs: 'Overview' (which is active), 'Style', 'Atoms', and 'Molecules'. The main content area is titled 'Overview' and contains sections for 'Purpose' and 'Structure'. Under 'Structure', it says 'This styleguide has been divided into 3 main sections: Styles, Atoms, Molecules'. It also notes that within each section, you can copy and paste code. The 'SASS' section at the bottom states that the styleguide works directly with local Sass files and provides a preview of affected components.

KSS Notation

SuiteCommerce uses Knyle Style Sheets (KSS) notation to automate the creation of a Sass style guide. KSS is simply a documentation syntax for CSS, written in a human-readable way. A style guide serves as a place to publish KSS documentation and visualize different states of user interface elements defined in your Sass.



Note: KSS does not affect the performance or rendering of CSS in any way. It is meant to document your styles in a human-readable way and automate the creation of a related style guide.

The following elements of this notation appear before each element defined in a Sass file. These are parsed and used to create a style guide automatically using the developer tools.

- **Title** – defines the element. Best practice is to use an intuitive, human-readable title.
- **Description** – briefly explains the intended purpose and implementation of the element.
- **Markup** – provides the HTML markup used to generate the element.
- **Index** – defines where the element exists within the style guide structure.

For example:

```
2.0
2.1
  2.1.1
  2.1.2
2.2
  2.2.1
```

The following example comes from the base Sass style atom `_buttons.scss` and highlights the basics of KSS notation as used by SuiteCommerce Advanced.

```
///...

// Primary buttons
(This is an example of the Title element)
//
// The primary action is generally the action that completes a task or desired action.
// Primary buttons use bright and use saturated colors to provide stronger visual weight that
// catches the users attention to avoid confusion and competing with secondary actions.
(This is an example of the Description element)
//
// Markup:
// <button class='button-primary button-medium'>Button Primary</button>
(This is an example of the Markup element)
//
// Styleguide 2.1.1
(This is an example of the Index element)

.button-primary {
  color: white;
  font-weight: $sc-font-weight-normal;
  background: $sc-color-button-primary;
  display: inline-block;
  border: 1px solid $sc-color-button-primary;
  border-radius: $sc-border-radius;

//...
```

Mobile First

SCA is developed with a Mobile First approach. Mobile First design results in faster, sleeker sites while enabling for easier design. Developing sites with the Mobile First approach promotes using only the most

important elements on your screen while avoiding cluttering the screen with secondary elements not critical to the current task. This enhances the user experience by allowing users to quickly find and finish the task at hand whether it be finding a product to purchase in a B2C environment or reordering in bulk in a B2B environment.

Using the Mobile First approach, element styles are defined for mobile devices and then scaled progressively for tablet and desktop display.

For example, in the following code sample the `.button-medium` style is defined with a width of 100%. Only when the media is of `$screen-sm-min` size (tablet) is the width adjusted. The `default` media sizes are all defined in Sass variable files.

```
.button-medium{
    padding:$sc-base-padding * 2.5 $sc-base-padding * 4;
    letter-spacing:0.5px;
    font-size: $sc-button-medium-font-size;
    width:100%;
    text-align: center;
    line-height:1;

    @media(min-width: $screen-sm-min){
        width: auto;
    }
}
```

Theme Manifest

Your theme's workspace includes a `manifest.json` file for the theme. This file is located in each theme directory within the Workspace directory. The theme's manifest is a JSON file that includes all the information required to compile resources for an active theme. This topic explains this file.

Workspace/<THEME_DIRECTORY>/manifest.json

This file lists all HTML templates, Sass, and assets related to the active theme that you downloaded when you ran the `gulp theme:fetch` command. You only need to edit this file to introduce any new HTML, Sass, or assets you create as part of your theme customizations. You can edit Skin labels as well at your option.

 **Note:** See [Add a New File to a Theme](#) for instructions on how to edit this file when you add any new HTML, Sass, or assets.

This file also manages any extension overrides. However, the theme development tools add the necessary overrides to this manifest when you deploy your customizations. This section describes the following sections of the theme manifest:

- [Theme Metadata](#)
- [Overrides](#)
- [Templates](#)
- [Sass](#)
- [Skins](#)
- [Assets](#)
- [Record Paths](#)

Theme Metadata

The first entries in the manifest file include metadata about the theme or extension itself. These fields are automatically populated when you initially run the `gulp theme:deploy` command.

```
{
  "name": "StandardTheme",
  "vendor": "SuiteCommerce",
  "type": "theme",
  "target": "SuiteCommerce",
  "version": "1.2.0",
  "description": "Standard theme for SuiteCommerce",
  //...
}
```

- Name (string) – uniquely identifies the name of the theme. This field is required.
- Vendor (string) – identifies the vendor as a string. This field is required.
- Type (string) – indicates if the type as a **theme**. This field is required.
- Target (comma-separated string) – indicates the SuiteCommerce Applications supported by the theme. This field is required.
- Version (string) – indicates the version of the theme, such as 1.0.0. This field is required.
- Description (string) – provides a description of the theme as it appears in NetSuite. This field is optional.

Overrides

The **override** object is only included if you introduce extension overrides. When you use the `Override` method, the Gulp.js commands detect any HTML or Sass overrides and include them in this file automatically.

For example, if you override the `_error.sass` file of the **Extension1** extension and run the `gulp theme:deploy` command, the theme development tools add the following override your theme's `manifest.json` file as part of the deployment process:

```
//...
"override": [
  {
    "src": "Overrides/Extension1/Modules/MyExamplePDPExtension1/Sass/_error.scss",
    "dst": "Extension1/Modules/MyExamplePDPExtension1/Sass/_error.scss"
  },
  ...
]
```

Templates

The **templates** object lists all HTML template files included in the theme by application. The **application** object includes one object per application (**shopping**, **myaccount**, and **checkout**). Each application lists each file in the **files** array.

You manually add any new template files to this array that you introduced as part of your theme customizations.

```
//...
"templates": {
```

```

"application": {
    "shopping": {
        "files": [
            "Modules/AddToCartModifier/Templates/add_to_cart_modifier.tpl"
            // ...
        ]
    }
    "myaccount": {
        "files": [
            // ...
        ]
    }
    "checkout": {
        "files": [
            // ...
        ]
    }
}
//...

```

Sass

The `sass` object declares the paths to each application entry point and all Sass files to be loaded when you deploy. You manually add any new Sass files to the `files` array that you introduced as part of your theme customizations.



Note: When listing a new Sass file, declare each file in an order that makes the most semantic sense within the Sass hierarchy.

```

//...
"sass": {
    "entry_points": {
        "shopping": "Modules/Shopping/shopping.scss",
        "myaccount": "Modules/MyAccount/myaccount.scss",
        "checkout": "Modules/Checkout/checkout.scss"
    }
    "files": [
        "Modules/Shopping/shopping.scss",
        "Modules/MyAccount/myaccount.scss",
        "Modules/Checkout/checkout.scss",
        "Modules/twitter-bootstrap-sass@3.3.1/assets/stylesheets/bootstrap/_alerts.scss",
        // ...
    ]
}
//...

```

Skins

The `skins` array is automatically populated when you run either the `gulp theme:local` or `gulp theme:deploy` commands. This array defines an object for each new skin preset file located in the Skins directory when you run these commands. Each skin object includes the following properties:

- **name** – declares the name of the skin as it appears in SMT. As a default, this value equals the file name.
- **file** – declares the name and location of the skin preset file. This name must match the name of the file in the Skins directory.

```
//...
,  'skins': [
  {
    'name': 'winter_skin'
    , 'file': 'Skins/winter_skin.json'
  }
, {
    'name': 'spring_skin'
    , 'file': 'Skins/spring_skin.json'
  }
, {
    'name': 'summer_skin'
    , 'file': 'Skins/summer_skin.json'
  }
, {
    'name': 'fall_skin'
    , 'file': 'Skins/fall_skin.json'
  }
]
//...
```

By default, If necessary, you can edit the manifest.json file to provide a more visually appealing value for the **name** property.

Example 2. Example

You want to provide four selectable skins for a theme, each with a different color scheme to correspond with the different seasons: winter, spring, summer, and fall. You create four individual skin preset files in your theme's Skins directory: winter_skin.json, spring_skin.json, summer_skin.json, and fall_skin.json.

You use the theme developer tools to run the `theme gulp:local` command. The developer tools automatically edit your theme's manifest.json file:

Later, you decide that you want the SMT Theme Customizer to display different names for each skin. You open the manifest.json file and make the following edits to each skin object's **name** property:

Assets

The **assets** object defines paths to the images and fonts located in the theme directory's assets folder. This defines an array for each image and font used. These paths are relative to the theme's assets folder path. This is where you add any new asset files introduced as part of your theme customizations.

```
//...
"assets": {
  "img": {
    "files": [
      "img/favicon.ico",
      "img/image-not-available.png",
      "img/add-to-cart-logo.png",
    ]
  }
}
```

```
},
"font-awesome": {
  "files": [
    "font-awesome/FontAwesome.otf",
    "font-awesome/custom/fontawesome-webfont.eot",
    // ...
  ],
  "fonts": {
    "files": [
      "fonts/DancingScript-Bold.ttf",
      "fonts/DancingScript-Regular.ttf",
      // ...
    ]
  }
}
//...
```

Record Paths

The final part of the manifest file lists the path to the theme, the Extension record and Activation IDs as stored in NetSuite.

```
//...
"path": "SuiteBundles/Bundle 193239/SuiteCommerce Base Theme",
"extension_id": "4535",
"activation_id": "59"
```

Extensions

 **Applies to:** SuiteCommerce Web Stores

This topic introduces you to extension development through the following topics:

- [Develop Your Extension](#) – This topic provides a general checklist to follow when creating an extension. This process includes building baseline files and calling the Extensibility API to customize the application and add new features or functions.
- [Create Page Types](#) – This topic explains how to create a new page type within your extension for use with Site Management Tools (SMT).
- [Extension Manifest](#) – This topic describes the manifest.json file, an auto-generated file that manages all compilation information for your extension. This topic also explains how to manually edit this file.
- [Troubleshoot Activation Errors](#) – This topic explains how to use the Extension Manager in NetSuite to investigate errors with your activations.

You build extensions to interact with the Extensibility API to accomplish some task. Instead of altering frontend or backend code, you create extensions using **SuiteCommerce Components** to interact with the API. The API then makes calls to deeper structures of the code base. See the help topic [Extensibility API](#) for more details.



Important: To develop an extension, you must have experience working with JavaScript, HTML, and Sass/CSS. The level of experience required depends on the types of changes you want to make. Advanced JavaScript programming skills, including knowledge of APIs, Backbone.js and jQuery are required.

Benefits of Using Extensions

Extensions introduce added functionality to a SuiteCommerce website through any number of JavaScript, SuiteScript, configuration JSON, and other files bundled into a single SuiteApp or deployed to a NetSuite account for later activation using the Manage Extensions wizard (included with the SuiteCommerce Extensions Management SuiteApp). One important benefit of extensions is that they allow non-technical users to extend and update their site by installing and activating any number of pre-developed features from a marketplace.

Extensions provide many other benefits:

- Partners can publish and distribute extensions as bundled SuiteApps.
- In-house site developers working with SuiteCommerce or SuiteCommerce Advanced (SCA) can create and manage their own extensions and activate them for any domains associated with a site.
- Extension developers can access much of the functionality that the NetSuite platform currently provides (SuiteScript, custom records/fields/forms, etc.).
- Extensions eliminate version lock within SCA. This means that users have access to easier upgrades that do not compromise the functionality of features previously activated or enabled for a site.
- Extensions rely on accessing the Extensibility API, which ensures NetSuite developers maintain backwards compatibility with SuiteCommerce code or with previous releases of SCA.

Develop Your Extension

Extensions require interacting with the Extensibility API. The Extensibility API is available to extension developers for both SuiteCommerce and SuiteCommerce Advanced (SCA). Be aware of the following requirements:

- **SuiteCommerce** – The Extensibility API is required to create extensions for SuiteCommerce.
- **SuiteCommerce Advanced (Aconcagua and later)** – Creating extensions using the Extensibility API is a best practice for developing Suite Commerce Advanced. However, if your SCA customizations require access to JavaScript, SuiteScript, or configuration objects that are not accessible using this API, you must use the core SCA developer tools and customization practices. See [Core SCA Source Code](#) for details.
- **SuiteCommerce Advanced (Kilimanjaro and earlier)** – The Extensibility API is not available for these implementations of SCA. To take advantage of themes, extensions, and the Extensibility API, you must either migrate your site to the latest release of SCA or use the core SCA developer tools and customization practices to customize your site. See [Core SCA Source Code](#) for details.

Extension Development Steps

Follow this general guide to develop your extension:

Step	Description	More Information
1	Create a set of baseline extension files. The extension developer tools give you a starting point for extension development. Follow these instructions to create a baseline Hello World extension.	Create Extension Files
2	Customize your extension. You can introduce JavaScript, SuiteScript, Json, HTML templates, and Sass files, as needed. The Extensibility API is available to access frontend and backend modules within the application.	Extensibility API
3	Add any assets to your extension source files, such as fonts and images. This step requires manually updating your extension's manifest.json file.	Extension Manifest
4	Optionally build your extension to integrate with Site Management Tools (SMT). You can create your extension as a Custom Content Type (CCT) or create new CMS Page Types and designate templates as selectable layouts for new and existing pages in SMT.	Create a Custom Content Type Create Page Types
5	Deploy your extension. You can test your extension on a local server. However, you must first deploy your extension files to your account and activate the extension on a domain before testing locally.	Deploy an Extension to NetSuite
6	Activate your extension. To see your extension in action, you must activate it for a domain.	Manage Themes and Extensions
7	After activating your extension for a domain in NetSuite, you can test your extension code on a local server for debugging purposes.	Test an Extension on a Local Server
8	When you are satisfied with your extension code, deploy using the extension developer tools.	Deploy an Extension to NetSuite

Step	Description	More Information
	 Important: You must activate the extension before any deployed changes take effect. See the help topic Manage Themes and Extensions for details.	
9	Create a bundle for your extension. If you intend to publish your extension and share it with other accounts, follow these instructions to bundle your extension as a SuiteApp.	Theme and Extension SuiteApps

Create Page Types

 **Applies to:** SuiteCommerce Web Stores

Page types let Site Management Tools (SMT) administrators introduce a variety of pages and layouts to a website. When SMT admins create new pages in SMT, they define it as a specific page type. This can be either the default Landing Page or any newly created page type introduced through an extension. SMT admins can also change the layout of existing pages. When SMT admins edit an existing core page, they create an enhanced page and can select from a list of different layouts, based on the layout options defined by the extension.

However, for these elements to be available in SMT, extension developers must first set up an extension to register page types and templates using the Extensibility API.

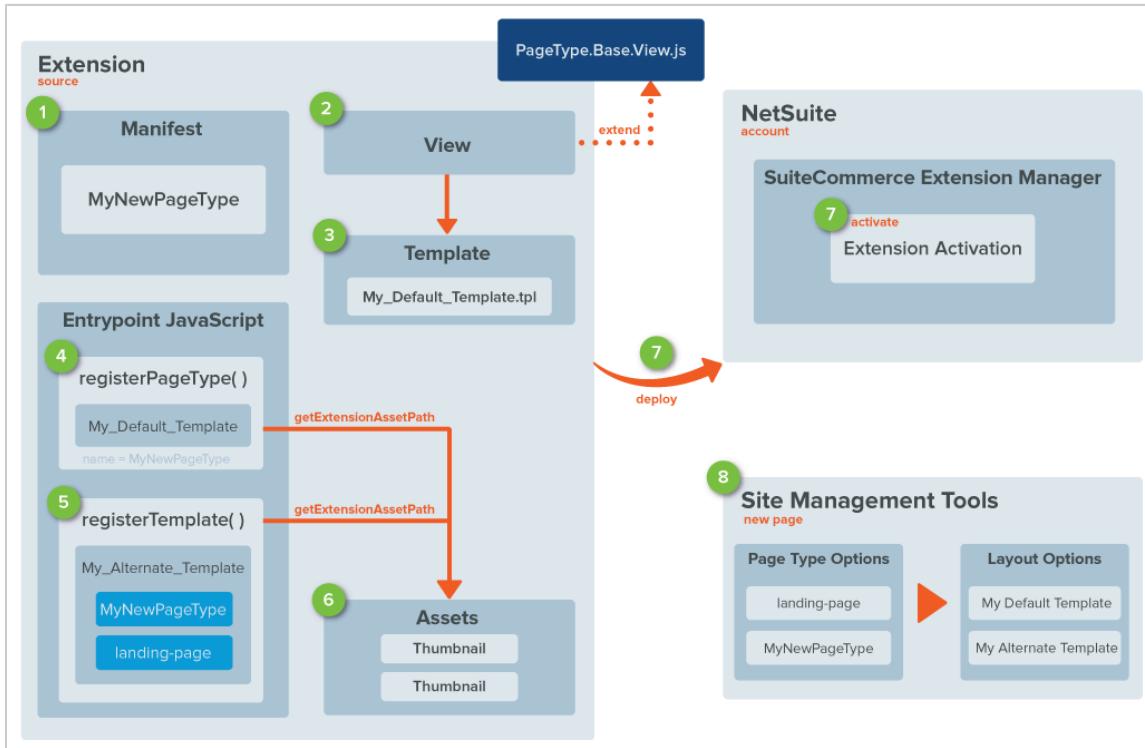
 **Note:** Page types are associated with CMS Page Type records. For more information on these records in NetSuite, see [CMS Page Type Record](#). For more information on creating new pages using SMT, see the help topic [Page Layout Selector](#).

The following steps explain how to set up a SuiteCommerce extension to:

- Create a new CMS Page Type record in NetSuite
- Register new page types to be available when creating new pages for a SuiteCommerce site using SMT
- Create templates and define them as either a default layout for a page type or as alternative layout options



Note: These steps assume that you have already started building an extension using the extension developer tools. See [Extension Development Steps](#) for a checklist of steps required to build an extension.



Step	Description	More Information
1	<p>Set up your extension to create a new CMS Page Type record in NetSuite. This requires manually editing your extension's manifest.json file.</p> <p>During testing and development, you need to create this CMS Page Type record manually.</p> <p>Note: Your extension automatically creates the CMS Page Type record when it is installed into an account from a bundle. During early testing, however, your extension is not yet part of a bundle, so you must manually create it.</p>	Add a CMS Page Type Record
2	Create a view for your new page type. This JavaScript file is essential to render the instance of the page type. This file must extend PageType.Base.View.js.	Create a View to Render a Registered Page Type
3	Create a template to act as the default layout for your registered page type.	SMT Templates and Areas
4	Register your new page type using the Extensibility API.	Register a Page Type
5	Register alternative layouts using the Extensibility API. Every page type includes a default layout. However, you can register other templates to be available as alternative layout options for a new or existing page types.	Register a Template

Step	Description	More Information
6	Create template thumbnails as assets. Each template file corresponds to a different layout option, selectable in SMT when creating a new page. Each template includes a thumbnail preview image that you include with the extension.	Create Layout Thumbnails
7	Deploy your changes. You can test your extension on a local server. However, you must first deploy your extension files to your account and activate the extension on a domain to test the functionality in SMT. <div style="border: 1px solid #f0e68c; padding: 5px; margin-top: 10px;">  Important: You must activate the extension before any deployed changes take effect. See the help topic Manage Themes and Extensions for details. </div>	Deploy an Extension to NetSuite.
8	Log into SMT and use the Layout Selector when creating a new page or editing an enhanced page.	Page Layout Selector

Add a CMS Page Type Record

You set up your extension's manifest.json file to automatically create a new CSM Page Type record. When a user installs your extension from a bundle, NetSuite automatically creates the record based on the parameters you set here.

To set up your extension to create a CMS Page Type record:

1. Open your extension's manifest.json file.
This is located in your Workspace/[extension] folder.
2. Add the `page` object and `types` array.
Create a new object in the `types` array for each new page type you want your extension to create. See [Page](#) in the [Extension Manifest](#) topic for details.
3. Save your manifest.json file.

Page Types in a Development Environment

During early testing and development, no bundle exists for your extension. You can test an extension by deploying it to an account. However, since the bundle installation is what triggers the CMS Page Type record creation, you must manually create the record in NetSuite to test your extension. This is not the typical user experience, but it is necessary to build the mechanism during development.

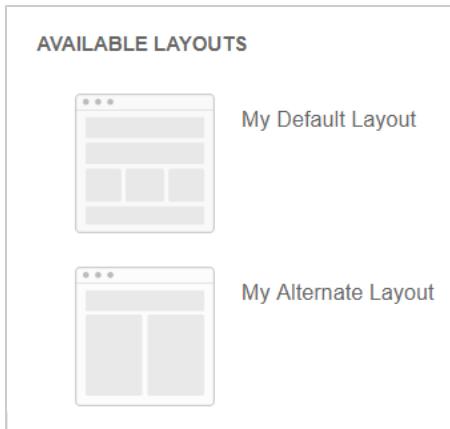
Observe these minimum requirements for testing a CMS Page Type record with your extension:

- The **Name** and **Display Name** fields are mandatory.
- The **Name** field must match the `name` parameter in the `registerPageType()` method in your extension's entrypoint JavaScript file. No spaces or special characters are allowed in this field. See the help topic [Register a Page Type](#) for details.
- To appear in SMT, the **CMS Creatable** field must be set to **Yes**.

To create a CMS Page Type record manually, see [CMS Page Type Record](#).

Create Layout Thumbnails

Part of setting up a layout in your extension includes specifying a layout thumbnail. This image acts as a small preview or representation of the associated layout. Here is an example of thumbnails in use in Site Management Tools (SMT):



You can download a collection of stock images to use for your thumbnails or as a starting point when developing your own: [Stock SMT Layout Thumbnails](#).

If you create your own thumbnails, follow these best practices when creating thumbnails:

- When possible, use the stock thumbnails provided as a starting point.
- Format thumbnail images as SVG files. Other image types are supported, but SVGs provide more versatility and scalability with typically smaller file sizes.
- Limit file sizes to NetSuite file size limitations. See the help topic [Best Practice for Preparing Files for Upload to the File Cabinet](#) for details.
- Create thumbnail images with either of the following pixel dimensions:
 - 84 x 84 px
 - 180 x 180 px
 - 245 x 245 px
- Introduce thumbnail images within your extension as assets. This requires editing your extension's manifest.json file and using HTML helpers to register them. See the following help topics for details:
 - [Asset Best Practices](#)
 - [Extension Manifest](#)
- Include a reference to these thumbnails when you register your page types or layouts. See the help topics [Register Page Types and Templates for SMT](#) and [HTML Helpers](#) for help.

Extension Manifest

Your extension's Workspace directory includes a manifest.json file, which includes all the information required to compile resources for the extension.

`../Workspace/<EXTENSION_DIRECTORY>/manifest.json`

This file is auto-generated when you execute the `gulp extension:create` command and includes all JavaScript, JSON, SuiteScript, HTML templates, Sass, and assets required to compile and activate your extension.

Manual Edits

This file lists all JavaScript, JSON, SuiteScript, HTML templates, Sass, and assets related to your extension. Although this file is automatically generated, you may need to update it manually if adding Sass entry points for any newly introduced Sass files or changing CCT labels.

 **Warning:** Potential data loss. Besides compiling and deploying your extension to a local server, the `gulp extension:local` and `gulp extension:deploy` commands update the extension's `manifest.json` file and overwrites any manual changes you made to this file.

To preserve manual changes to `manifest.json` use the following commands to test your extension locally or deploy to NetSuite:

- `gulp extension:local --preserve-manifest`
- `gulp extension:deploy --preserve-manifest`

Extension Metadata

The first entries in the manifest file include metadata about the extension itself. These fields are automatically populated when you initially run the `gulp extension:create` command.

```
{
  "fantasyName": "My Cool Extension!",
  "name": "MyCoolExtension",
  "vendor": "Acme",
  "type": "extension",
  "target": "SuiteCommerce",
  "version": "1.0.0",
  "description": "My cool extension does magic!",
  // ...
}
```

- **fantasyName** (string) – identifies the label for the extension. This can include special characters.
- **name** (string) – uniquely identifies the name of the extension. This field is required. This must be alphanumeric characters without spaces.
- **vendor** (string) – identifies the vendor as a string. This field is required.
- **type** (string) – indicates if the type as an **extension**. This field is required.
- **target** (comma-separated string) – indicates the SuiteCommerce Applications supported by the theme. This field is required.
- **version** (string) – indicates the version of the theme, such as 1.0.0. This field is required.
- **description** (string) – provides a description of the extension as it appears in NetSuite. This field is optional.



Important: As a best practice, use semantic versioning (SemVer) notation. For more information, see <https://semver.org/>.

Assets

The **assets** object defines paths to the images and fonts located in the theme directory's assets folder. This defines an array for each image and font used. These paths are relative to the theme's assets folder path. Extensions treat services as assets. These are created on NetSuite servers when you activate/deploy your extension.

If the extension developer tools detect file name with the pattern **XYZ.ServiceController.js**, they create the service (.ss) file and add it to the manifest. Later, when you activate the extension, the declared .ss moves to the backend as an asset.

```
//...
"assets": {
  "img": {
    "files": []
  },
  "fonts": {
    "files": []
  },
  "services": {
    "files": [
      "services/MyCoolModule.Service.ss",
      "services/AdditionalCoolModule.Service.ss"
    ]
  }
},
//...
```

Configuration

The **configuration** object defines paths to the JSON files in your extension directory's Configuration folder.

```
//...
"configuration": {
  "files": [
    "Modules/MyCoolModule/Configuration/MyCoolModule.json",
    "Modules/AdditionalCoolModule/Configuration/AdditionalCoolModule.json"
  ]
},
//...
```

Templates

The **templates** object lists all HTML template files included in the extension by application. The **application** object includes one object per application (**shopping**, **myaccount**, and **checkout**). Each application lists each file in the **files** array.

```
//...
"templates": {
  "application": {
    "shopping": {
      "files": [
        "Modules/MyCoolModule/Templates/acme_mycoolextension_mycoolmodule_list.tpl",
        "Modules/MyCoolModule/Templates/acme_mycoolextension_mycoolmodule_edit.tpl",
        "Modules/AdditionalCoolModule/Templates/acme_mycoolextension_additionalcoolmodule_list.tpl",
        "Modules/AdditionalCoolModule/Templates/acme_mycoolextension_additionalcoolmodule_edit.tpl"
      ]
    },
    "myaccount": {
      "files": [
        "Modules/MyCoolModule/Templates/acme_mycoolextension_mycoolmodule_list.tpl",
        "Modules/MyCoolModule/Templates/acme_mycoolextension_mycoolmodule_edit.tpl",
        "Modules/AdditionalCoolModule/Templates/acme_mycoolextension_additionalcoolmodule_list.tpl",
        "Modules/AdditionalCoolModule/Templates/acme_mycoolextension_additionalcoolmodule_edit.tpl"
      ]
    }
  }
},
```

```

    "files": [
        "Modules/MyCoolModule/Templates/acme_mycoolextension_mycoolmodule_list.tpl",
        "Modules/MyCoolModule/Templates/acme_mycoolextension_mycoolmodule_edit.tpl",
        "Modules/AdditionalCoolModule/Templates/acme_mycoolextension_additionalcoolmodule_list.tpl",
        "Modules/AdditionalCoolModule/Templates/acme_mycoolextension_additionalcoolmodule_edit.tpl"
    ],
},
"checkout": {
    "files": [
        "Modules/MyCoolModule/Templates/acme_mycoolextension_mycoolmodule_list.tpl",
        "Modules/MyCoolModule/Templates/acme_mycoolextension_mycoolmodule_edit.tpl",
        "Modules/AdditionalCoolModule/Templates/acme_mycoolextension_additionalcoolmodule_list.tpl",
        "Modules/AdditionalCoolModule/Templates/acme_mycoolextension_additionalcoolmodule_edit.tpl"
    ]
}
},
//...

```

Sass

The `sass` object declares the paths to each application entry point and all Sass files to be loaded when you deploy. If you introduce a new Sass file into your extension workspace, you must manually add it to the `files` array within the `sass` object.

 **Note:** If manually listing Sass files, declare each file in an order that makes the most semantic sense within the Sass hierarchy. If the developer tools automatically added a sass file, the entry is always at the end of the `files` array. However, your dependencies may require that this order be changed. Check the order in which it was added., because it will always we be added at last.

```

//...
"sass": {
    "entry_points": {
        "shopping": "Modules/MyCoolModule/Sass/_mycoolextension-mycoolmodule.scss",
        "myaccount": "Modules/MyCoolModule/Sass/_mycoolextension-mycoolmodule.scss",
        "checkout": "Modules/MyCoolModule/Sass/_mycoolextension-mycoolmodule.scss"
    },
    "files": [
        "Modules/MyCoolModule/Sass/_mycoolextension-mycoolmodule.scss",
        "Modules/MyCoolModule/Sass/_mycoolmodule.scss",
        "Modules/AdditionalCoolModule/Sass/_mycoolextension-additionalcoolmodule.scss",
        "Modules/AdditionalCoolModule/Sass/_additionalcoolmodule.scss"
            "Modules/AdditionalCoolModule/Sass/sass-extension_myNewSassFile.scss"
    ]
},
//...

```

JavaScript

The `javascript` object declares the paths to each application entry point and all JavaScript files to be loaded when you deploy.

```
//...
```

```

"javascript": {
  "entry_points": {
    "shopping": "Modules/MyCoolModule/JavaScript/Acme.MyCoolExtension.MyCoolModule.js",
    "myaccount": "Modules/MyCoolModule/JavaScript/Acme.MyCoolExtension.MyCoolModule.js",
    "checkout": "Modules/MyCoolModule/JavaScript/Acme.MyCoolExtension.MyCoolModule.js"
  },
  "application": {
    "shopping": {
      "files": [
        "Modules/MyCoolModule/JavaScript/Acme.MyCoolExtension.MyCoolModule.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.Router.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.List.View.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.Edit.View.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.Collection.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.Model.js",
        "Modules/AdditionalCoolModule/JavaScript/Acme.MyCoolExtension.AdditionalCoolModule.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.Router.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.List.View.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.Edit.View.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.Collection.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.Model.js"
      ]
    },
    "myaccount": {
      "files": [
        "Modules/MyCoolModule/JavaScript/Acme.MyCoolExtension.MyCoolModule.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.Router.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.List.View.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.Edit.View.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.Collection.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.Model.js",
        "Modules/AdditionalCoolModule/JavaScript/Acme.MyCoolExtension.AdditionalCoolModule.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.Router.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.List.View.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.Edit.View.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.Collection.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.Model.js"
      ]
    },
    "checkout": {
      "files": [
        "Modules/MyCoolModule/JavaScript/Acme.MyCoolExtension.MyCoolModule.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.Router.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.List.View.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.Edit.View.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.Collection.js",
        "Modules/MyCoolModule/JavaScript/MyCoolModule.Model.js",
        "Modules/AdditionalCoolModule/JavaScript/Acme.MyCoolExtension.AdditionalCoolModule.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.Router.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.List.View.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.Edit.View.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.Collection.js",
        "Modules/AdditionalCoolModule/JavaScript/AdditionalCoolModule.Model.js"
      ]
    }
  }
}

```

```

        }
    },
//...

```

Ssp-libraries

The **ssp-libraries** object declares the paths to all SuiteScript files to be loaded when you deploy.

```

//...
"ssp-libraries": {
    "entry_point": "Modules/MyCoolModule/SuiteScript/Acme.MyCoolExtension.MyCoolModule.js",
    "files": [
        "Modules/MyCoolModule/SuiteScript/Acme.MyCoolExtension.MyCoolModule.js",
        "Modules/MyCoolModule/SuiteScript/MyCoolModule.ServiceController.js",
        "Modules/MyCoolModule/SuiteScript/MyCoolModule.Model.js",
        "Modules/AdditionalCoolModule/SuiteScript/Acme.MyCoolExtension.AdditionalCoolModule.js",
        "Modules/AdditionalCoolModule/SuiteScript/AdditionalCoolModule.ServiceController.js",
        "Modules/AdditionalCoolModule/SuiteScript/AdditionalCoolModule.Model.js"
    ]
}
//...

```

CCT

If your extension is set up to include a CCT, your manifest's metadata includes specific information regarding your CCT.

The **cct** array declares the specific information required to build a CCT in NetSuite. This includes the following attributes:

- **label** (required) – lists the name you gave your CCT.
- **icon** – lists the path of the icon to be displayed in SMT for the CCT.
- **settings_record** (required) – lists the **ID** of the custom record you associate with this CCT.
- **registercct_id** (required) – lists the **Name** field of the CMS Content Type Record for your CCT. This is also the value of the **id** property within the **registerCustomContentType()** method of your CCT module's entry point JavaScript file.
- **description** – lists the description you gave your CCT.

```

//...
"cct": [
{
    "label": "This is My CCT!",
    "icon": "img/cct_acme_mycct_icon.svg",
    "settings_record": "customrecord_cct_acme_mycct",
    "registercct_id": "cct_acme_mycct",
    "description": "My cool CCT does magic!"
},
{
    "label": "This is My Second CCT",
    "icon": "img/cct_acme_mycct2_icon.svg",
    "settings_record": "customrecord_cct_acme_mycct2",
    "registercct_id": "cct_acme_mycct2",
}
]

```

```

        "description": "My cool CCT does magic!"
    }
]
}

```

Page

If your extension is set up to include a new CMS Page Type, you must manually add a page type definition within your extension's manifest.json. When an Extension SuiteApp is installed into a NetSuite account, this entry ensures that the CMS Page Type record is created in NetSuite. This is required if creating a new Page Type for use with Site Management Tools (SMT).

The **page** object includes the **types** array, which introduces each new page type included in the extension. All fields correlate to the CMS Page Type record in NetSuite:

- **name** (required) – Sets the name of the Page Type. This becomes the name of the CMS Page Type record. This must match the **name** attribute when registering your Page Type with **PageTypeComponent.registerPageType()** in your extension. See the help topic [Register Page Types and Templates for SMT](#) for details. This is required.
- **displayName** (required) – This string sets the label displayed in SMT for the Page Type when creating a new page.
- **description** – (optional) This string sets the description of the Page Type.
- **baseUrlPath** (optional) – This string sets the base URL path for all the pages of this page type. For example, if you have a page type called **blog** and a base URL path of **blog**, all blog pages are accessed by [mysite.com/blog/\[page url\]](http://mysite.com/blog/[page url]).
- **settingsRecord** (optional) – This string sets the ID of any custom record you associate with this page type, inheriting the custom record settings when you add or edit the page type.

```

//...
"page": {
    "types": [
        {
            "name": "blog",
            "displayName": "Blog",
            "description": "This is a blog page type",
            "baseUrlPath": "blog",
            "settingsRecord": "custrec_page_type_test"
        },
        {
            "name": "other-page-type",
            "displayName": "My other page type",
            "description": "This is another page type",
        }
    ]
}
//...

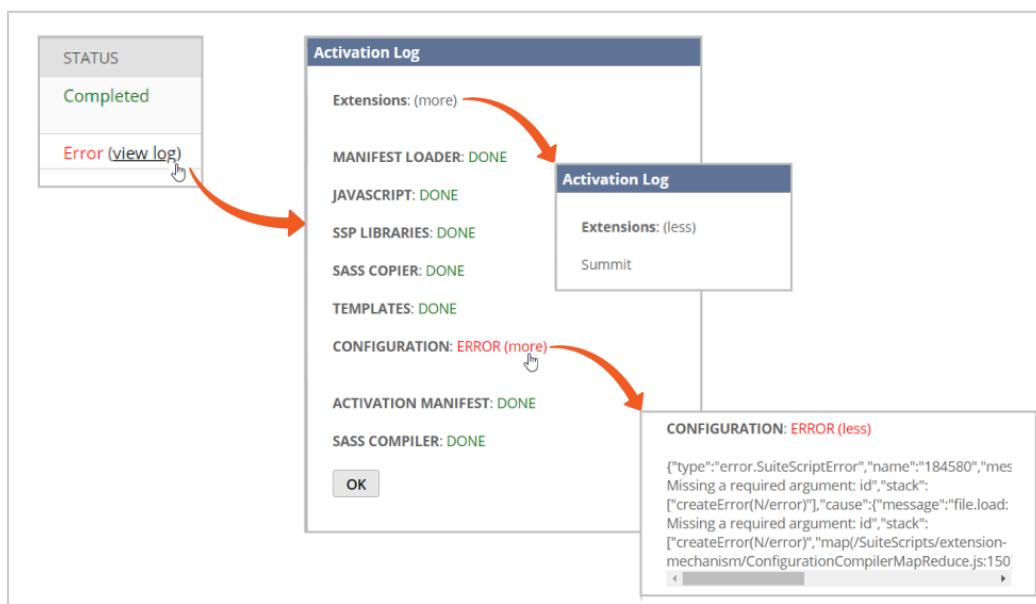
```

Troubleshoot Activation Errors

If you experience any errors during activation, you can use the Extension Manager to troubleshoot. If an activation fails, the **Status** column displays **Error (view log)** and provides a log to help with your investigation.

To troubleshoot errors:

1. In NetSuite, go to Setup > SuiteCommerce Advanced > Extension Manager.
 2. Click the **view log** link next to the error in the **Status** column to open a popup and display the Activation Log.
- The Activation Log provides expandable information to help you troubleshoot the error.
3. On the Activation Log popup, do one of the following:
 - a. Click **(more)** next to Extensions to view a list of the theme and extensions that are part of the failed activation.
 - b. Click **(more)** next to the failed process to expand the error log and troubleshoot the cause.
 - c. Click **(less)** to collapse.



Commerce Custom Fields

Applies to: SuiteCommerce Web Stores

SuiteCommerce lets you access core and custom field values through the Commerce API, Items API, and SuiteScript. You can add custom fields to your SuiteCommerce site in two ways:

- **Create Custom Fields Using an Extension** — When you install and activate the SuiteCommerce Custom Fields extension, you add a subtab to your SuiteCommerce configuration record that lets you specify which custom fields you want to add, the location, and the order in which the fields appear.

The SuiteCommerce Custom Fields extension lets you add fields to Checkout and Product Detail Pages of your website. Custom Checkout fields only support the custom transaction body field type.

- **Create Custom Fields by Customizing Templates** — Alternatively you can configure your site and customize template files as part of a theme to render information or ask for information stored in these fields on your web store's Shopping, Checkout, and My Account pages.

Note: If you are using SuiteCommerce Advanced Aconcagua or later, you implement custom fields as part of a theme or using an extension. If you are using SuiteCommerce Advanced Kilimanjaro or earlier, you implement custom fields by customizing the core SCA source code.

SuiteCommerce supports the following kinds of custom field records:

- **Custom Item Fields** – add custom item field data associated with an item. These fields apply to the Product Details Page (PDP), Checkout, and My Account.
- **Custom Transaction Body Fields** – add custom transaction fields to the body of a web store transaction (order as a whole). You can also include some field metadata (such as the field label or item options). These fields apply to Checkout and My Account.
- **Custom Transaction Line Fields** – add a custom transaction field to the column of a transaction record (across multiple transaction lines in an order). These fields apply to the PDP, Cart, Checkout, and My Account.
- **Custom Transaction Item Options** – add a custom transaction field that applies to a transaction line (one line item within an order). These fields apply to the PDP, Cart, Checkout, and My Account.

Note: Site Builder Reference implementations support custom fields appearing in Checkout and My Account only.

Create Custom Fields Using an Extension

Applies to: SuiteCommerce Web Stores

The SuiteCommerce Custom Fields extension lets you add custom fields to the Checkout application and Product Detail Pages of your website without customizing any source code or themes.

Important: SuiteCommerce extensions are only available if provisioned, installed, and activated for a selected domain in your account. For more information, see the help topic [NetSuite SuiteCommerce Extensions](#).

Note: If you are a SuiteCommerce Advanced customer, you must be implementing the Aconcagua release or later to take advantage of extensions.

You can use custom fields to add more information about items in your web store or add and request additional information about the checkout process. Custom fields are grouped together in a separate block, which you can set off by adding a header.

This topic includes information on the following custom field types:

- [Custom Fields for Checkout](#)
- [Custom Fields for the PDP](#)

Custom Fields for Checkout

The SC Custom Fields extension lets you add custom transaction body fields to the checkout application in SuiteCommerce sites. You can use these fields to request or provide additional information, such as shipping instructions, special order requests, and delivery time slots.

SC Custom Fields lets you add text, check box, and date-type transaction body fields to SuiteCommerce sites. You can add custom fields before or after existing checkout modules. For example, you can include fields before shipping method, billing address, or payment method blocks in the standard checkout flow.

To display custom transaction body fields in the Checkout application of your site, perform the following steps:

1. [Create Custom Fields for Checkout](#)
2. [Configure Custom Fields for Checkout](#)

Create Custom Fields for Checkout

To add new fields to Checkout, you must create custom transaction body fields in your NetSuite account. For more information on how to create custom fields in NetSuite, see the help topic [Custom Transaction Body Fields](#).

 **Note:** The SC Custom Fields extension only supports transaction body fields. To implement custom item, custom transaction line, and custom transaction item option fields, see [Create Custom Fields by Customizing Templates](#).

The SC Custom Fields extension allows the following transaction body field types:

- Check Box
- Date
- Text Area
- Long Text
- Free-Form Text

When you create a transaction body field for the SC Custom Fields extension, be aware of the following:

- The **Field Type** must match the field type you want to include in the checkout application.
- The **Web Store** and **Sale** fields must be checked in the **Applies To** subtab.
- If you select the **Mandatory** box in the **Validation & Defaulting** subtab, the field is mandatory for all domains and you must mark it as required in the SuiteCommerce configuration of all the domains of your website.

If you want the new transaction body field to be required in only one domain, you should mark the field as required in the configuration of that domain only. You do not need to mark it as mandatory during creation.

- The subtab where you want to display the custom field on the Sales Order record should be selected in the **Display** subtab. Otherwise, the field is shown in the **Custom** subtab of the Sales Order record.

 **Note:** The character limitation for the custom fields in SuiteCommerce is set by the field type in NetSuite. For more information on the limits, see the help topic [Table of Custom Field Type Descriptions](#).

When you have created your custom transaction body fields, you are ready to [Configure Custom Fields for Checkout](#).

Configure Custom Fields for Checkout

Use the SC Configuration record to choose which fields to add to Checkout and how they display. You can also set feedback messages that display to users on your site.

Consider the following points when determining the order and positioning of your custom fields:

- You can add a custom field before or after a standard checkout module. For example, if your SC site uses the standard checkout flow, you can add custom fields before or after the shipping method block in the Shipping Address page.

- If you want to change the order of existing modules in the checkout application, you should do this before adding new custom fields. For more information, see [Reorder Checkout Modules](#).
- The order in which you list custom checkout fields in the **Custom Fields** subtab of the SC Configuration record determines the order of the custom fields in Checkout.



Important: Only custom transaction body fields are supported by the SC Custom Fields extension for the Checkout Application. Standard NetSuite transaction body fields cannot be used.

To configure Custom Fields for Checkout:

1. In NetSuite, go to Setup > SuiteCommerce Advanced > Configuration.
2. Select the website and domain where the Custom Fields extension is activated and click **Configure**.
3. Navigate to the **Extensions** tab.
4. In the **Custom Fields** subtab, set the following fields:

Messaging Fields:

Field	Description
Required Field Warning Message	Sets the message to display when the user leaves a required field blank. The placeholder [[field]] is replaced by the field name automatically.
Required Fields General Warning	Sets the message to display at the top of the page when the user leaves a required field blank.
Loading Message	Sets the message to display when the custom fields are loading.
Loading Error Message	Sets the message to display when an error occurs while loading.
Saving Error Message	Sets the message to display when an error occurs while saving.

Checkout Custom Fields:

For each custom field you want to include in Checkout, insert a row in the Checkout Custom Fields table, and complete the following fields as needed.

Field	Description
Field ID	Links the field ID of the custom field you want to employ with a Checkout field. To find the field ID, go to Customization > Lists, Records & Fields > Transaction Body Fields and open the Transaction Body Field record.
Position	Determines the position in Checkout where the custom field appears.
Module	Determines the module in Checkout on which the custom field's position is based. You can choose from the following Checkout modules: <ul style="list-style-type: none"> ▪ Shipping Address ▪ Shipping Method ▪ Gift Certificate ▪ Payment Method ▪ Review Shipping ▪ Review Payment ▪ Terms and Conditions
Field Type	Determines the type of the custom field. The Field Type must be identical to the Field Type selected in the Transaction Body Field record.

Field	Description
Label	Sets the label for the custom field to display to users.
Placeholder	Sets the placeholder text to display in the custom field.
Required	If checked, makes the custom field required for a user in Checkout. You must check this box if the Mandatory field is checked on the Transaction Body Field record.

5. Click **Save**.

Custom Fields for the PDP

Using the SC Custom Fields extension, you can choose to display any standard or custom item field on the PDP. Additionally, experienced customers can add Schema.org item properties to the fields to improve search engine optimization.

Activating the Custom Fields extension adds several native NetSuite fields to the Custom Fields section of the SC Configuration record, which you can use to display additional item details.

You can also create your own custom fields to include on the PDP. Standard or custom item fields display on the PDP in the SKU label format.

To make native NS or custom fields available for use on the PDP, you must:

1. Update Field Sets
2. Configure Custom Fields for the PDP

Update Field Sets

The field sets for a website determine the data that is exposed to site templates. To employ custom fields on the PDP, you must add several fields to the site's **details** field set. For more information on field sets, see the help topic [Define Field Sets](#).

These instructions include the fields added automatically with the extension, but you must also add any additional fields you want to display on the PDP.

To add custom fields to the Web Site Setup Record:

1. Go to Setup > SuiteCommerce Advanced > Set Up Web Site.
2. Click **Edit** next to the website to which you are adding custom fields.
3. Navigate to the **Field Sets** subtab.
4. Add the following fields to the **details** field set:
 - **UPC Code**
 - **Manufacturer**
 - **MPN**
 - **Manufacturer Country**
 - **Item Weight**

For each field, perform the following steps:

- a. Locate the **Fields Included in Field Set** column of the **details** field set and click the **Set** button.
- b. Select the fields from the **Field Name** list and click **Add**.

- c. In the Field Set window, click **Submit**.
- d. In the Field Set row, click **OK**.
5. After adding all of the fields to the field set, click **Save**.

Configure Custom Fields for the PDP

Use the SC Configuration record to choose which fields to add to the PDP. You can also associate each field with a schema property to improve search engine optimization.

To configure Custom Fields for the PDP:

1. In NetSuite, go to Setup > SuiteCommerce Advanced > Configuration.
2. Select the website and domain where the Custom Fields extension is activated and click **Configure**.
3. Navigate to the **Extensions** tab.
4. In the **Custom Fields** subtab, PDP Fields section, set the following fields for each custom field you want to include on the PDP:

Field	Description
Label and Field ID	<p>Links to the label and field ID of the field to include on the PDP.</p> <p>The item field ID is listed as <code>[[fieldid]]</code> to automatically display the value. If an item does not have a value for a field, the field does not display for that item.</p> <p>You can use more than one field within the same line to display the fields in conjunction. For example, if the Label and Field ID field is set to Item Weight: [[weight]] [[weightunit]], the line displays as Item Weight: 7 lb.</p>
Show	If checked, displays this field on the PDP.
Schema.org Property	<p>Links to a schema property tag to improve search engine optimization.</p> <p>You can find a list of suggested schema properties and more information at Schema.org.</p> <p>This feature is for experienced users; NetSuite does not verify if the schema property is correct.</p>

5. Click **Save**.

Create Custom Fields by Customizing Templates

ⓘ Applies to: SuiteCommerce Web Stores | SuiteCommerce Advanced

You can configure your site and customize template files as part of a theme to render information or ask for information stored in these fields on your web store's Shopping, Checkout, and My Account pages.



Important: If you are using SuiteCommerce Aconcagua and later, you implement custom fields as part of a theme. See [Themes](#) for details on creating themes.

If you are using SuiteCommerce Advanced, you implement custom fields by customizing the core SCA source code. See [Customize and Extend Core SuiteCommerce Advanced Modules](#) for details.

You can set up custom item, custom transaction body, and custom transaction column and transaction item option fields in SuiteCommerce by customizing the template files:

- Set Up Custom Item Fields
- Set Up Custom Transaction Body Fields
- Set Up Custom Transaction Line and Transaction Item Option Fields

Set Up Custom Item Fields

Set up custom item fields to appear in the PDP, Checkout, and My Account. This requires setting up the custom item field in NetSuite and overriding the correct template to render the information in your web store.

To set up a custom item field in NetSuite:

 **Note:** The custom item field information rendered to your web store is read-only.

1. In NetSuite, create the custom item field. See the help topic [Custom Item Fields](#) for details.
For a custom item field to render in your SuiteCommerce web store, observe the following:
 - Enter a Label and ID for your custom item. If you do not specify an ID, NetSuite will create one for you. Use these when adding field sets and customizing templates.
 - Set the Subtype (**Applies To** subtab) to either **Both** or **Sale**.
2. In NetSuite, navigate to Setup > SuiteCommerce Advanced > Set Up Web Site and click **edit** next to your site.
3. In the Web Site record, click **Actions > Rebuild Search Index** and wait for the index to build.
4. On the **Field Sets** tab, perform the following as applicable:
 - If adding the field to the PDP, locate the **Details** field set name and add the custom field ID to the **Fields Included in Field Set** column.
 - If adding the field to Checkout or the Purchase History page of My Account, add the field ID to the **Order** field set name and add the custom field ID to the **Fields Included in Field Set** column.

 **Note:** The **order** field set is the default for the Review Order page, but you can change this by editing the **fieldKeys.itemsFieldsAdvancedName** property in the SuiteCommerce Configuration Record. See [Backend Subtab](#) for details.

To display the custom item field in the PDP or in Checkout:

1. As part of your theme, update the appropriate template:
 - **product_details_full.tpl** – exposes the field as read-only information to the PDP.
 - **transaction_line_views_cell_navigable.tpl** – exposes the field as read-only information to the Review Your Order page and in the Order Summary information displayed throughout Checkout.

See [Themes](#) for details on creating themes.

 **Note:** If you implementing a SuiteCommerce Advanced site using the Kilimanjaro release or earlier, you must override the appropriate template. See [Customize and Extend Core SuiteCommerce Advanced Modules](#) for details.

2. Add the following code to any HTML tag within the template that does not have the **data-view** attribute:

```
{model.item.custitem_my_custom_field_id}
```

In this example, `custitem_my_custom_field_id` is the ID of your custom item field

Note: If the custom field record's **Type** is set to **Multiple Select**, the values render on your site separated by commas.

- Save your code and test on a local server or deploy to NetSuite and activate as required.

To display the custom item field in My Account:

- As part of your theme, update the `transaction_line_views_cell_actionable.tpl` template.

See [Themes](#) for details on creating themes.

Note: If you implementing a SuiteCommerce Advanced site using the Kilimanjaro release or earlier, you must override the appropriate template. See [Customize and Extend Core SuiteCommerce Advanced Modules](#) for details.

- Add the following code to the template wherever you want the field to appear:

```
{model.item.custitem_my_custom_field_id}
```

In this example, `custitem_my_custom_field_id` is the ID of your custom item field

- Save your code and test on a local server or deploy to NetSuite and activate as required.

Set Up Custom Transaction Body Fields

Set up custom transaction body fields to appear in Checkout and My Account. This requires setting up the custom transaction body field in NetSuite and overriding the correct template to render the information in your web store.

To set up a custom transaction body field:

- In NetSuite, set up the custom transaction body field. See the help topic [Custom Transaction Body Fields](#) for details.

For a custom transaction body field to render in your SuiteCommerce web store, observe the following:

- Create a Label and ID for your custom item. If you do not specify an ID, NetSuite will create one for you. Use these when configuring your site and customizing templates.
- Check **Sale** and **Web Store (Applies To)** subtab.
- SCA supports the **Mandatory** option in the **Validation & Defaulting** subtab, but as information only. Transaction body fields are not validated in the frontend (client side). If a custom transaction body field is setup as **mandatory**, the field will be mandatory for Quotes as well.

- In NetSuite, go to Setup > SuiteCommerce Advanced > Configuration.
- In the **Advanced** tab and **Custom Fields** subtab, add your transaction body field ID to the **Sales Order** property. See [Custom Fields Subtab](#) for more information on this configuration property.



Note: If any transaction body fields were available in the Vinson release of SCA or earlier, you must expose them in the configuration as well.

To display the custom transaction body field:

You can add a custom transaction body field to any module template linked to a checkout wizard step or in the Purchase History pages of My Account. See [Supported Field Types](#) for a list of supported field types.

- As part of your theme, update the appropriate template:
 - To render information in Checkout, override any applicable checkout wizard template.
 - To render the information in My Account, override the `order_history_details.tpl` or the `order_history_summary.tpl`, as required.

See [Themes](#) for details on creating themes.



Note: If you implementing a SuiteCommerce Advanced site using the Kilimanjaro release or earlier, you must override the appropriate template. See [Customize and Extend Core SuiteCommerce Advanced Modules](#) for details.

- In your template, you can include the custom field's metadata as an option. This is important for rendering the label of a field, for example. The `label`, `type`, and `mandatory` metadata components apply to all supported transaction body field types. The `options` metadata component applies to **List/Record** types only.



Note: After changing the `type`, `label`, or `mandatory` attributes of a transaction body field, your site uses the metadata values for up two hours. If you configure a new transaction body field, and that field's attribute is not in the cache, the cache reloads, attempting to get the requested metadata. If the Sales Order Field ID property does not belong to any transaction body field, the application does not render that field.

To display metadata for the custom field, include the following helpers (where `custbody_my_transaction_body_field` is the ID of your custom transaction body field):



Note: The metadata is available in the Sales Order object as part of the `__customFieldsMetadata` attribute. In your template override, access custom field metadata by introducing the following script (where `custbody_my_transaction_body_field` is the ID of your custom transaction body field).

Metadata Fields	Information	Script
Label	Display the label of the field	<code>model.__customFieldsMetadata.custbody_my_transaction_body_field.label</code>
Type	Display the type of the field	<code>model.__customFieldsMetadata.custbody_my_transaction_body_field.type</code>
Mandatory	Display the Mandatory field. This is information only and is not validated in the frontend (client side).	<code>model.__customFieldsMetadata.custbody_my_transaction_body_field.mandatory</code>
Options	Display the options for a List/Record field.	<code>model.__customFieldsMetadata.custbody_my_transaction_body_field.options</code>

- To display the value of a transaction body field, include the following helper (where `custbody_my_transaction_body_field` is the ID of your custom transaction body field):

```
  {{model.options.custbody_my_transaction_body_field}}
```

4. To request information from a user, include an **input**, **textarea**, or **select** component (where the **name** property is the custom field ID). Whenever the Change event is triggered on the field, the value of the HTML component is set in the model associated with the template.

The following examples use the **Label** metadata to display the label of the custom TBF:

```
<div>
  {{model.__customFieldsMetadata.custbody_my_transaction_body_field.label}}:
    <input type="text" name="custbody_my_transaction_body_field" value="{{model.options.custbody_my_
transaction_body_field}}">
</div>
```

```
<div>
  {{model.__customFieldsMetadata.custbody_my_transaction_body_field_sel.label}}
  <select name="custbody_my_transaction_body_field_sel">
    {{#each model.__customFieldsMetadata.custbody_my_transaction_body_field_sel.options}}
      <option value="{{id}}" {{#ifEquals id ../model.options.custbody_my_transaction_body_field_
sel}selected="true">{{/ifEquals}}>
        {{text}}
      </option>
    {{/each}}
  </select>
</div>
```

```
<div>
  {{model.__customFieldsMetadata.custbody_my_transaction_body_field_2.label}}
  <textarea name="custbody_my_transaction_body_field_2">
    {{model.options.custbody_my_transaction_body_field_2}}
  </textarea>
</div>
```

Supported Field Types

SuiteCommerce supports custom transaction body fields of the following **Type**:

Type	Example HTML5 Elements Used to Render the Field
Check Box	text, checkbox
Currency	text
Date	text, date
Date/Time	text, datetime
Decimal Number	text, number
Document	select
Email Address	text, email
Free Form Text	text
Hyperlink	url

Type	Example HTML5 Elements Used to Render the Field
Inline HTML	text
Integer Number	text, number
List/Record	input, select
Long Text	text
Multiple Select	select
Password	text, password
Percent	text
Phone Number	text, tel
Rich Text	text
Text Area	text, textarea
Time of Day	text, time

Set Up Custom Transaction Line and Transaction Item Option Fields

Set up custom transaction line fields to appear in your PDP, Cart, Checkout, and My Account. This requires setting up the custom fields in NetSuite. SuiteCommerce provides a set of default templates (listed below) to render these fields, but you can override these as required.

- product_views_option_color.tpl
- product_views_option_dropdown.tpl
- product_views_option_radio.tpl
- product_views_option_text.tpl
- product_views_option_textarea.tpl
- product_views_option_email.tpl
- product_views_option_phone.tpl
- product_views_option_currency.tpl
- product_views_option_float.tpl
- product_views_option_integer.tpl
- product_views_option_percent.tpl
- product_views_option_password.tpl
- product_views_option_url.tpl
- product_views_option_timeofday.tpl
- product_views_option_datetimetz.tpl
- product_views_option_tile.tpl
- product_views_option_checkbox.tpl
- product_views_option_date.tpl



Note: Transaction line fields and transaction item options render in the same locations on your site. To set up these fields, follow the same set up and configuration procedures and customize the same template file.

To set up a Custom Transaction Line and Transaction Item Option:

1. In NetSuite, set up the transaction line field or transaction item option. See the help topics [Custom Transaction Line Fields](#) and [Custom Transaction Item Options](#) for details.

For a custom transaction line or transaction item option to render in your SuiteCommerce web store, observe the following:

- Enter a label and ID for your custom item. If you do not specify an ID, NetSuite will create one for you. Use these when customizing templates to render the field.
 - To enable a custom transaction line field to render in the PDP, check **Sale Item** and **Store Item (Applies To)** subtab.
 - To enable a custom transaction item option to render in the PDP, check **Sale** and **Web Store (Applies To)** subtab.
 - SCA supports the **Mandatory** option in the **Validation & Defaulting** subtab. Some of these fields are validated in the frontend (client side). Others are validated in the backend.
 - SCA does not support all transaction item option types.
 - All the transaction line fields and item options are displayed automatically if the **Show Only Items Listed in: ItemOptions and Custom Transaction Line fields** property is unchecked. Enabling this property only displays fields with the ID specified in the item options and custom transaction line field's **Item options and custom transaction line fields** property.
2. In NetSuite, navigate to Setup > SuiteCommerce Advanced > Configuration.
 3. In the **Shopping Catalog** tab and **Item Options** subtab, check the **Show Only Items Listed in: Item Options and Custom Transaction Line Fields** property and set up your item options fields to display. See [Item Options Subtab](#) for more information.

Supported Field Types

SuiteCommerce supports transaction line fields and transaction item options of the following **Type**:



Note: Some of these fields are validated on the frontend (client side). Others are validated on the backend (NetSuite). Invalid fields in the frontend result in error messages displayed at the field where the error occurred. Invalid fields in the backend result in error messages at the top of the page.

Type	Validation	Example HTML5 Elements Used to Render the Field
Check Box	Backend	checkbox
Currency	Frontend	text
Date	Backend	date, datepicker (Bootstrap)
Date/Time	Backend	text
Decimal Number	Frontend	number
Email Address	Frontend	email
Free Form Text	Frontend	text

Type	Validation	Example HTML5 Elements Used to Render the Field
Hyperlink	Frontend	url
Integer Number	Frontend	number
List/Record	Frontend	input, select
Password	Backend	password
Percent	Frontend	text
Phone Number	Frontend	tel
Text Area	Frontend	textarea
Time of Day	Backend	text

Theme and Extension SuiteApps

 **Applies to:** SuiteCommerce Web Stores

SuiteCommerce lets you bundle themes and extensions as SuiteApps, making them available for distribution to other NetSuite accounts. Although some steps to bundle a SuiteApp are already documented in the help topics, you need to be aware of some settings and processes specific to bundling SuiteCommerce theme and extensions.

Before You Begin

Before you bundle a SuiteCommerce theme or extension, be aware of the following:

- These procedures assume that you have created or updated a theme or extension, thoroughly tested it on a local server or on a test domain, and that you have deployed the files to a NetSuite account. Refer to the following help topics for more information on building themes and extensions:
 - [Themes](#) – explains how to create themes.
 - [Extensions](#) – explains how to create extensions.
- You must deploy themes and extensions to the NetSuite Account where you are creating the SuiteApp.
- The SuiteCommerce Extensions Management SuiteApp provides a default installation script. Use this baseline file to create unique installation scripts for each SuiteApp you build.
- You must use semantic versioning (SemVer) notation when creating a theme or extension. This allows SuiteApps to be updated over time. The developer tools prompt you for this information when you deploy your files. SuiteCommerce themes and extensions require a three-digit notation (Major.Minor.Patch). For more information, see <https://semver.org/>.

Bundle Themes and Extensions as SuiteApps

To bundle SuiteCommerce themes and extensions, follow these steps:

- [Step 1: Prepare Themes and Extensions for Bundling](#)
- [Step 2: Create a Unique Installation Script for Your Theme or Extension SuiteApp](#)
- [Step 3: Create the SuiteApp for Your Themes or Extensions](#)

Step 1: Prepare Themes and Extensions for Bundling

The first step is to prepare any folders containing your themes or extensions to enable bundling.

To prepare your theme or extension for bundling:

1. In your NetSuite File Cabinet, navigate to the folder containing your theme or extension. This folder is the one containing your theme and extension files, including the manifest.json file. For example:

```
SuiteScript/Deploy_Extensions/<VendorName>/
<ThemeOrExtensionName>@<Version>/
```
2. Click **edit** next to the extension folder you are including in the bundle.
3. In the Document Folder record, check the **Available for SuiteBundles** option.
4. Click **Save**.

5. Repeat this for every theme and extension you want to include in a SuiteApp.

You are now ready to create an installation script.

Step 2: Create a Unique Installation Script for Your Theme or Extension SuiteApp

Before bundling your themes or extensions as a SuiteApp, you must complete the following steps:

1. Create an installation scrip file.
2. Create an Installation Script record.
3. Deploy your installation script.

A bundle installation script is essential to allow SuiteApp installation, update, and uninstallation processes to run properly. The SuiteCommerce Extension Management SuiteApp comes with a default bundle installation script that you can use, titled: **ExtensionBundle.js**. As a best practice, copy the default file to create a unique script for each SuiteApp you want to build.



Important: Never edit the default ExtensionBundle.js file directly. As a best practice, create a unique installation script file for each theme or extension SuiteApp you want to build, even if you do not include any customizations. Store your installation scripts in a location associated with your theme and extension files.

This default script includes the necessary functions to install, update, or uninstall the SuiteApp and its contents. This includes the following trigger functions:

- beforeInstall
- afterInstall
- afterUpdate
- beforeUninstall



Note: You only need one installation script per SuiteApp. The script applies to any included themes and extensions.

To create an installation script file:

1. In the NetSuite File Cabinet, browse to **SuiteBundles/Bundle xxxxxx**, where **xxxxxx** equals the Bundle ID of the SuiteCommerce Extension Management SuiteApp.
See the help topic [Install Your SuiteCommerce Application](#) for the latest Bundle ID of the SuiteCommerce Extension Management SuiteApp.
2. Make a copy of the default ExtensionBundle.js file:
 - a. Click **Copy Files**.
 - b. Expand the **Filters** area.
 - c. In the **Copy To** list, select a location in the File Cabinet to store your copy.
 - d. Check the box next to ExtensionBundle.js.
 - e. Click **Copy**.
 - f. Rename your new file as needed. (Optional)
To rename a file, navigate to its location and edit the file record.
3. Edit your new installation script file to meet your needs (optional).

The default script can handle most requirements. However, if you have custom needs or want to make special validations or operations in NetSuite records, you can edit the script to meet your needs. For specific information and instructions regarding installation script files, see the help topic [SuiteScript 2.0 Bundle Installation Script Type](#).



Important: If you need to edit ExtensionBundle.js, maintain the validations and operations that came with default file.



Note: The ExtensionBundle.js file that comes with the 18.2 release of the SuiteCommerce Extension Management SuiteApp support updates. However, if you previously created an extension as a SuiteApp prior to the 18.2 release, we recommend recreating the SuiteApp with the latest update to take advantage of the version management functionality. This includes creating a new installation script with SuiteScript 2.0.

To create an Installation Script record:

1. Go to Customization > Scripting > Scripts > New.
2. In the Script File field, click **List**.
3. In the **-All-** list, scroll to select the location of your installation script (this is your copy of ExtensionBundle.js).
4. Select your installation script from the list.
5. Click **Create Script Record**.
6. From the Select Script Type list, select **Bundle Installation**.
7. In the new Script record, name your bundle and provide a description (optional).
8. In the **Scripts** tab, confirm that the following script fields populated correctly.
 - **Before Install Function:** beforeInstall
 - **After Install Function:** afterInstall
 - **After Update Function:** afterUpdate
 - **Before Uninstall Function:** beforeUninstall
9. Click **Save**.

To deploy your installation script:

1. Create a Script Deployment. See the help topic [Steps for Defining a Script Deployment](#).
2. Name your script and provide an ID, beginning with underscore (_).
3. Set the **Status** field to **Released**.



Important: Failure to set this field to **Released** results in a failed installation.

4. Choose the **Log Level** to meet your needs.
5. When you have finished, click **Save**.

You are now ready to create your SuiteApp.

Step 3: Create the SuiteApp for Your Themes or Extensions

After you have successfully created and deployed your installation script, you are ready to build your SuiteApp. Follow the instructions detailed in [Creating a Bundle with the Bundle Builder](#).

Observe the following information when building a SuiteApp for SuiteCommerce themes and extensions:

- When choosing the installation script, select the script you created earlier.
- You cannot use the same Installation Script record across multiple SuiteApps when bundling SuiteCommerce themes or extensions. You must select a unique Installation Script record for each SuiteApp.
- When prompted to select objects for the SuiteApp, browse to **File Cabinet/Folders** and choose the objects associated with each theme or extension you are including in your SuiteApp. Look for the objects that match your deployed themes and extensions.
- You can create theme and extension SuiteApps as managed or unmanaged bundles.
- You can bundle any number of themes and extensions within one SuiteApp.

Update Themes and Extensions

This topic explains how to update SuiteCommerce themes and extensions that are part of a SuiteApp.

Before You Begin

Before releasing an updated SuiteCommerce theme or extension, be aware of the following best practices:

- Business users can choose whether or not to activate an updated theme or extension. Therefore, always develop your theme and extension updates to be backwards compatible with previous versions.
For example: NetSuite records are maintained within a NetSuite account. If your extension update requires altering a custom record, you could break backwards compatibility with previous versions of the extension.
- Always test theme and extension functionality and backwards compatibility on a development or sandbox account before releasing the bundle.
- If you bundled a theme or extension prior to the 18.2 release of the SuiteCommerce Extensions Management SuiteApp, we recommend rebuilding the installation script using SuiteScript 2.0 the next time you update a theme or extension included in the bundle.
- You must increment the version of at least one theme or extension when updating a SuiteApp. If the SuiteApp does not include a theme or extension with a later version, the update fails.
- The SuiteApp version has no programmatic connection with the versioning system of themes and extensions.
- If your theme or extension update requires customizing the installation script, you can edit the existing script. You do not need to build a new script.
- These procedures only apply to themes and extensions that are being released as part of a bundled SuiteApp. Updates to custom themes or extensions (as developed by internal site developers) are made using the developer tools.

Update Themes and Extensions Within a SuiteApp

To update a theme or extension that is part of a SuiteApp:

1. Update your theme or extension as required. See [Customization](#) for more information.

2. Deploy your updated theme or extension to the NetSuite account used to create the original SuiteApp.

Increment each version using a three-digit SemVer notation (Major.Minor.Patch). Failure to use this results in failed SuiteApp updates. For more information on semantic versioning, see <https://semver.org/>.

3. Confirm that the manifest.json file associated with the updated theme or extension reflects the correct version.

- a. In NetSuite, navigate to the location where you deployed your theme or extension update.

For example:

```
SuiteScript/Deploy_Extensions/<VendorName>/  
<ThemeOrExtensionName>@<Version>/
```

- b. Open the associated manifest.json file and confirm the version.

The manifest.json file should reflect the latest version in SemVer notation. For example, if you deployed an update to MyCoolExtension 1.0.0 to version 2.0.0, your manifest.json file should look similar to the following example:

```
{
  "fantasyName": "My Cool Extension!",
  "name": "MyCoolExtension",
  "vendor": "Acme",
  "type": "extension",
  "target": "SuiteCommerce",
  "version": "2.0.0",
  "description": "This is my cool update!",
  // ...
}
```

- c. If the manifest does not reflect the correct version, redeploy or edit the manifest file in your file cabinet.
4. If your theme or extension update requires changes to the installation script, edit the installation script as needed.

See the help topic [SuiteScript 2.0 Bundle Installation Script Type](#) for details.

5. Edit the SuiteApp using SuiteBundler in NetSuite. See the help topic [Creating a Bundle with the Bundle Builder](#) for details.

Core SCA Source Code

 **Applies to:** SuiteCommerce Advanced

If you are customizing Javascript, SuiteScript or Configuration objects that are not available using the Extensibility API, you need to use the core SCA developer tools and customization procedures described in this section.

More specifically, you must read and understand this section if:

- You are customizing a site that implements the Kilimanjaro release of SCA or earlier.
- You are customizing a site that implements the Aconcagua release of SCA or later and need to access JavaScript, SuiteScript, or configuration objects that are not available using the Extensibility API. See the help topic [Extensibility Component Classes](#) for a list of components exposed using the Extensibility API.



Important: If you are implementing the Aconcagua Release of SCA or later, the best practice is to use themes and extensions to customize your site. To customize HTML or Sass files for these implementations, you must use the theme developer tools. See [Theme Developer Tools](#) and [Extension Developer Tools](#) for details on using these tools.

Core SuiteCommerce Advanced Developer Tools

 **Applies to:** SuiteCommerce Advanced

This section outlines the process for using the core SuiteCommerce Advanced (SCA) developer tools. Follow the procedures outlined in this section if:

- You are developing an SCA site using the Kilimanjaro release or earlier.
- You are developing an SCA site using the Aconcagua release or later and you are not using extensions to interact with the Extensibility API.



Important: If you are implementing the Aconcagua Release of SCA or later, the best practice is to use themes and extensions to customize your site. However, if your customizations require access to objects not available using the Extensibility API, use the procedures outlined in this section to extend code using custom modules. These tools can be run in parallel with the theme and extension developer tools.

The core SCA developer tools let you customize the application from your local development environment, then deploy your customized application to NetSuite or a local environment for testing. You can also create new features that extend the functionality of existing modules or create your own custom features in SuiteCommerce Advanced.

The SCA developer tools are based on open-source software and are integrated within your existing development environment. These tools enable you to do the following:

- Store code locally within a version control system.
- Create and edit code locally in your preferred text editor or IDE.
- Compile the application locally.

- Deploy the application to NetSuite or a local server environment for testing. You can also deploy directly to your production or sandbox accounts in NetSuite.

This section assumes that the SCA SuiteApp is installed into your account and that you have already followed the instructions to download and set up Node.js and Gulp.js. If you have not already done so, perform the steps outlined in the following topics:



Note: SuiteApp installation is generally performed by your system or NetSuite administrator. This process installs all of the required application files in NetSuite. It also installs a downloadable zip file containing the source code for SCA.

After the applicable SuiteApp is installed, the source files are available in the file cabinet. You can download this zip file to your local environment.

- [Install Node.js](#)
- [Install Gulp.js](#)



Note: See [The Build Process](#) for more information about how Gulp.js performs these tasks.

Permissions

To use Gulp.js to deploy source files to NetSuite, you must use either the System Administrator role, the Store Manager role, or a custom role with the following permissions set to Full:

- Documents and Files
- Website (External) publisher
- Web Services

Set Up Your Development Environment



To set up your development environment you must install Gulp.js and download the SuiteCommerce Advance source files from NetSuite. You must also install Node.js which is a JavaScript development platform required by Gulp.js.

Download the Source Files

After installing Node.js and Gulp.js, you have completed the basic installation requirements for the developer tools. To begin locally editing and customizing SuiteCommerce Advanced, you must first download the source files from the NetSuite file cabinet. Generally, you only need to download the source files the first time you set up your local development environment. You may also need to download the source files to ensure you have the correct base version or the most recent version after a bundle update.

Your File Cabinet includes the following two implementations:

- **Source** – this implementation contains all of the original installation files. It is locked to prevent any changes. The Source implementation can be used in the future for creating diffs against your customizations to troubleshoot issues.
- **Development** – the Development implementation is where you begin your work.

In the _Sources directory of the Development implementation, there is a zipped source file available. Download this file to your local development environment to begin working on your customizations and to create Distribution files. Appropriate default Touch Points are automatically deployed to this implementation. After deployed to a website, your site is available at the domain defined for that site.



Important: With the Aconcagua release of SuiteCommerce Advanced, these implementations no longer contain HTML and Sass files. To customize the look and feel of your site, you must use the theme developer tools to implement themes. See [Theme Developer Tools](#) for details. These tools can be operated simultaneously with the SCA developer tools.

To download the SuiteCommerce Advanced source files:

1. Login to your NetSuite account where SuiteCommerce Advanced is installed.
2. Access the File Cabinet at Documents > Files > File Cabinet.
3. In the File Cabinet, go to Web Site Hosting Files > Live Hosting Files > SSP Applications > [SSP Application] > Source > _Sources.
4. Click the name of the .zip file to download it.
5. Extract the .zip file to the directory where you want to store your source files.

By default, the zip file extracts into a top-level directory named **SuiteCommerce Advanced <release_name>**. You can change the name of this top-level directory, if necessary. However, do not change the names of subfolders, files, or their structure within this directory.

After downloading and extracting the zip files, you can begin editing or customizing SuiteCommerce Advanced. If necessary, you can add these files to a version control system.

Install Additional Files in the Source Directory

After downloading and extracting the SuiteCommerce Advanced source directory, you must perform several tasks within the top-level directory to finish installing the developer tools. These tasks are:

- Run **npm** install to install addition Node.js files.
- Run **gulp** to create additional directories required by the developer tools.

To install additional required files:

1. Depending on your platform, open a command line or terminal window.
2. Go to the top-level directory of the SuiteCommerce Advanced source files downloaded previously. This directory contains the **Modules** directory and is where additional files are created.



Note: Additional commands in subsequent steps are also performed within this directory.

3. Install additional Node.js packkages using following command:

```
npm install
```

This command installs the dependencies required by the developer tools in the root directory of the SuiteCommerce Advanced source. These files are stored in the **node_modules** directory.



Note: This command may take several minutes to complete.

4. Run Gulp.js using the following command:

```
gulp
```



Note: If you encounter Sass Compilation Errors when running Gulp, you need to manually correct the compilation errors. For more information, see [Troubleshooting the Developer Tools](#).

The first time you run this command, it creates a folder called **LocalDistribution** which contains the combined SuiteCommerce Advanced application. After this command completes successfully, you will see this folder at the top level of the SuiteCommerce Advanced source directory.

The files installed in this procedure are required by the build process. They are installed in the root directory of the SuiteCommerce Advanced source files. Each time you download a new version of the source files, you must perform these procedures in the new version of the SuiteCommerce Advanced source directory. Also, if you have multiple copies of this directory in your development or test environment, you must run these procedures on each copy.

Set Up SCA 2019.1 for Theme and Extension Developer Tools

Applies to: SuiteCommerce Advanced

This topic applies to developers implementing the 2019.1 release of SuiteCommerce Advanced who have not been updated to the 2019.1 release of the SuiteCommerce Extensions Management bundle.

Until such time that your account is updated to the 2019.1 release of the SuiteCommerce Extensions Management bundle, you must use the developer tools provided with the 2018.2.1 release to create themes and extensions for the 2019.1 release of SuiteCommerce Advanced.

Before doing so, however, you must customize your developer environment as described in this section. The customization you create depends on the developer tools you intend to use (theme or extension developer tools).

This customization alters the **whoService()** method in the `gulp/tasks/local.js` file. The **whoService()** method returns information used by the application's `local.ssp` file to know the resources and location that must be loaded. You must edit the `url` for the `css`, `templates`, and `js_extensions` objects to use the correct tools. The URL you specify depends on the tools you intend to use.

Refer to the following table for port designations based on the developer tools you intend to use:

2018.2.1 Developer Tools	Local Host Port
Theme Developer Tools	7778
Extension Developer Tools	7779

To customize your developer environment for SCA 2019.1:

- If you have not yet done so, complete set up of your SCA developer environment. See [Download the Source Files and Install Additional Files in the Source Directory](#) for details.
- Access your root source directory and open the following file:
`gulp/tasks/local.js`
- Customize the `css` object:
 - Locate the following lines within the `whoServices()` method:

```
var css = {
  tag: 'link'
, resource: 'css'
```

```

    , url: `${protocol}://${host}/css/${app}.css`
}
```

- b. Replace `, url: `${protocol}://${host}/css/${app}.css`` with the following.

In the following code, replace [PORT] with the local host port for the tools you intend to use.

```

, url: `http://localhost:[PORT]/tmp/css/${app}.css`
```

For example, to use the theme developer tools, your code should look like this:

```

var css = {
    tag: 'link'
, resource: 'css'
, url: `http://localhost:7778/tmp/css/${app}.css`
}
```

4. Customize the `template` object:

- a. Locate the following lines within the `whoServices()` method:

```

, templates = {
    tag: 'script'
, resource: 'templates'
, url: null
}
```

- b. Replace `, url: null` with the following.

In the following code, replace [PORT] with the local host port for the tools you intend to use.

```

, url: `http://localhost:[PORT]/tmp/${app}-templates.js`
```

5. If you are using the extension developer tools, customize the `js_extensions` object:

- a. Locate the following lines within the `whoServices()` method:

```

, js_extensions = {
    tag: 'script'
, resource: 'js_extensions'
, url: null
};
```

- b. Replace `, url: null` with the following.

```

, url: `http://localhost:7779/tmp/extensions/${app}_ext.js`
```



Note: If you are using the theme developer tools, keep the `url` property set to `null`.

6. Find the following lines:

```

var response;
if(process.is_SCA_devTools)
{
    css.url = null;
```

```

        response = [
            css
        ,   require_patch
        ,   requirejs
        ,   templates
        ,   js_core
        ,   js_extensions
    ];
}

```

- Comment out the following line:

```
css.url = null;
```

Your resulting code should look like this snippet:

```

var response;
if(process.is_SCA_devTools)
{
    //css.url = null;

    response = [
        css
    ,   require_patch
    ,   requirejs
    ,   templates
    ,   js_core
    ,   js_extensions
];
}

```

- Save the file.

You should now be able to use the 2018.2.1 theme and extension developer tools to create themes and extensions on your SCA site.

SCA on a Local Server



In addition to deploying to NetSuite, you can also run SuiteCommerce Advanced on a local server to quickly test changes you make to the application. The local server is installed with the as part of the Node.js installation and uses the Express web framework. When you enter the **gulp local** command, the server starts automatically.

When the server starts, Gulp.js initializes watch tasks that listen for changes to files in the JavaScript, Templates, or Sass directories. When you save changes to a file, gulp automatically recompiles the source files and updates the LocalDistribution directory. Gulp also outputs a message to the console when an update occurs.

The local server is primarily used to test frontend changes. When accessing the SuiteCommerce Advanced application on the local server, backend services used by the application are run on NetSuite. Therefore, any local changes you make to services or backend models must be deployed to NetSuite before they are accessible to the local server. Also, you must deploy any changes to services or backend models to NetSuite in order for them to work on the local server.



Note: If you modify the distro.json file to add additional files or modules, you must restart your local server to see changes.

To deploy to the local server:

1. Go to your command line or terminal window.
2. Enter the following command from the top-level directory of the SuiteCommerce Advanced source files (the same directory used during the developer tools installation):

gulp local

If this is the first time you are running **gulp local** in this directory, this command creates a sub directory called LocalDistribution. It then compiles the source files and outputs them to this directory.

3. Navigate to the local version of the application using one of the following URLs:
 - **Shopping:** http://<DOMAIN_NAME>/c.<ACCOUNT_ID>/<SSP_APPLICATION>/shopping-local.ssp
 - **My Account:** http://<DOMAIN_NAME>/c.<ACCOUNT_ID>/SSP_APPLICATION/my_account-local.ssp
 - **Checkout:** http://<DOMAIN_NAME>/c.<ACCOUNT_ID>/SSP_APPLICATION/checkout-local.ssp

In the above URL patterns, you must replace the following variables with values for your specific environment:

- *DOMAIN_NAME* — replace this value with the domain name configured for your NetSuite website record.
- *ACCOUNT_ID* — replace this value with your NetSuite account ID.
- *SSP_APPLICATION* — replace this value with the URL root that you are accessing.

The URLs you use should be similar to the following examples:

<http://www.mysite.com/c.123456/my-sca-ssp/shopping-local.ssp>

http://www.mysite.com/c.123456/my-sca-ssp/my_account-local.ssp

<http://www.mysite.com/c.123456/my-sca-ssp/checkout-local.ssp>



Note: When accessing the secure checkout domain using HTTPS on the local server, you must use a different URL. See [Secure HTTP \(HTTPS\) with the Local Server](#) for more information.

When the local server is running, Gulp.js automatically compiles the application when you save any changes within the Modules directory. You can refresh your browser to see the changes.

Deploy to NetSuite

After installing and configuring Node.js and Gulp.js in your local developer environment as described in the section [Developer Environment](#), you must configure your environment to connect with and deploy to a NetSuite SSP Application.

To configure deployment to NetSuite, you must perform the following:

- [Deploy Local Source Files to an SSP Application in NetSuite](#)
- [Deploy the SSP Application to Your Site](#)

Deploy Local Source Files to an SSP Application in NetSuite

SuiteCommerce Advanced enables you to deploy directly to an SSP application in NetSuite. When the SuiteCommerce Advanced bundle is installed in your NetSuite account, the following SuiteCommerce

Advanced distributions are created in Web Site Hosting Files > Live Hosting Files > SSP Applications > NetSuite Inc. - SCA <version>:

- **Source**– all files in this distribution are locked down. This includes all of the original SSP applications and supporting files as well as the sources.zip file to be downloaded and used for local development.
- **Development** – the development folder is where you deploy your local customizations to.

Use Gulp.js commands to deploy the files generated earlier in the LocalDistribution folder in your NetSuite File Cabinet.

To deploy local source files to a NetSuite SSP application:

1. Go to your command line or terminal window.
2. From the top-level directory of the SuiteCommerce Advanced source files (the same directory used during the developer tools installation), enter the following command.
`gulp deploy`
If this is the first time you are running `gulp deploy` in this directory, this command creates a sub directory called **DeployDistribution**. It then compiles the source files and outputs them to this directory.
3. When prompted, enter your NetSuite email and password.



Note: The developer tools do not support emails or passwords containing special characters such as + and %.

4. When prompted, select the NetSuite account where SuiteCommerce Advanced is installed.



Important: You must log in using the SCDeployer role to access your NetSuite account. Failure to use this role may result in an error. See [Developer Tools Roles and Permissions](#) for instructions on setting up this role.

5. When prompted to choose your Hosting Files folder, select the Live Hosting Files option.
6. When prompted to choose your Application Publisher, select the NetSuite Inc. - SCA <version> option.
7. When prompted to choose your SSP Application, select the Development option.

After entering your connection settings, the contents of the DeployDistribution folder on your local system are uploaded to the NetSuite file cabinet. This process may take a few minutes. Wait for the process to complete before proceeding.



Note: The first time you run the `gulp deploy` command, the connection settings are stored in the `.nsdeploy` file in the root directory of your source SuiteCommerce Advanced files. During subsequent deployments only the login credentials are required. If you need to change the SSP application you are deploying to, you can manually edit the `.nsdeploy` file with the updated information. For details, see [Changing Your Connection Information](#). Also, during the initial deployment to NetSuite, the Gulp.js commands create a manifest file within the NetSuite file cabinet. This file contains the list of files uploaded and a hash of its content. On subsequent deployments, the Gulp.js commands use the manifest file to determine new or changed files. Only those files are updated during deployment to the NetSuite file cabinet.

Deploy the SSP Application to Your Site

After installing the SuiteCommerce Advanced SuiteApp, the Development SSP application is automatically configured with all of the required scripts and appropriate touch points. You must deploy the SSP application to the web site configured within NetSuite.

To deploy the Development SSP application to your site:

1. Go to Setup > SuiteCommerce Advanced > SSP Applications and then click **View** next to the SuiteCommerce Advanced - Dev <version> application.
2. In the Site dropdown field, select the site you want to deploy this application to.



Note: Only sites already set up are available for selection. If you have not set up a web site record go to Setup > SuiteCommerce Advanced > Web Site Set Up > New. For detailed instructions, see the help topic [Getting Started](#).

3. Click **Save**.

After the SSP application has been deployed to your site you can view the site by navigating to the domain defined for that site.

Gulp Command Reference for SCA Developer Tools

The following table lists the most commonly used gulp commands:

Command	Description
gulp	Creates the LocalDistribution directory if it does not exist. This command also compiles the source modules into a deployable application
gulp deploy	Compiles the application and deploys it to NetSuite. As part of the compilation process, gulp deploy minimizes the application by removing all white space. This command also creates the DeployDistribution directory if it does not exist. See Contents of the DeployDistribution and LocalDistribution Directories for information on the output of this command. In addition to compiling the application, this command creates the .nsdeploy file if it does not exist.
gulp deploy --no-backup	Compiles and deploys the application, but does not upload a backup of the source files.
gulp deploy --nouglify	Compiles and deploys the application like gulp deploy , but does not compress the source files. This command is useful for debugging because it produces output files that are easier to read.
gulp deploy --source <source_type>	Compiles and deploys only the type of source files specified. Possible options are: <ul style="list-style-type: none"> ▪ javascript ▪ ssp-libraries ▪ ssp-files ▪ services
gulp deploy --to	Enables you to deploy to a different NetSuite account by overriding the settings defined in the .nsdeploy file.
gulp jshint	Runs the JSHint utility on the SuiteCommerce Advanced source files. This utility verifies and validates JavaScript files to help detect possible errors.
gulp local	Compiles the Sass and template files into a functional application, but does not compile the JavaScript files of the application. After compilation, this command starts a local server. This server watches for changes to the SuiteCommerce Advanced source files. After the server starts, any changes you make to a JavaScript are automatically recompiled and visible in the browser. The server also watches for changes to the Sass and template files. See SCA on a Local Server for more information.

	This command outputs files to the LocalDistribution directory. See Contents of the DeployDistribution and LocalDistribution Directories for information on the output of this command.
gulp clean	Removes the DeployDistribution and LocalDistribution directories and the .nsdeploy file.
gulp local styleguide	<p>This command compiles your Sass, parses all KSS blocks declared in the Sass files, and creates a style guide accessible in your localhost (localhost:3000/).</p> <p>This command requires the Kilimanjaro release of SuiteCommerce Advanced or later. See Style Guide for details.</p>

The Build Process

To understand how the developer tools work and why they are necessary, it is important to understand how the SuiteCommerce Advanced source files are organized and how the application is compiled.

The SuiteCommerce Advanced Source Directory

You can download the SuiteCommerce Advanced source files directly from the File Cabinet in NetSuite. These files are contained in the top-level directory. The following table describes each of the files and folders contained in this directory:

File / Folder	Description
LocalDistribution	<p>Created when you run the <code>gulp local</code> command. This directory contains all of the files associated with the compiled application used by the local server.</p> <p>When you run this command Gulp.js outputs the compiled application files to this directory. After compilation, the contents of this directory are deployed to the local Node.js server.</p> <p>See Contents of the DeployDistribution and LocalDistribution Directories for information on the contents of this directory.</p> <div style="background-color: #ffffcc; padding: 10px;">  Important: Do not manually edit the files in this directory. It is created and updated when you run the gulp command. </div>
DeployDistribution	<p>Contains all of the files associated with the compiled application. This file is created when you run the <code>gulp deploy</code> command.</p> <p>When you run the command the compiled application files are output to this directory. After compilation, the contents of this directory are deployed to your NetSuite account.</p> <p>Contents of the DeployDistribution and LocalDistribution Directories</p> <div style="background-color: #ffffcc; padding: 10px;">  Important: Do not manually edit the files in this directory. It is created and updated when you run the gulp command. </div>
gulp	Contains all of the files required by Gulp.js. This file is created when Gulp.js is installed. You should not edit the files within this directory.
Modules	Contains source code for all of the SuiteCommerce Advanced modules.
node_modules	Stores the dependencies and other files required by Node.js. This directory is created when running the <code>npm install</code> command.
.nsdeploy	Defines how Gulp.js connects to your NetSuite account during deployment. This file is created the first time you run the <code>gulp deploy</code> command.

distro.json	Lists all of the modules used by the SuiteCommerce Advanced application. It also specifies the name and version number of the distribution.
gulpfile.js	Contains all the JavaScript code for Gulp.js. You should only edit this file if you need to add a new task to Gulp.js .

The distro.json and ns.package.json Files

To determine which files to include with the Distribution folder of SuiteCommerce Advanced, Gulp.js uses two types of configuration files. These files are:

- distro.json
- ns.package.json

The distro.json file exists in the top level of the SCA directory. This file defines which modules are included within SuiteCommerce Advanced. It also specifies additional configuration information.

Gulp.js parses this file to determine which files within the Modules directory to combine or copy into the Distribution folder. The distro.json file contains the following parameters:

- **name**: specifies the name of the application. By default this value is SuiteCommerce Advanced <version>.
- **version**: specifies the application version.
- **modules**: lists the modules SuiteCommerce Advanced uses. Each module is listed on a separate line and uses the following format:

"<module_name>": "<module_version>"

By default the SuiteCommerce Advanced modules are listed in alphabetical order because there are no dependencies between modules. However, when customizing SuiteCommerce Advanced, your modules may have dependencies on other modules. In that case, you must ensure that any required modules are listed before other modules.

- **taskConfig**: defines the gulp tasks required by the build process. In general, you do not need to edit these tasks..
- **copy**: specifies the source and target paths when copying files to the Distribution directory.

While the distro.json file defines the modules that are included in SuiteCommerce Advanced, the ns.package.json file determines which files within an individual module are included. Each module included in the distro.json must have a corresponding ns.package.json file. This file must be at the top level of the module folder. The ns.package.json file uses the following structure:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ],
    "templates": [
      "Templates/*.txt"
    ],
    "ssp-libraries": [
      "SuiteScript/*.js"
    ],
    "services.new": [
      "SuiteScript/*.Service.ss"
    ]
  }
}
```

```

        ]
    }
}
```

In the above example, there is a mapping between file types and partial paths to the location of the files. In general, Gulp.js uses wildcards to specify the contents of a directory. For example, "JavaScript/*.js" includes all files with the ".js" extension that are in the JavaScript folder. However, when creating your own modules, you can point to specific files within a directory, for example:

```

...
"javascript": [
    "JavaScript/MyJavaScriptFile.js"
]
...
```

The types shown in the above examples (javascript, templates, services, etc.) correspond to a specific Gulp task. Gulp tasks determine how certain file types are handled. For example, the javascript Gulp task causes all JavaScript files within a distribution to be compiled into a single file. Other Gulp tasks are responsible for copying files from the Modules directory to the Distribution directory.

NetSuite recommends that you use only the default Gulp tasks when creating or editing a module. To use a custom file type within ns.package.json, you must create your own custom Gulp task.

Gulp tasks are stored in the following location:

`SCA/gulp/tasks`



Note: This directory is created when you install Gulp.js. It is not included in the downloaded version of the SuiteCommerce Advanced source directory. See [Developer Environment](#) for more information.

How Files Are Combined

The SuiteCommerce Advanced source code is divided into multiple modules. These modules contain multiple subdirectories which, in turn, contain individual files, including JavaScript and Sass files. The contents of each of these files contains a logical structure where each element is separated by white space.

While this organization makes the source files easy to read and understand, it is inefficient when passing these files across a network to a web browser. Most modern web applications use some method of combining multipl files spread across multiple directories into a smaller number of combined files. These files are more efficient for a web browser to load because of the fewer number of requests required and smaller total number of bytes transferred.

To combine these files, SuiteCommerce Advanced uses Gulp.js to perform the following tasks:

- Determine which files need to be included in the web application.
- Combine all of the JavaScript, Sass, and SSP library files into single files.
- Copy any additional resources required by the application. For example, Gulp.js copies any image files, SSP files or services.

All resources required by SuiteCommerce Advanced, including combined and copied files, are stored in the **Distribution** directory. This directory is deployed to NetSuite. See [Contents of the DeployDistribution and LocalDistribution Directories](#) for information on the output of the gulp tasks.

Contents of the DeployDistribution and LocalDistribution Directories

- **gulp deploy**: creates a directory called DeployDistribution. This directory contains the combined source files that are deployed to NetSuite.
- **gulp local**: creates a directory called LocalDistribution. This directory contains files that are used by the local server.

The files in the output directories are automatically generated. You should not directly edit any of the files in the DeployDistribution or LocalDistribution directories. However, you may need to view the contents of these directories when troubleshooting.

Both the DeployDistribution and LocalDistribution contain the following subdirectories and file:

- **css** : contains the CSS style sheets for each application. The gulp tasks generate these files by combining all of the Sass files of the application modules.
- **css_ie**: contains the CSS style sheets for each application that are specific to Internet Explorer.
- **font-awesome**: contains a font that SuiteCommerce Advanced uses to display application icons.
- **img** : contains the images used by the application. These includes all images used in menus, headers, etc. It does not include, for example, images return by the Item Search API.
- **javascript**: contains the combined and condensed JavaScript code for each application. This code is combined from all of the JavaScript code defined in the modules
- **javascript-dependencies** : contains compiled JavaScript files containing the dependencies for each application.
- **languages**: contains localized versions of each application
- **services**: contains all of the SuiteScript services used in SuiteCommerce advanced. Files are copied from the SuiteScript directory of each module to this directory.
- **cart.ssp**: contains the SSP application for the user cart. The gulp tasks copy this from the SuiteScript directory of the CheckoutApplication application module.
- **checkout-local.ssp** : contains the local SSP application for the Checkout application. The gulp tasks generate this file automatically. The local server uses this SSP application. See [SCA on a Local Server](#) for more information.
- **checkout-environment.ssp** : contains an SSP application that defines the environment properties for the Checkout application. The gulp tasks copy this from the SuiteScript directory of the CheckoutApplication application module.
- **download.ssp** : contains an SSP application used by the My Account application. The gulp tasks copy this file from the SuiteScript directory of the MyAccountApplication application module.
- **goToCart.ssp** : contains an SSP application used by the Shopping application. The gulp tasks copy this file from the SuiteScript directory of the ShoppingApplication application module.
- **logOut.ssp** : contains an SSP application used by the Shopping application. The gulp tasks copy this file from the SuiteScript directory of the ShoppingApplication application module.
- **my_account.ssp** : contains the SSP application for the My Account application. The gulp tasks copy this file from the SuiteScript directory of the MyAccountApplication application module.
- **my_account-local.ssp** : contains the local SSP application for the My Account application. The gulp tasks generate this file automatically. The local server uses this SSP application. See [SCA on a Local Server](#) for more information.
- **myaccount.environment.ssp** : contains an SSP application that defines the environment properties for the My Account application. The gulp tasks copy this from the SuiteScript directory of the MyAccountApplication application module.

- **print-statement.ssp**: contains an SSP application used by the My Account application. The gulp tasks copy this file from the SuiteScript directory of the MyAccountApplication application module.
- **shopping.environment.ssp**: contains an SSP application that defines the environment properties for the Shopping application. The gulp tasks copy this from the SuiteScript directory of the ShoppingApplication application module.
- **shopping.ssp**: contains the SSP application for the Shopping application. The gulp tasks copy this file from the SuiteScript directory of the ShoppingApplication application module.
- **shopping.user.environment.ssp**: contains an SSP application that defines the user-specific environment properties for the Shopping application. The gulp tasks copy this from the SuiteScript directory of the ShoppingApplication application module.
- **shopping-local.ssp**: contains the local SSP application for the Shopping application. The gulp tasks generate this file automatically. The local server uses this SSP application. See [SCA on a Local Server](#) for more information.
- **ssp_libraries.js**: contains combined and condensed SSP library containing all of the backend modules used by SuiteCommerce Advanced. For more information, see [The ssp_libraries.js File](#).
- **version**: contains the time stamp when the gulp task was run.

The **ssp_libraries.js** File

The **ssp_libraries.js** file contains the server-side JavaScript code used by SuiteCommerce Advanced. When you deploy SuiteCommerce Advanced to NetSuite, the gulp task installs this file in the SSP Application.

When compiling the application, the gulp tasks generate the **ssp_libraries.js** file by combining all of the following:

- Backend models — includes all of the backend models defined in the application modules.
- SspLibraries module — contains JavaScript files that provide server-side methods and utilities used by the backend models. This module includes the following files:
 - Application.js — defines functions for interacting with SuiteScript and the Commerce API. These functions obtain context and environment information from NetSuite. This file also provides methods for sending HTTP responses and errors and methods for returning paginated results.
 - Configuration.js — defines backend configuration for SuiteCommerce Advanced. For more information, see [Backend Configuration](#).
 - Console.js — creates the server-side console used to access to the SSP application.
 - Events.js — defines the core utilities and base classes to create high-level back-end entity models used by services.
 - Models.Init.js — defines global variables available to all backend models.
 - SC.Models.js — defines the base class used by backend models.
 - Utils.js — defines global utility methods that perform server-side tasks, including searching records and formatting currencies.

Changing Your Connection Information

When deploying to NetSuite using the **gulp deploy** command, Gulp.js uses the **.nsdeploy** file to determine how to connect to NetSuite. When you run **gulp deploy**, the script automatically configures the **.nsdeploy** file based on the information you provide when running **gulp deploy**. However, if there are changes to your account information or target folder, you may need to edit this file.

To customize the **.nsdeploy** file:

1. Open the **.nsdeploy** file in a text editor.

2. Edit the following parameters to reflect your account and system information:

- **email**: the email address of the account where you want to deploy.
- **account**: the account number where you want to deploy.
- **role**: the role of the user for this account.
- **hostname**: the hostname where your account is located.
- **target_folder**: the ID of the SSP application folder.

Customize and Extend Core SuiteCommerce Advanced Modules

 **Applies to:** SuiteCommerce Advanced

 [View a Related Video](#)

SuiteCommerce Advanced (SCA) provides a fully functional application that you can use to implement your ecommerce solutions. The Kilimanjaro release of SCA and earlier were designed to give you access to all SCA source code and to let you customize the application to fit your specific business. The source code for the SCA application is organized into modules. You can make changes to existing modules or add new modules to modify or enhance the functionality of a SCA web store. You can change everything from the interface of your site to adding your own custom logic and functionality.

This section outlines the process for using the core SCA developer tools. Follow the procedures outlined in this section if:

- You are developing a SCA site using the Kilimanjaro release or earlier.
- You are developing a SCA site using the Aconcagua release or later and you are not using extensions to interact with the Extensibility API.



Important: If you are implementing the Aconcagua Release of SCA or later, the best practice is to use themes and extensions to customize your site. However, if your customizations require access to objects not available using the Extensibility API, use the procedures outlined in this section to extend code using custom modules.



Important: To make changes to SCA, you must have experience working with JavaScript, HTML, and CSS. The level of experience required depends on the types of changes you want to make. The core functionality of SCA can be augmented or modified, but advanced JavaScript programming skills, including knowledge of Backbone.js and jQuery, are required.

Before making changes to SCA, you must read and understand [Best Practices for Customizing SuiteCommerce Advanced](#). When reviewing these best practices, it is important to understand the following concepts:

- **Customize** – refers to all changes you make to SCA. Customizations can include new features and modules that you create, extensions to JavaScript functionality, and overrides to JSON or template files.
- **Extend** – refers to customizations you make to JavaScript code or style sheets. In the case of JavaScript, this refers to changes that alter or enhance the behavior of properties and methods. This involves using the JavaScript prototype of an object or using a helper method like Backbone.extend.
- **Override** – refers to changes where you replace the functionality of an entire property, method, or file with your own custom version. In some contexts, customizing a template, override is the only option.

In general, however, you should not use this process when making changes to JavaScript source files or style sheets.

- **Configuration Modification** – refers to changes you make to the SuiteCommerce Configuration record user interface by modifying the configurationManifest.json file. This involves creating custom JSON configuration modification files. This method requires the Vinson release of SCA or later.

Before customizing modules, read and understand the following topics:

- [Core SuiteCommerce Advanced Developer Tools](#) – describes how to setup your local development environment.
- [Architecture](#) – describes how source code and modules are structured.
- [Item Search API](#) – describes how to construct queries for product search results.
- [Configuration File Types](#) – describes how JSON configuration files impact the SuiteCommerce Configuration record user interface.

Best Practices for Customizing SuiteCommerce Advanced

NetSuite recommends following these best practices when customizing or extending SuiteCommerce Advanced (SCA). These best practices are designed to provide an easier path for migration when you have changed or added features to the application. Following these best practices enables you to install new versions of a bundle more quickly to take advantage of new features and issue fixes.

Different types of files and resources in SCA have different recommended best practices. These are described in the following sections:

- [Organize Source Code for Custom Modules](#)
- [Extend JavaScript](#)
- [Customize the Configuration Manifest](#)
- [Extend JavaScript Configuration Files](#)
- [Override Template Files](#)
- [Extend Style Sheets](#)
- [Override Language Files, Images, and Resource Files](#)

Organize Source Code for Custom Modules

Before making changes to SCA, you must decide on the conventions you want to use to organize your custom modules. NetSuite recommends placing all of your customizations in a directory structure that is outside of the Modules/suite_commerce directory structure. For example, you may use a directory structure similar to the following:

```
SuiteCommerce Advanced
...
Modules
  extensions
  suitecommerce
  third_parties
...
```

In this example hierarchy, your custom modules are stored in the **extensions** directory. Within your custom directory, name your modules using a version naming convention such as:

`module_name@x.y.z`

This convention corresponds to the SEMVER (Semantic Versioning) schema. NetSuite recommends that you use this schema when organizing your modules. For example, if you were creating a new module, your module may be named: **MyCustomModule@1.0.0**

This naming convention should also apply when extending an existing module. For example, if you are customizing the Case module, your custom module may be named: **CaseExtension@1.0.0**.

See <http://semver.org> for more information on the SEMVER schema.

Another consideration when determining how to organize your customization source code is to determine how to organize template overrides. As mentioned in [Override Template Files](#), you must override the entire template file, and each template file can only be overridden once. However, the view that uses the template file may be extended in multiple modules. Therefore, you may want to have a convention where template overrides are always performed in their own custom module. The module containing the template override could then be used by other modules that extend the functionality of the view.

Extend JavaScript

When customizing a property, object, or method within JavaScript, you should use the JavaScript prototype object. By calling the JavaScript **prototype** function, you can extend a specific property, object, or method without having to extend or override an entire model or view.

The primary advantage of using the JavaScript prototype method is that it improves the chances that your customizations continue to work when migrating to a newer version of SCA. When using JavaScript prototypes, the original version of the method or property, is still present along with your customizations. If a new version of SCA contains any issues fixes or enhanced features that are related to that method or property, both the new functionality and your changes are still in place.

See [Add a Child View to a Composite View](#) for an example of how to implement prototyping.

For best results, do not use the override feature of the developer tools when customizing JavaScript files. When overriding a file, all the functionality of a method or property is replaced. When a new version of SCA is installed, any fixes and enhancements are overridden. This can cause problems in both the core SCA functionality and in your customizations.

Customize the Configuration Manifest

This section applies to **Vinson** implementations of SCA and later. For details on how to configure properties using pre-Vinson release of SCA or later, see [Extend JavaScript Configuration Files](#).

SCA uses the configurationManifest.json to build the SuiteCommerce Configuration record's user interface and display configurable properties. You can customize the metadata stored in the individual JSON files used to generate this manifest. For example, you can create custom modules to introduce new properties to the manifest or customize existing JSON files to change default values, modify the appearance or location of a property in the user interface, or change property options, etc.



Important: Never alter or delete the configurationManifest.json file. To make changes to property metadata, customize the individual JSON files only. When you deploy your site using the developer tools, these individual files and customizations concatenate into and update the configurationManifest.json file. For detailed information on how JSON configuration files affect the configurationManifest.json and the SuiteCommerce Configuration record, see [Configure Properties](#).

You have two options for customizing the configurationManifest.json file:

- Create a new property using a custom module
- Customize existing property metadata

Create a New Property

When creating a new property, you must create a custom module and include your changes in JSON format using the JSON Schema V4. You use this custom file to create new properties and any metadata determining their location and behavior in the SuiteCommerce Configuration record.

The following steps are required to create a new property:

1. Create a custom module and any required files and subdirectories.
2. Use JSON Schema V4 to declare any new properties and metadata to appear in the manifest.
3. Deploy your changes to NetSuite.

See [Create JSON Configuration Files](#) for details.

Customize Existing Properties

To customize existing properties and their associated metadata, you must customize the individual source JSON configuration files using the Configuration Modification method. This allows any module to push changes to properties defined in the Configuration subdirectory of other modules. You can use this method to modify any existing JSON configuration file and add, replace, or remove metadata associated with any existing property. You can do so over any elements within in one JSON configuration file or organize changes across multiple custom modules. See [Modify JSON Configuration Files](#) for specific information and examples of the Configuration Modification method.

Extend JavaScript Configuration Files

This section applies to **pre-Vinson** implementations of SCA only. For details on how to configure properties using Vinson release of SCA or later, see [Configure Properties](#).

SCA uses multiple configuration files to define configuration properties in pre-Vinson implementations of SCA. These properties are defined within JavaScript files as objects and properties. See [Configure Properties](#) for specific information on the configuration differences between implementations and the properties they define.

In general, you must modify the configuration files to change configuration properties of a module or to include new custom module dependencies within the application. When customizing configuration files, you must create a custom module that redefines the configuration properties you want to change. However, unlike other custom modules, you do not need to use the JavaScript prototype method to extend configuration properties. In general, the following steps are required to customize configuration files:

1. Create a custom module.
2. Include the JSON configuration file as a dependency.
3. Redefine the configuration properties.

See [Extend Frontend Configuration Files](#) and [Extend the Backend Configuration File](#) for more specific information on how to perform these steps.

In situations where a configuration file defines an object containing properties, you should only redefine the specific property you want to change. You should avoid redefining the entire object. This helps ensure that there are no conflicts between your customizations and newer versions of SCA. For example, a newer version of SuiteCommerce Advance may add additional properties that would be overwritten if you redefine the entire object.

Before customizing configuration files, you should create a convention for customizing configuration files that makes sense for your development environment. For example, you may want to create a custom module for each configuration file. In this scenario, you extend all configuration properties of the SC.MyAccount.Configuration module within a single custom module.

In some situations, you may want to define a custom module for each individual configuration property you override. In this scenario, for example, you may have a module called PaymentWizardCustomConfiguration@1.0.0 where you define an override for the payment wizard configuration object of the SC.MyAccount.Configuration module.

Override Template Files



When extending a template file, you must use the file override method provided by the developer tools. The primary reason for this requirement is that you cannot extend specific parts of a template. This means that all changes you make to a template file must be made in the same custom file that you create. See [Override a Template File](#) for an example of how to use the override method.

When making changes to a template, you must be aware of the properties and objects that are returned by the `getContext()` method of the template's view. The `getContext()` method can be seen as a contract that ensures that when migrating to a new version of SCA the same data is available to the template in the new version. For compatibility with previous versions, a new release of SCA includes the same properties and objects returned by the `getContext()` method.

New properties or objects may be included in new versions of SCA, but the existing properties and objects continue to available. This helps ensures that your template customizations work after migration because they will have access to the same data as the previous version.

If you need to expose your own properties or objects to the template, add them to the `getContext()` method by extending that method using the best practices outlined in [Extend JavaScript](#).

Extend Style Sheets

SCA uses the Sass scripting language, which is transformed into CSS when you use the developer tools to compile and deploy the application. Sass files have a built-in mechanism for extending style definitions, by using the `@extend` keyword. This keyword enables you to add the styles of one selector to another one. This enables you to extend only a specific part of a style definition. See [Design Hierarchy](#) for information on working with Sass in SCA.

To extend a Sass file, you should create a custom module containing a Sass file. This file should only contain the style elements you want to extend as well as any new style elements you define.

Nesting

Although not a design pattern, keep the following in mind when reading and customizing Sass. SCA uses the ampersand symbol (&) as a nesting technique for classes. This can be handy to simplify your Sass.

For example, use the following Sass:

```
.my-class {
  &.my-other-class {}
}
```

This results in the following CSS:

```
.my-class.my-other-class {}
```

Developing Sass in a Local Environment

When you run the `gulp local` command, the Sass preprocessor compiles the CSS for your local site, certain naming conventions affect the various application style sheet files (shopping.css, myaccount.css,

and checkout.css). Using the source map, the browser understands and relates your classes to the DOM with the classes defined at the file level during development. This means that, when using the browser developer tools in a local environment, you can easily search for classes the same way you would at the file level during development.

Override Language Files, Images, and Resource Files

When extending the following resources, you must use the file override method provided by the developer tools:

- Languages files
- Images
- Resource files

The primary reason for this requirement is that these resources can only be customized at the file level. Each of these resources can only be overridden once. Therefore, make all changes to these resources within the same file.

Customization Examples

This topic provides basic examples for extending SuiteCommerce Advanced (SCA) and creating your own custom modules.



Important: If you are developing the Aconcagua release of SCA or later, the best practice is to implement your changes as themes and extensions. See [Customization](#) for more details.

Create a Custom Module

SuiteCommerce Advanced (SCA) is designed so that you can extend it by creating new modules to add functionality specific to your web store. When creating a new module, be aware of the following requirements:

- The top level directory of your module should have a directory of the format: `module_name@x.y.z` where `module_name` is the name of the module and `x.y.z` is the version number. See [Organize Source Code for Custom Modules](#)
- The directory containing your custom module must be in the Modules directory. Modules shipped with SCA are stored in two sub-directories, `suitecommerce` and `third_parties`. NetSuite recommends creating a third sub-directory to store all custom module code. Examples in this document name this directory `extension`, as in `Modules/extension/`. However, you can name this directory any intuitive name as required.
- Your new modules directory must have an `ns.package.json` file to define all of the module dependencies required by the developer tools. For more information, see [Core SuiteCommerce Advanced Developer Tools](#) for more information.
- The module name must be unique. You cannot have duplicate module names, even when they reside in different folders.
- You must update the `distro.json` file to ensure that your custom module is loaded into the application.

To add a custom module:

1. Create a custom module within your custom directory with the format of `ModuleName@version`.

For example, Modules/extensions/MyCustomModule@1.0.0.



Important: You cannot use existing module names, even when those modules reside in different folders.

2. Create the subdirectories within the module.

The exact subdirectory structure depends on the requirements of your module and your implementation of SCA. NetSuite recommends that you use the same directory structure as existing application modules. Typically, you will need JavaScript, Sass, SuiteScript, and Templates subdirectories.

3. Create the necessary files for your module.

This is where all of your modules logic is contained. At a minimum, you will need something like the following:



Note: This procedure does not include examples of backend models or services files which are often necessary for backend integration with NetSuite.

- **An entry point JavaScript file** – this file acts as an entry point to the module and must return an object containing the `mountToApp` property that receives the application as a parameter.

```
//MyNewModule.js
define('MyNewModule'
,
  [
    'MyNewModule.Router'
  ]
,
  function (
    Router
  )
{
  'use strict';

  return {
    mountToApp: function (application)
    {
      // Initializes the router
      return new Router(application);
    }
  };
});
```

- **A router** – this JavaScript file extends the Backbone router to map URL patterns to the views defined within the module.

```
//MyNewModule.Router.js
define('MyNewModule.Router'
,
  [
    'MyNewModule.View'
    , 'Backbone'
  ]
,
  function (
    MyNewModuleView
    , Backbone
  )
{
```

```
'use strict';

//@class Address.Router @extend Backbone.Router
return Backbone.Router.extend({

    routes: {
        'my-new-module': 'MyNewModuleRouter'
    }

    , initialize: function (application)
    {
        this.application = application;
    }

    // list myNewRouter output
    , MyNewModuleRouter: function ()
    {
        var view = new MyNewModuleView({
            application: this.application
        })
        view.showContent();
    }
});

});
```

- **A view** – this view is called from the router. Often, a view contains a `getContext` function that describes all of the variables that are passed from the view to the template.

```
//MyNewModule.View.js
define(
    'MyNewModule.View'
    , [
        'my_new_module.tpl'
        , 'Backbone'
        , 'jQuery'
    ]
    , function (
        MyNewModuleTemplate
        , Backbone
        , jQuery
    )
{
    'use strict';

    //@class Address.List.View List profile's addresses @extend Backbone.View
    return Backbone.View.extend({

        template: MyNewModuleTemplate

        , events: {
            'click [data-action="test"]': 'testAction'
        }

        , testAction: function ()
        {

```

```

        alert("This is a test action")
    }

    , getContext: function ()
    {
        return {
            //@property {String} myNewModuleContextVar
            myNewModuleContextVar: 'myVariable'
        };
    }
);
});

```

- **A template file** – this file contains the markup used in combination with the views. Note that the SCA implementation leverages the Handlebars templating engine.

```
<h2>This is my template file for my new module. It takes variables, {{myNewModuleContextVar}}, passed from the View.</h2>
```



Important: Template file names must be unique. You cannot have duplicate template file names even when they belong to different modules.

4. If your custom module contains any configurable properties, you must include a Configuration subdirectory and place your Configuration Modifications here. See [Modify JSON Configuration Files](#) for details.



Note: This step only applies to **Vinson** implementations of SCA and later.

5. Create a new **ns.package.json** file within the Modules directory.

This file must contain entries for all of the directories required by the module. Use the following example as a template:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ],
    "templates": [
      "Templates/*.tpl"
    ],
    "ssp-libraries": [
      "SuiteScript/*.js"
    ],
    "services.new": [
      "SuiteScript/*.Service.ss"
    ],
    "sass": [
      "Sass/**/*.scss"
    ]
  }
}
```

6. Update the distro.json file in the root directory of the SCA source code.

Update this file with the following two changes:

- Add an entry for the new module in the `modules` object:

```
{
  "name": "SuiteCommerce Advanced 1.0.0",
  "version": "1.0.0",
  "modules": {
    "suitecommerce/Account": "1.0.0",
    "extensions/myNewModule": "1.0.0",
    "suitecommerce/AjaxRequestsKiller": "1.0.0",
    "suitecommerce/ApplicationSkeleton": "1.0.0",
    ...
  }
}
```

- Define any application dependencies in the `javascript` object.

For example, if the module is required in the Shop Flow application, add the module to the `SC.Shopping.Starter` **entrypoint**.

```
"javascript": [
  {
    "entryPoint": "SC.Shopping.Starter",
    "exportFile": "shopping.js",
    "dependencies": [
      "Backbone.View.Plugins",
      "jQuery.html",
      "ItemDetails",
      "myNewModule",
      ...
      "UrlHelper",
      "CMSAdapter"
    ],
  },
]
```

7. View your changes.

After creating your new module, you can test it by viewing your changes in the application. If you are running a local server, you can view your changes by reloading your website. See [SCA on a Local Server](#) for more information.

If you are viewing your site in NetSuite, you can deploy your changes using the developer tools. See [Deploy to NetSuite](#) for more information.

Modify JSON Configuration Files

ⓘ Applies to: Vinson and later

This section applies to **Vinson** implementations for SuiteCommerce Advanced (SCA) and later. For details on configuring SCA for pre-Vinson release and earlier, see [Extend JavaScript Configuration Files](#).



Important: Never alter or delete the `configurationManifest.json` file. To make changes to property metadata, customize the individual JSON files only. When you deploy your site using the developer tools, these individual files and customizations concatenate into and update the `configurationManifest.json` file. For detailed information on how JSON configuration files affect the `configurationManifest.json` and the SuiteCommerce Configuration record, see [Configure Properties](#).

This section explains how to use the Configuration Modification method to change existing configurable properties and associated metadata. When you deploy this modification to NetSuite, the SuiteCommerce

Configuration record's user interface will reflect any changes specified. Configuration Modification requires knowledge of JSONPath query schema, see <https://github.com/s3u/JSONPath> for more information.



Important: Making changes to core source files or changing any vital functionality of the application can make migrating to future releases difficult. Before making changes to SCA, see [Best Practices for Customizing SuiteCommerce Advanced](#).

To modify the SuiteCommerce Configuration record's user interface:

1. Create the directory structure for a custom module.

In this example, the name of the module being customized is the RecentlyViewedItems module. Following best practices, the example custom directory is titled RecentlyViewedItemsExtension@1.0.0.

2. Create a Configuration subdirectory in your new module directory.
3. Create a new JSON file in the Configuration subdirectory. In this example, the name of the JSON file being customized is SearchResultsPerPage.json. Following best practices, the custom JSON file is titled SearchResultsPerPageModification.json.
4. Add modification code to your custom JSON file.

See [Configuration Modification Schema](#) and [Use Case Examples](#) for specific information and examples on how to structure this file.

5. Create and edit an ns.package.json file in the root directory of your custom module.

Add the following code to this file:

```
{
  "gulp": {
    "configuration": [
      "Configuration/*.json"
    ]
  }
}
```

6. Update the distro.json file in the root directory of the SCA source directory.

Add the name of your custom module to the list of modules defined in the `modules` object.

```
...
"modules": {
  "extensions/RecentlyViewedItemsExtension": "1.0.0",
  "suitecommerce/Account": "2.2.0",
  ...
}
```

7. Deploy your changes to NetSuite and access the SuiteCommerce Configuration record in NetSuite to view your changes. See [Deploy to NetSuite](#) for details



Note: You must deploy your customizations to NetSuite using the developer tools to apply any modifications to the SuiteCommerce Configuration record. You can confirm changes to the configurationManifest.json code on a local server, but modifications will not take effect until you deploy to NetSuite.

Extend Frontend Configuration Files

Applies to: Denali | Mont Blanc

This section applies to **pre-Vinson** implementations for SuiteCommerce Advanced (SCA) only. For details on configuring SCA for Vinson release or later, see [Modify JSON Configuration Files](#).



Important: Making changes to core source files or changing any vital functionality of the application can make migrating to future releases difficult. Before making changes to SCA, see [Best Practices for Customizing SuiteCommerce Advanced](#).

SCA enables you to configure the behavior of the frontend application by modifying configuration properties and objects. These properties are contained in configuration files that define objects that are loaded when the application starts. These properties are accessible to all modules within the application.

See [Configure Properties](#) for more information about these files and the properties they contain.

To redefine these properties for your installation you must create a custom module that includes the original configuration file as a dependency.

To extend a frontend configuration file:

1. Create the directory structure for your custom module.

In this example, the name of the custom module is Configuration, so the module directory would be Configurator@1.0.0.

2. Create the JavaScript subdirectory in your new module directory.
3. Create a new JavaScript file in the JavaScript directory of your custom module.

In this example, the name of the JavaScript file is Configurator.js

4. Add the `mountToApp` method to your custom JavaScript file.

For example, the code you add should look similar to the following:

```
define('Configurator'
  [
    'SC.Configuration'
  ]
  function (
    Configuration
  )
{
  'use strict';
  return {
    mountToApp: function ()
    {
      //Add your custom properties here.
    }
}
```

```
    };
});
```

This code performs the following tasks:

- Lists the dependencies required. When customizing a configuration file, you must include the object it returns as a dependency.
 - Defines the `mountToApp` method. This method is required to load your custom module into the application. This method also contains the custom properties you are configuring.
5. Add your custom properties to the block within the `mountToApp` method as shown in the example above.

The following example shows how to redefine different configuration properties:

```
Configuration.imageNotAvailable = 'http://www.tnstate.edu/sociology/images/Image%20Not%20Available.jpg';

Configuration.addToCartBehavior = 'goToCart';

Configuration.facetDelimiters.betweenFacetNameAndValue = '%';

Configuration.productReviews.loginRequired = true;

Configuration.typeahead.maxResults = 4;
```

6. Create and edit the `ns.package.json` in the root directory of your custom module.

For example, the code you add should look similar to the following:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

7. Update the `distro.json` file in the root directory of the SCA source directory. You must update this file in two places:

- Add the name of your custom module to the list of modules defined in the `modules` object.
- Add the name of your custom module to the `dependencies` array of the application whose configuration file you are customizing. To ensure that your customized configuration properties are available to all modules, place your module at the top of this list.

8. View your changes.

If you are running a local server, you can view your changes by reloading your website. See [SCA on a Local Server](#) for more information.

If you are viewing your site in NetSuite, you can deploy your changes using the developer tools. See [Deploy to NetSuite](#) for more information.

Extend the Backend Configuration File

 **Applies to:** Denali | Mont Blanc

This section applies to **pre-Vinson** implementations for SuiteCommerce Advanced (SCA) only. For details on configuring SCA for Vinson release or later, see [Configure Properties](#).



Important: Making changes to core JavaScript source files or changing any vital functionality of the application can make migrating to future releases difficult. Before making changes to SCA, see [Best Practices for Customizing SuiteCommerce Advanced](#).

SCA uses a backend configuration file to modify the behavior of NetSuite features. See [Backend Configuration](#) for more information.

During deployment, the backend configuration file is combined and deployed as a SuiteScript library in NetSuite. Like other JavaScript customizations, the recommended best practice is to create a custom module that redefines the configuration properties that you want to modify. However, since this file is deployed as a SuiteScript library, you only add your custom module to the distro.json file as a dependency. You do not need to define the `mountToApp` method.

To extend the backend configuration file:

1. Create the directory structure for your custom module.

In this example, the name of the custom module is BackendConfigurator, so the module directory would be `BackendConfigurator@1.0.0`.

2. Create the SuiteScript subdirectory in your new module directory.

Since the backend configuration file is a SuiteScript file, this directory is required.

3. Create a new JavaScript file in the SuiteScript directory of your custom module.

In this example, this file is `BackendConfigurator.js`

4. Add code to redefine the backend configuration properties you want to change.

For example, the code you add should look similar to the following:

```
define('BackendConfigurator'
  , [
    'Configuration'
  ]
  , function (
    config
  )
{
  'use strict';
  //disable CMS
  config.useCMS = false;
});
```

This example performs the following:

- Lists the dependencies required by the custom module. When customizing the backend configuration file, the only required dependency is the `Configuration` object. This object is defined in the `Configuration.js` file located in the `Modules/Ssplibraries` directory of the SCA source directory.
- Redefines a single configuration property. In this case, the custom module is setting the `useCMS` property to `false`.

5. Create and edit the `ns.package.json` in the root directory of your custom module.

Your file should contain code similar to the following:

```
{  
  "gulp": {
```

```

    "ssp-libraries": [
      "SuiteScript/*.js"
    ]
}

```

6. Edit the distro.json file.

You must add your custom module to the distro.json file for it to be loaded into the application. Since this module is extending an SSP library, you must add it to the following locations:

- Add the name of your custom module to the **modules** array. To ensure that your customizations are not overwritten by other modules, add the custom module to the top of the array.
- Add the name of your custom module to the **dependencies** array of the **ssp-libraries** object.

7. View your changes.

Since the backend configuration file is a SuiteScript file and stored as an SSP library, you must deploy your custom module directly to NetSuite to view your changes. See [Deploy to NetSuite](#) for more information.

Add a Child View to a Composite View



Important: Making changes to core JavaScript source files or changing any vital functionality of the application can make migrating to future releases difficult. Before making changes to SuiteCommerce Advanced (SCA), see [Best Practices for Customizing SuiteCommerce Advanced](#).

Adding a child view to a parent view is a common way of extending the functionality of SCA. For example, you can easily add a message to a page by adding a `GlobalViews.Message.View` view as a child view. Adding a child view requires making two types of changes to the SCA source code:

- Extend the **childViews** object of a view. Since this requires a change to the JavaScript, you should create a custom module that uses JavaScript prototyping to add the child view.
- Override the template file. In addition to adding the view to the **childViews** object, you must also edit the HTML template to implement the child view in a page.



Note: Since a template file can only be overridden once, you may want to define the override in a different custom module created specifically for template overrides when making your own customizations. See [Override a Template File](#) for more information.

To extend the `childViews` object of a view:

1. Create the directory structure for your custom module.
 - a. Create a directory called **extensions** within the **Modules** directory.
 - b. In the **extensions** directory, create a subdirectory called **HeaderExtension@1.0.0**.
 - c. In the **HeaderExtension** directory, create a subdirectory called **JavaScript**.
 - d. Also in the **HeaderExtension** directory, create another subdirectory called **Templates**.

In general, when performing your own customizations you should create a directory structure similar to the above procedure. See [Organize Source Code for Custom Modules](#) for more information.

2. Create a new JavaScript file called HeaderExtension.js.
3. Define your custom module and dependencies by adding the following code to this file.

```
define('HeaderExtension'
  [
    'underscore',
    'Header.View',
    'GlobalViews.Message.View'
  ]
, function (
  -
, HeaderView
, GlobalViewsMessageView
)
{
  'use strict';

  //Additional code goes here.

});
```

This code defines the dependencies required by your custom module. See [Asynchronous Module Definitions \(AMD\) and RequireJS](#) for information on defining dependencies within a module. This module includes the following views as dependencies:

- Header.View – is required to extend the **childViews** object.
 - GlobalViews.Message.View – is required to add a message view to the application header.
4. Add the **mountToApp** method to Header.Extension.js (or Header.Extension.ts) as shown in the following:

```
return {

  mountToApp: function (application)
{

  HeaderView.prototype.childViews.HeaderExtension = function()
  {
    return new GlobalViewsMessageView({
      message: 'Hello World! - This is an Example of a GlobalMessageView!'
        , type: 'success'
        , closable: true
    });
  }
}
```

This code performs the following:

- Specifies a **return** statement that returns the **mountToApp** method. This method is required to load a module into the application.
- Extends the **childViews** object of the Header.View module using JavaScript prototyping.

5. Copy the original template file to your custom module.
 - a. Copy the header.tpl file from the Modules/suite_commerce/Header/Templates directory to the Templates directory of your custom module.

- b. Edit the custom template file by adding the following HTML code:

```
<div data-view="HeaderExtension"></div>
```

Add the HTML code at a place in the template where it will be displayed in the Header view. For example, you can add it directly above the `<div class="header-menu-cart">` tag.

6. Create the ns.package.json file

- Create a file called ns.package.json in the `HeaderExtension` directory.
- Add the following to the ns.package.json file:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*"
    ],
    "templates": [
      "Templates/*"
    ]
  },
  "overrides": {
    "suitecommerce/Header@1.1.0/Templates/header.tpl": "Templates/header.tpl"
  }
}
```

7. Add an entry for your module to the distro.json file located in the root directory of the SCA source code.

You must add your custom view to the `javascript` object within the distro.json file. Your code should look similar to the following:

```
"javascript": [
{
  "entryPoint": "SC.Shopping.Starter",
  "exportFile": "shopping.js",
  "dependencies": [
    "Header.Extension.View",
    "Backbone.View.Plugins",
    "jQuery.html",
    "ItemDetails",
    ...
  ]
}]
```

In this example, we are only customizing a single file, so we only add the Header.Extension module to the `javascript` object. In cases where you are customizing or overriding an entire application module, you may need to add the application module name here.

8. View your changes.

If you are running a local server, you can view your changes by reloading your website. See [SCA on a Local Server](#) for more information.

If you are viewing your site in NetSuite, you can deploy your changes using the developer tools. See [Deploy to NetSuite](#) for more information.

Override a Template File



Important: Making changes to core JavaScript source files or changing any vital functionality of the application can make migrating to future releases difficult. Before making changes to SuiteCommerce Advanced (SCA), see [Best Practices for Customizing SuiteCommerce Advanced](#).

When customizing SCA, you may need to change the HTML code of the application. To add, remove, or change the HTML, you must customize the template file for the module that corresponds to the feature you want to change. To customize a template, you must use the override method provided by the developer tools. Since there is no mechanism for overriding or extending a specific part of a template, you must override the entire file.

This example describes how to override a template file to add additional text to the product description page. You can use this example as a guide when customizing your own templates.

To override a template:

1. Create the directory structure to store your customized template.
 - a. Create a directory called **extensions** within the **Modules** directory.
 - b. In the **extensions** directory, create a subdirectory called **ItemDetailsExtension@1.0.0**.
When creating a new custom module, the module name must be unique. You must not have duplicate module names, even if those modules reside in different folders.
 - c. In the **ItemDetailsExtension@1.0.0** directory, create a subdirectory called **Templates**.
 - d. If required, create another subdirectory called **Sass**.

In general, when creating custom modules, you should create a directory structure similar to that described in the procedure above. See [Organize Source Code for Custom Modules](#) for more information.

2. Copy the original template file to your custom directory.

Copy the **item_detail.tpl** file from **Modules/suite_commerce/ItemDetails@x.y.z/Templates** to the **Templates** directory of your custom application module. The string **x.y.z** corresponds to the version of ItemDetails module in your version of SCA.

3. Create the **ns.package.json** file.

You must create this file in the **ItemDetailsExtension@1.0.0** directory.

4. Add the required code to the **ns.package.json** file.

The contents of your **ns.package.json** file should be similar to the following:

```
{
  "gulp": {
    "templates": [
      "Templates/**"
    ],
    "sass": [
      "Sass/**/*.scss"
    ],
    "overrides": {
      "suitecommerce/ItemDetails@x.y.z/Templates/item_details.tpl": "Templates/item_details.tpl"
    }
  }
}
```



Note: You must replace the string `x.y.z` in the above example with the version of the ItemDetails module in your version of SCA.

The first section of this example defines the required objects used by gulp to include your customized template and Sass files when combining the application.

The second section contains the override directive which maps the original file in the SCA source distribution directory to your custom template file. In this example, this mapping tells the gulp.js combiner to override a specific file.

5. Edit your custom template.

This procedure shows the basic steps required to modify a template. However, you can fully customize your templates as required. See [Design Architecture](#) for more information.

1. Open the `item_details.tpl` file you copied above.
 2. In your editor, search for the string **Add to Cart**.
- This string defines the label for the add to cart button.
3. Replace this string with the string 'Add to Your Cart.'

6. Create a custom Sass file, if required.

See [Extend a Sass File](#) for more information.

7. Update the `distro.json` file.

You must add your custom module to the `distro.json` file to ensure that the gulp tasks include your custom template. If you are customizing Sass files, you must also add a reference to them in the appropriate location in the `distro.json` file. See [Extend a Sass File](#) for more information.

- Add an entry for the new module in the list of modules in the `modules` object as shown below:

```
{
  "name": "SuiteCommerce Advanced Mont Blanc",
  "version": "2.0",
  "buildToolsVersion": "1.1.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/ItemDetailsExtension": "1.0.0",
    "suitecommerce/Account": "2.1.0",
    ...
  }
}
```

In this example, the ItemDetailsExtension module is added at the beginning of the list of modules. However, you can add the module anywhere in the `modules` object. If a module contains only customized template or Sass files, the order of precedence in this list does not matter.

8. View your changes.

If you are running a local server, you can view your changes by reloading your website. See [SCA on a Local Server](#) for more information.

If you are viewing your site in NetSuite, you can deploy your changes using the developer tools. See [Deploy to NetSuite](#) for more information.

When viewing your changes, you should see new button label and the item details and item image should be reversed.

Extend a Sass File



Important: Making changes to core JavaScript source files or changing any vital functionality of the application can make migrating to future releases difficult. Before making changes to SuiteCommerce Advanced (SCA), see [Best Practices for Customizing SuiteCommerce Advanced](#).

SCA enables you to customize your web store to easily apply global style changes while supporting migration to later versions. When customizing styles, you create a custom module that overrides specific Sass variables already defined in the BaseSassStyles module. You then redefine application dependencies so that base Sass styles import your customizations in the correct order.

This example shows how to create a custom module to change the default background and foreground colors.

To extend a Sass file:

1. Create the directory structure to store your custom module.
 - a. Create a directory called **extensions** within the **Modules** directory.
 - b. In the **extensions** directory, create a subdirectory called **CustomSass@1.0.0**.
When creating a new custom module, the module name must be unique. You must not have duplicate module names, even if those modules reside in different folders.
 - c. In the **CustomSass@1.0.0** directory, create a subdirectory called **Sass**.

In general, when creating custom modules, you should create a directory structure similar to that described in the procedure above. See [Organize Source Code for Custom Modules](#) for more information.
2. Create a Sass file.
 - a. In the Sass directory, create a new file called `_custom-sass.scss`.
 - b. Add Sass variable definitions to this file.

These variable definitions override the base Sass variables defined in the BaseSassStyles module. See [Design Hierarchy](#) for more information.

```
$sc-color-primary: #0000ff; // originally #f15c28;
$sc-color-secondary: #00ff00; // originally #5B7F8C;
$sc-color-theme: #00ff00; // originally #5B7F8C;
$sc-color-link: #00ff00; // originally #2f9ac3;
$sc-color-theme-light: #00aa00; // originally #9cb6bf
$sc-color-theme-background: #000000;

$sc-color-dark-copy: #e7d13b; // originally #1f2223;
$sc-color-copy: #ede39f; // originally #404040;
body {
  background-color: $sc-color-theme-background;
}
```

3. Create the ns.package.json file.
 - a. Create a file called `ns.package.json` in the **CustomSass@1.0.0** directory.

- b. Add the following to the ns.package.json file.

```
{
  "gulp": {
    "sass": [
      "Sass/**/*.scss"
    ]
  }
}
```

4. Update the distro.json file.

- a. Add an entry for your module to the distro.json file located in the root directory of the SCA source code.

```
{
  "name": "SuiteCommerce Advanced Mont Blanc",
  "version": "2.0",
  "buildToolsVersion": "1.1.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/CustomSass": "1.0.0",
    "suitecommerce/Account": "2.1.0",
    ...
  }
}
```

- b. Split the BaseSassStyle entry in the sass object and add your custom module between them. This ensures that the Sass variables are loaded in the correct order as follows:

1. The main-variables Sass file.
2. The custom Sass variables defined in the CustomSass module.
3. Additional Sass files of the BaseSassStyles module. Including these last ensures that the additional Sass files import the custom variable definitions.

```
{
  "module": "BaseSassStyles",
  "include": [
    "main-variables"
  ],
  "CustomSass": {
    "module": "BaseSassStyles",
    "include": [
      "main-atoms",
      "main-molecules",
      "bootstrap-overrides"
    ]
  }
}
```

```
},
```

Note: You must define these dependencies for each application whose Sass variable you want to change.

5. View your changes.

If you are running a local server, you can view your changes by reloading your website. See [SCA on a Local Server](#) for more information.

If you are viewing your site in NetSuite, you can deploy your changes using the developer tools. See [Deploy to NetSuite](#) for more information.

Create a CCT Module

Applies to: Kilimanjaro

A custom content type (CCT) lets you create custom functionality as content that you can dynamically manage using Site Management Tools. Implementing CCTs for SuiteCommerce Advanced (SCA) requires two separate, but related activities. If you are implementing the Kilimanjaro release of SCA, you must first create a custom SCA module to contain the JavaScript, HTML, Sass, and resources required for your CCT. This topic explains how to accomplish this. The second activity involves setting up custom records in NetSuite for your custom content type using Site Management Tools.

See [Custom Content Type](#) for details on setting up your CCT using Site Management Tools.



Important: If you are developing SuiteCommerce or the Aconcagua release of SCA or later, you must create CCTs using extensions. See the help topic [Create a Custom Content Type](#).



Note: You must have Site Management Tools enabled in your NetSuite account to implement CCTs on your SCA site. See the help topic [Site Management Tools](#) for information on how to set up SMT.

This procedure explains how to do the following:

1. Create a Custom Module
2. Create an Entry Point JavaScript file
3. Create a View
4. Create a Template
5. Set Up Your ns.package.json and distro.json Files
6. Deploy your code to your NetSuite account

Create a Custom Module for Your CCT

Your CCT module must include the following files at a minimum:

- **An entry point JavaScript file** – this includes the `mountToApp()` method, which links your custom module to a CMS Content Type record, making it available for inclusion in your website.

- **A view JavaScript file** – this file listens for and interprets events and defines the variables for use by the template. These variables are the various fields within your CMS Content Type record.
- **An HTML template** – this file uses Handlebars.js helpers and HTML to render the content to your site.

The following procedures use a fictitious CCT named **SC.CCT.ImageViewer** as an example. This CCT introduces a simple image viewer that gets data from either a custom record (as set up in Site Management Tools) or from certain data previously made available in the browser. You can download the code samples described in these examples here: [Example ImageViewer CCT](#).



Note: You can create a custom CCT module to retrieve, create, and update data stored in NetSuite records using services and models. To do this, you must implement these files as you would any custom module. See [Module Architecture](#) for details.

To create a CCT Module:

1. Create a custom module to contain your CCT files.

As a best practice, use the following format when naming your CCT Module:

SC.CCT.cctName@x.y.z, where **cctName** is the name of your CCT and **x.y.z** is the version number.

For example:

SC.CCT.ImageViewer@0.0.1

2. In your new module, create the following subdirectories at a minimum:

- **JavaScript** – contains the entry point JavaScript file and all views.
- **Templates** – contains the HTML template that will render your CCT.



Note: If your CCT introduces any new Sass, SuiteScript, or Configuration files, add subdirectories for these files as well.

Create an Entry Point JavaScript File

The entry point file is necessary to mount your module to the application. This provides the connection between SuiteCommerce Advanced (SCA) and Site Management Tools. For more information on the architecture and purpose of this file, see [Entry Point](#).

To create the entry point:

1. In your CCT module's JavaScript directory, create a new .js file.
2. Name this file to intuitively relate to your module.

For example:

./SC.CCT.ImageViewer@0.0.1/JavaScript/SC.CCT.ImageViewer.js

3. Define your entry point dependencies.

This includes the view that you create later. For example, your entry point might look similar to the following:

```
define(
  'SC.CCT.ImageViewer'
, [
```

```

        'SC.CCT.ImageViewer.View'
    ]
    function (
        SCCCTImageViewerView
    )

```

4. Create the `mountToApp()` method.

This method is required to initialize your CCT module, making it available to the application. Wrapped inside the `mountToApp` method is `getComponent('CMS').registerContentType()`, which passes the `id` and `view` variables to SMT:

- The `id` variable loads the CMS Content Type record, connecting your module to the CCT and making your CCT available in Site Management Tools.



Important: The value of the `id` variable must match the **Name** field of the CMS Content Type record, and it must be all lowercase. See [CMS Content Type Record](#) for details.

- The `view` variable initializes the view.

In the following example, the `id` variable is '`sc_cct_imageviewer`', which matches the CMS Content Type record's **Name** field.

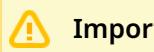
```

{
    'use strict';

    // @class SC.CCT.ImageViewer
    return {
        mountToApp: function mountToApp (application)
        {
            application.getComponent('CMS').registerContentType({
                id: 'sc_cct_imageviewer'
                , view: SCCCTImageViewerView
            });
        }
    };
}

```

5. Save the file.

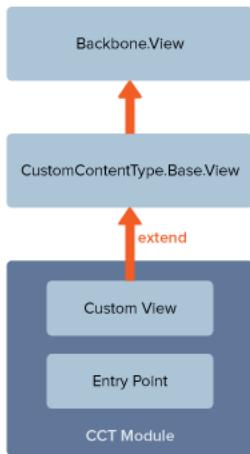


Important: Your SCA source code includes a module named `SC.CCT.Html`. This module connects your SCA application to the four core CCTs that come with the Site Management Tools SuiteApp. Do not customize or alter the contents of this module.

Create a View File

Although the entry point loads your custom module as a CCT in Site Management tools, your CCT still has no data to work with. The view file is necessary to access data, listen to and interpret user events, and specify the context for the template to render the data.

Included with your SuiteCommerce Advanced (SCA) source files, the `CustomContentType` module introduces the `CustomContentType.Base.View.js`. This file extends `BackboneView.js`, initializes the CCT settings, and includes the base CCT class from which all custom CCT modules extend. You create your custom view to extend `CustomContentType.Base.View.js`.



Accessing Data

Each view can access SCA data about an item, itemlist, product, or category (available to the DOM via certain SCA modules). Your view can also return the values for Custom Record fields linked to the CMS Content Type record.



Important: Each view requires the `getContext()` method to expose this data to your templates.

Accessing SCA Data

If your CCT requires access to data not associated with a custom record's fields, you can use the `contextDataRequest` array property to access some SCA objects. If your CCT requires access to this data, you can set up your view to consume the information that the following `contextData` objects provide, assuming the information is previously provided in the DOM at the location that you place the CCT.

contextData Object	View	Information Returned
<code>category</code>	<code>Facets.Browse.View.js</code>	Returns the data of the category you are navigating
<code>item</code>	<code>ProductDetails.Base.View.js</code>	Returns the data of the item in the product details page
<code>itemlist</code>	<code>Facets.Browse.View.js</code>	Returns the current list of items in the search page
<code>product</code>	<code>ProductDetails.Base.View.js</code>	Returns the data of the product in the product details page



Note: By default, when you add SCA content into an area using the SMT Admin, the application runs the `validateContextDataRequest` method to check that you have all the requested contexts. In some cases, the `contextData` object might request information that is optional, and the data might not exist. In these cases, the `validateContextDataRequest` method fails because the data request comes back empty. To account for this, set up your view to override the `validateContextDataRequest` method to always return `true`.

Example:

The example ImageViewer CCT requests the name of the `item` object. Your code might look similar to the following:

```
//...
```

```

, contextDataRequest: ['item']
, validateContextDataRequest: function()
{
    return true;
}
, getContext: function getContext()
{
    //...
    if (this.contextData.item)
    {
        var item = this.contextData.item();
        //...
    }
    //...
}
//...

```

Accessing Custom Record Fields

By default, the template receives a context object for each property defined in each associated Custom Record within a **settings** object. This is where you define the fields, by **Field ID**.

Example:

The example ImageViewer CCT uses a custom record with the following fields:

- **custrecord_sc_cct_iv_valign** – sets the vertical alignment of a text object.
- **custrecord_sc_cct_iv_text** – declares some text to display.
- **custrecord_sc_cct_iv_imageurl** – declares the URL to an image.

Within the **getContext()** method, you declare specific fields associated with your Custom Record as **settings**. Assuming the **custrecord_sc_cct_iv_valign** field requires a custom list of options, your code might look similar to the following.

```

//...
, getContext: function()
{
    var texts = []
    , imageUrl = ''
    , valign = this.valign[this.settings.custrecord_sc_cct_iv_valign] || this.valign['3'];

    var set_text = Utils.trim(this.settings.custrecord_sc_cct_iv_text)
    , set_texts = set_text ? set_text.split('\n') : []
    , set_imageUrl = Utils.trim(this.settings.custrecord_sc_cct_iv_imageurl);

    texts = set_texts.length ? set_texts : texts;
    imageUrl = set_imageUrl ? set_imageUrl : imageUrl;
}
//...

```

To create the View:

1. In your CCT module's JavaScript directory, create a new .js file.
2. Name this file to intuitively relate to your module as a view.

For example:

`../SC.CCT.ImageViewer@0.0.1/JavaScript/SC.CCT.ImageViewer.View.js`

3. Define the view's dependencies.

For the example ImageViewer CCT, your code might look like this:

```
define(
  'SC.CCT.ImageViewer.View'
  [
    'CustomContentType.Base.View'
    , 'SC.Configuration'

    , 'sc_cct_imageviewer.tpl'

    , 'Utils'
    , 'jQuery'
  ]
  function (
    CustomContentTypeBaseView
    , Configuration

    , sc_cct_imageviewer_tpl

    , Utils
    , jQuery
  )
{
}
```

4. Build your view according to your CCT's requirements.

5. Save the view file.

For the example ImageViewer CCT, your view might look similar to the following:

```
define(
  'SC.CCT.ImageViewer.View'
  [
    'CustomContentType.Base.View'
    , 'SC.Configuration'

    , 'sc_cct_imageviewer.tpl'

    , 'Utils'
    , 'jQuery'
  ]
  function (
    CustomContentTypeBaseView
    , Configuration

    , sc_cct_imageviewer_tpl

    , Utils
    , jQuery
  )
{
  'use strict';

  return CustomContentTypeBaseView.extend({
```

```

    template: sc_cct_imageviewer_tpl

    , install: function (settings, context_data)
    {
        this._install(settings, context_data);

        var promise = jQuery.Deferred();

        // The setTimeout method emulates an ajax call when the CCT is executed for the first time.
        setTimeout(function()
        {
            promise.resolve();
        }, 4000);

        return promise;
    }

    , contextDataRequest: ['item']

    , validateContextDataRequest: function()
    {
        return true;
    }

    , valign: {
        '1': 'top'
        , '2': 'center'
        , '3': 'bottom'
    }

    , getContext: function()
    {
        var texts = []
        , imageUrl = ''
        , valign = this.valign[this.settings.custrecord_sc_cct_iv_valign] || this.valign['3'];

        if (this.contextData.item)
        {
            var item = this.contextData.item();

            texts = [item.get('_name')];

            var thumbnail = item.get('_thumbnail');
            imageUrl = thumbnail.url ? thumbnail.url : thumbnail;
        }

        var set_text = Utils.trim(this.settings.custrecord_sc_cct_iv_text)
        , set_texts = set_text ? set_text.split('\n') : []
        , set_imageUrl = Utils.trim(this.settings.custrecord_sc_cct_iv_imageurl);

        texts = set_texts.length ? set_texts : texts;
        imageUrl = set_imageUrl ? set_imageUrl : imageUrl;

        return {
    
```

```

        hasText: !!texts.length
      , texts: texts
      , hasImage: !!imageUrl
      , imageUrl: imageUrl
      , valign: valign
    );
  }
});
});

```

Create a Template File

The template is necessary to render the data defined by the view.

To create a template:

1. In your CCT module's Templates directory, create a new .tpl file.
2. Name this file to intuitively relate to your module as a template.

For example:

`../SC.CCT.ImageViewer@0.0.1/Templates/sc_cct_imageviewer.tpl`

3. Create the HTML required to render the data defined by the view.

For the example ImageViewer CCT, your template might look similar to the following:

```

<div class="sc-cct-imageviewer">
  <div class="sc-cct-imageviewer-slider-container">
    <div class="sc-cct-imageviewer-image-slider">
      <ul data-sc-cct-imageviewer class="sc-cct-imageviewer-image-slider-list">
        <li>
          <div class="sc-cct-imageviewer-slide-main-container">
            {{#if hasImage}}
              
            {{/if}}
            {{#if hasText}}
              <div class="sc-cct-imageviewer-slide-caption sc-cct-imageviewer-slide-caption-{{valign}}">
                {{#each texts}}
                  <h2 class="sc-cct-imageviewer-slide-caption-title">{{this}}</h2>
                {{/each}}
              </div>
            {{/if}}
          </div>
        </li>
      </ul>
    </div>
  </div>

```

4. Save the template.

Set Up Your ns.package.json and distro.json Files

To set up your ns.package.json file:

1. Open your root CCT module directory.

- Create a new file in the custom module and name it `ns.package.json`.

For example:

`Modules/SC.CCT.ImageViewer@0.0.1/ns.package.json`

- Build the `ns.package.json` file using the following code:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*"
    ],
    "templates": [
      "Templates/*"
    ]
  }
}
```



Note: If your CCT includes any custom Sass, SuiteScript, auto-generated services, or configuration files, you must account for this here as well.

To set up your `distro.json` file:

- Open the `distro.json` file.

This file is located in the top-level directory of your SCA development directory.

- Add an entry for the new CCT module in the `modules` object to ensure that the Gulp tasks include your code when you deploy to NetSuite. It should look similar to the following:

```
{
  "name": "SuiteCommerce Advanced 1.0.0",
  "version": "1.0.0",
  "modules": {
    //...
    "suitecommerce/SC.CCT.Html": "1.0.0",
    "suitecommerce/SC.CCT.ImageViewer": "0.0.1",
    //...
  }
}
```

- Define any application dependencies for the desired application (Shopping, My Account, or Checkout), within the `javascript` object.

For the example ImageViewer CCT, you add the module to the `SC.Shopping.Starter` entry point.

```
//...
"javascript": [
  {
    "entryPoint": "SC.Shopping.Starter",
    "exportFile": "shopping.js",
    "dependencies": [
      "Backbone.View.Plugins",
      "jQuery.html",
      "ItemDetails",
      //...
      "SC.CCT.Html",
    ]
  }
]
```

```
"SC.CCT.ImageViewer"
],
//...
```

- If your CCT includes any Sass, include the module definition in the dependencies array of the desired application (Shopping, My Account, or Checkout) within the `sass` object.

For the example ImageViewer CCT, your code might look similar to the following:

```
//...
"sass": {
  //...
  "applications": [
    {
      "name": "Shopping",
      "exportFile": "shopping.css",
      "dependencies": [
        {
          //...
          "PickupInStore",
          "Location.SCA",
          "SC.CCT.ImageViewer"
        ]
      ]
    }
  ]
}
```

- Save the `distro.json` file.
- If you have not already set your NetSuite records, complete the tasks as defined in [Custom Content Type](#).



Important: Creating a custom module is only one step in the process. You must still perform the steps necessary to set up your custom module for use in SMT.

- Test your custom module in your local environment or deploy to your NetSuite account. See [Core SuiteCommerce Advanced Developer Tools](#) for details.

Customize the Checkout Application

Much of the SuiteCommerce Checkout application can be customized or configured by editing standard configuration properties and by using existing extensions or creating new extensions. See the help topic [Checkout](#) for detailed information and procedures on how to do this.

If, however, you need to implement an advanced customization for a SuiteCommerce Advanced solution, you can create custom modules and, using best practices, modify the checkout experience as described in this section of the documentation. See [Best Practices for Customizing SuiteCommerce Advanced](#).



Important: Whenever possible, advanced customization should be accomplished by creating new extensions. This allows for easier maintenance, portability, and release independence. Only implement advanced customizations by creating custom modules when the exposed extensibility APIs are not sufficient for your business use case. For details on the extensibility API see the help topic [Extensibility API](#), [Frontend Extensibility API](#), and [Extensions](#).

By default, several modules are included in each checkout configuration file. Available modules include:

- `OrderWizard.Module.Address.Billing`: lets the user add, edit, remove, or choose a billing address.

- **OrderWizard.Module.Address.Shipping:** lets the user add, edit, remove, or choose a shipping address.
- **OrderWizard.Module.Confirmation:** provides a confirmation message after an order is submitted.
- **OrderWizard.Module.PaymentMethod.Creditcard:** lets the user add, edit, remove, or choose a credit card as a payment method.
- **OrderWizard.Module.PaymentMethod.GiftCertificates:** lets the user add a gift certificate code as a payment method.
- **OrderWizard.Module.PaymentMethod.Invoice:** lets the user select an invoice option as a payment method.
- **OrderWizard.Module.PaymentMethod.PayPal:** lets the user select a PayPal account as a payment method.
- **OrderWizard.Module.PaymentMethod.Selector:** lets the user select from a list of payment methods.
- **OrderWizard.Module.PaymentMethod.PurchaseNumber:** displays a Purchase Order Number field to allow users to search by PO Number. This field will appear independently of the Payment Method selected by the user.
- **OrderWizard.Module.RegisterEmail:** gives guests the option of registering an email address.
- **OrderWizard.Module.RegisterGuest:** lets the user register with the site as a customer.
- **OrderWizard.Module.Shipmethod:** lets the user select a delivery method.
- **OrderWizard.Module.ShowPayments:** displays the payment type selected. This is typical of a review page.
- **OrderWizard.Module.ShowShipments:** displays the shipment method selected if MultiShipTo is enabled. This is typical of a review page.
- **OrderWizard.Module.TermsAndConditions:** creates a Terms and Conditions checkbox during the Review step. If this property is not set, the default is false.

To customize the checkout flow:

1. Create a custom module that includes the SC.Checkout.Configuration object as a dependency.

Note: Do not edit the SC.Checkout.Configuration.js file directly. See [Customize and Extend Core SuiteCommerce Advanced Modules](#) for information and best practices on customizing JavaScript.

2. Redefine the dependency **SC.Checkout.Configuration.Steps.Standard** to the desired checkout flow as defined below:

Checkout Flow	Dependency	Associated .js File
Standard (Default)	SC.Checkout.Configuration.Steps.Standard	SC.Checkout.Configuration.Steps.Standard.js
One Page Checkout	SC.Checkout.Configuration.Steps.OPC	SC.Checkout.Configuration.Steps.OPC.js
Billing First	SC.Checkout.Configuration.Steps.BillingFirst	SC.Checkout.Configuration.Steps.BillingFirst.js

3. Extend the checkout flow's associated .js file to customize the checkout configuration.

You can customize checkout in following ways:

- [Reorder Checkout Modules](#)
- [Add Checkout Modules](#)
- [Add or Remove Checkout Steps](#)

- Configure Checkout Step Properties
- Define Checkout Step URLs
- Extend the implementation to add your own custom cart



Note: The three checkout configurations shipped with SuiteCommerce Advanced are fully supported by NetSuite. You can create completely customized experiences, but note that advanced customization can create flows that do not make intuitive sense to the user. NetSuite cannot guarantee the outcome of advanced customizations.

Reorder Checkout Modules

You can reorder modules within a checkout step by repositioning the module within the code.

```

modules: [
    'OrderWizard.Module.PaymentMethod.GiftCertificates',
    'OrderWizard.Module.PaymentMethod.Selector',
    'OrderWizard.Module.Address.Billing',
    'OrderWizard.Module.RegisterEmail'
]

```



Important: If reordering modules, ensure that you do not disrupt any flow between modules. For example, you want to place both the shipping address and delivery method modules within the same step (Shipping Address). However, the delivery methods available depend on the shipping address. This creates a flow issue. To maintain flow consistency in this case, the shipping address module must always precede the delivery method module in a separate step. Be aware of flow inconsistencies that may exist within step or groups when customizing modules.

Add Checkout Modules

You can add modules to steps in multiple-step groups. For example, a guest shopper is prompted to register in the final confirmation step by default. By placing the OrderWizard.Module.RegisterGuest module in multiple steps, you can offer the registration at many points during the checkout process.

Each module is designed to use components and naming conventions consistent between the modules. When creating custom modules, we suggest that you follow similar conventions. For example, the RegisterEmail module contains the following components that are common among most other modules:

- **Module definition:** named from the most generic component (OrderWizard) to the most specific component (RegisterEmail).

```
define('OrderWizard.Module.RegisterEmail', ['Wizard.Module'], function (WizardModule)
```

- **Associated Template:** this defines the HTML layout for the rendered output. Templates are found in the templates folder of each specific module.

```
template: 'order_wizard_registeremail_module'
```

- **Render function:** the wizard object renders each step sequentially as it is defined in the code. The `_render()` function is responsible for displaying the template in the layout.

```
, render: function()
{
    this.profile = this.wizard.options.profile;

    // if the user is logged we dont render the module
    if (this.profile.get('isGuest') !== 'T')
    {
        this.trigger('ready', true);
    }
    else
    {
        this._render();
        if (this.profile.get('email') && this.wizard.isPaypalComplete())
        {
            this.trigger('ready', true);
        }
    }
}
```

- **Submit function:** defines the logic performed when the Continue button is clicked.

```
submit: function()
{
    var email = this.$('input[name=email]').val()
    , newsletter = this.$('input[name=sign-up-newsletter]').is(':checked')
    , self = this;

    // if the user is not guest or not change the current values we just resolve the promise
    if (this.profile.get('isGuest') !== 'T' || (email === this.profile.get('email') && this.profile.get('emailsubscribe') === (newsletter ? 'T' : 'F')))
    {
        return this.isValid();
    }

    this.profile.set('email', email);
    this.profile.set('confirm_email', email);

    return this.isValid().then(function() {
        self.profile.set('emailsubscribe', newsletter);
        return self.profile.save();
    });
}
```

- **isValid function:** defines error checking for entries.

```
isValid: function()
```

```
{
  var promise = jQuery.Deferred();

  if (this.wizard.options.profile.get('isGuest') !== 'T')
  {
    return promise.resolve();
  }

  if (!Backbone.Validation.patterns.email.test(this.wizard.options.profile.get('email')))
  {
    return promise.reject(_('Email is required').translate());
  }

  return promise.resolve();
}
```

Add or Remove Checkout Steps

Each checkout flow configuration file corresponds to a configured Checkout Flow Option. Each of these files contain multiple groups to achieve a specific task for that flow, such as adding shipping information. These groups contain one or more checkout steps to achieve that task. Each checkout step provides one or more wizard modules to guide the user through the checkout process.

Groups define the breadcrumbs available at the top of the page. Checkout steps then provide the modules that guide the shopper through the checkout process. You can add or remove checkout steps to any named group array by adding or removing steps as objects. You can also customize each step by editing properties.

For example, the following code snippet in the SC.Checkout.Configuration.Steps.Standard.js file defines the **Shipping Address** group. This group contains two checkout steps: Choose Shipping Address and Enter Shipping Address. Each checkout step contains three modules. You can customize the **Shipping Address** group by modifying each step's code.

```
{
  name: _('Shipping Address').translate()
, steps: [
  {
    name: _('Choose Shipping Address').translate()
    , url: 'shipping/address'
    , isActive: function ()
    {
      return !this.wizard.isMultiShipTo();
    }
    , modules: [
      OrderWizardModuleMultiShipToEnableLink
      , OrderWizardModuleAddressShipping
      , [OrderWizardModuleCartSummary, cart_summary_options]
    ]
  }
  , {
    name: _('Enter Shipping Address').translate()
    , url: 'shipping/selectAddress'
    , isActive: function ()
    {
```

```
        return this.wizard.isMultiShipTo();
    }
    , modules: [
        [OrderWizardModuleMultiShipToEnableLink, {exclude_on_skip_step: true}]
        , [OrderWizardModuleMultiShipToSelectAddressesShipping, {edit_addresses_url: 'shipping/selectA
ddress' }]
        , [OrderWizardModuleCartSummary, cart_summary_options]
    ]
}
]
```

Configure Checkout Step Properties

Use the following properties when configuring each checkout step. Set these properties in the applicable Checkout Steps configuration file.

- **name** (string): specifies the literal name for the step as it appears in the breadcrumb.
 - **getName** (function): specifies the dynamic name for the step as it appears in the breadcrumb. If this property is not defined, the default is the name property.
 - **url** (string): specifies the url for the step. This property is required and must be unique among all steps.
 - **continueButtonLabel1** (string): specifies the label of the **Continue** button for the step. If not specified, the default is Continue.
 - **hideBackButton** (boolean): specifies if the **Back** button appears on the site or not. If set to true, the button is hidden for this step. If not specified, the default is false.
 - **hideContinueButton** (boolean): specifies if the **Continue** button appears on the site or not. If set to true, the button is hidden for this step. If not specified, the default is false.
 - **hideSecondContinueButtonOnPhone** (boolean): specifies if the second **Continue** button displays on a smart phone or not. If set to true, the button is hidden for this step. If not specified, the default is false. Use this if there are too many **Continue** buttons in a step, such as the top, bottom and summary buttons).
 - **hideSummaryItems** (boolean): specifies if the cart summary's items are shown on this step or not. If set to true, the item is hidden for this step. If not specified, the default is false.
 - **hideBreadcrumb** (boolean): specifies if the page's breadcrumb is shown on this step or not. If set to true, the button is breadcrumb for this step. If not specified, the default is false.
 - **headerView** (string): specifies the main site header defined for the step. If not defined, the simplified header is used. You can define the normal 'header' or a custom one.
 - **footerView** (string): specifies the main site footer defined for the step. If not defined, the simplified footer is used. You can define the normal 'footer' or a custom one.
 - **bottomMessage** (string): defines a message to display at the bottom of the step above the **Continue** and **Back** buttons. If not specified, the default is no message.
 - **isActive** (function): specifies if the entire step is active (shown) or not, based on one or more conditions. The default implementation returns true. It is not required that all modules are active.
 - **save** (function): is a custom save function that finishes the wizard and saves the configuration. When used within this module, this property calls the submit function.
 - **present** (function): lets you customize the step to perform some action, such as trigger Google Analytics tracking.
 - **modules** (array): defines an array of modules used for the step.

Define Checkout Step URLs



Important: Each step must have a unique URL. This is important for maintaining site navigation, code statelessness, and for effective use of reporting, so you can perform analytics based off of URLs using tools such as Google Analytics.

You set each step's URL property to define the step's unique URL.

The screenshot shows the NetSuite Site builder interface. At the top, a browser window displays the URL <https://checkout.netsuite.com/c.1155683/checkout/index.ssp?n=5&sc=37#shipping/address>. Below the browser, the page title is "New NetSuite Site". Underneath that, a breadcrumb navigation shows "1. SHIPPING / 2. PAYMENT / 3. REVIEW & PLACE ORDER". The main content area is titled "Choose Shipping Address". On the left, there is a sidebar labeled "Ship To:". The right side contains a large block of JSON-like configuration code. A red box highlights the "url: 'shipping/address'" line in the code, which corresponds to the URL shown in the browser above. Another red box highlights the "modules: ['OrderWizard.Module.Address.Shipping']" line in the code.

```

{
  "checkoutSteps": [
    {
      "name": _("('Shipping').translate()"),
      "steps": [
        {
          "name": _("('Enter Shipping Address').translate()"),
          "getName": function() {
            if (this.wizard.options.profile.get('addresses').length) {
              return _("('Choose Shipping Address').translate()");
            } else {
              return _("('Enter Shipping Address').translate()");
            }
          },
          "url": "shipping/address",
          "modules": [
            'OrderWizard.Module.Address.Shipping'
          ]
        }
      ]
    }
  ]
}

```

Overview

 **Applies to:** SuiteCommerce Advanced

This section describes the architecture of the SuiteCommerce Advanced framework. You should have a solid understanding of the architecture if you intend to create extensions or do other advanced customization for your site.

- [Core Framework Technologies](#) – SuiteCommerce uses several 3rd party technologies to build the front-end framework. This section provides a general introduction to what is used.
- [SuiteCommerce Modules](#) – All of the front-end code is delivered in a collection of modules. This section describes the various types of modules and how they are structured.
- [Product Details Page Architecture](#) – The display of products in your web store is controlled by the product details page. This section provides detailed information on how this page is created and rendered.

Core Framework Technologies

ⓘ Applies to: SuiteCommerce Advanced

To understand how JavaScript source files are organized in modules and how these modules work together to create the SuiteCommerce application, you must understand the underlying technologies that the application is built on. To implement these technologies, SuiteCommerce uses several 3rd-party libraries that are common in web-based e-Commerce applications.

To discover more about these technologies, review the following sections:

Model View Controller (MVC) and Backbone.js

The JavaScript source files of each module follow the MVC architectural pattern. One of the core principles of the MVC pattern is to separate the presentation layer (view) of an application from the data (model) used by the application. All interaction between views and models are handled by the controller.

To implement the MVC pattern, SuiteCommerce Advanced uses the Backbone.js libraries. Backbone.js is an open source JavaScript library that provides a framework for developing web applications.

Structure of a Backbone.js Application

Backbone.js is based on the MVC pattern. Modules that define interfaces or handle data use Backbone.js to define routers, views, and models/collections. The Case module, for example, implements each of these:

- Routers: map URLs to client-side pages to methods that define actions and events. In feature modules, these methods initialize the views, models, and collections that handle the user interface and create frontend objects to handle data.
Each router contains a routes object that defines the mapping between URLs and methods.
- Views: contain methods that define the behavior of the user interface. Views are commonly associated with a model or collection. When a view is initialized, its corresponding model is usually passed. This ensures that the view has access to the data within the model. Most views define listeners that notify the view when an element within the model is modified. This enables a specific part of the application to be updated independently without having to reload the page.

With the Elbrus release of SuiteCommerce Advanced and later, all views are composite views by default and extend Backbone.CompositeView.js.

Views do not define the HTML or CSS of a feature or component. Instead, they specify the template that defines the HTML code for the component or feature. The templating engine handles the rendering and updating of the HTML.

- Models and Collections: define objects that contain data used by the application. They also define the methods that create, update, or delete data.

Backbone.js Module Example

For example, the Case module defines a router (Case.Router.js) that contains the following:

```
routes:
{
  'cases': 'showCasesList'
},
```

```
'cases?:options': 'showCasesList'
,
'cases/:id': 'showCase'
,
'newcase': 'createNewCase'
}
```

When a user clicks on an individual case from the list of support cases, instead of sending an HTTP request back to the server, the router calls the `showCaseList` method based on the value of the partial URL (cases). This method performs two main tasks:

- Initializes the model that contains data for a specific support case (Case.Model).
- Initializes the view that displays the data (Case.Detail.View)

The Case.Model model defines the data object that contains information about an individual support case. Case.Model extends the Backbone.Model by defining additional methods for performing frontend validation. When the router initializes Case.Detail.View, it passes the instance of Case.Model.

SuiteCommerce Advanced Backbone.js Implementation

The Backbone.js libraries provide a general framework for how a web application is structured and its general functionality. However, much of the specific functionality of the application must be implemented. SuiteCommerce Advanced stores Backbone.js files in the following directories:

- **Modules/third_parties/backbone.js**: contains the core Backbone.js libraries. In general, you should not modify the files in this directory.
- **Modules/suitecommerce/BackboneExtras**: contains extensions of core Backbone.js functionality, including Backbone.View and Backbone.Model. If you need to make changes to the Backbone.js framework, you may need to modify the files in this directory.

Asynchronous Module Definitions (AMD) and RequireJS

Although Backbone.js provides a general framework for web applications, it does not specify how application code should be organized. To organize code into modules, SuiteCommerce Advanced implements another design pattern called Asynchronous Module Definitions (AMD). AMD specifies how JavaScript files are loaded into an application. This includes the order in which files are downloaded and dependencies are evaluated.

To implement AMD, SuiteCommerce Advanced uses the RequireJS library. RequireJS is an open-source library that defines a framework for organizing code into modules. All modules in SuiteCommerce Advanced correspond to the syntax and requirements of AMD.

See [Dependencies](#) for more information on how modules implement the AMD design pattern.

Logic-less Templates and Handlebars.js

Another advantage of the Backbone.js libraries is that it provides open support for web templates and templating engines. Web templates contain the raw HTML of the SuiteCommerce website user interface. Each module that contains Backbone.js views uses a template to define the corresponding HTML. These files are stored in the Templates directory.

Templates define the HTML for discrete areas or features of the user interface. The template engine combines all of the template files into a single, finished web page.

SuiteCommerce Advanced uses the Handlebars.js library to implement templates and the template engine. One advantage of Handlebars.js is that it provides logic-less templates. This means that most of the business logic of the application is handled outside of the template. Within the Backbone.js framework, this logic is handled by the view. Using logic-less templates means that the HTML within the template is much easier to understand.

Although Handlebars.js is considered a logic-less template it does provide basic logical constructs. These are defined by Handlebars.js helpers. Helpers are primarily used to evaluate values of placeholders and display the appropriate HTML. Placeholders are like variables that contain information that is added to the HTML when the template is generated. This information is passed to the template by the `getContext` method of a view.

The following code snippet from the `case_list.tpl` template file shows how these are used to display the user interface according to the current state of the application.

```
<div class="case-list-results-container">
  {{#if hasCases}}
    <table class="recordviews-table">
      ...
      ...
      ...
    </table>
  {{else}}
    {{#if isLoading}}
      <p class="case-list-empty">{{translate 'Loading...'}}</p>
    {{else}}
      <p class="case-list-empty">{{translate 'No cases were found'}}</p>
    {{/if}}
  {{/if}}
</div>
```

In this example, the Handlebars.js engine evaluates the `hasCases` placeholder. This place holder corresponds to a property in the object passed to the template engine from the `getContext` method of the view. `hasCases` is a boolean property. If its value is true, the template engine outputs a table. If its value is false, the template engines checks the value of the `isLoading` property which is also passed from the `getContext` method. The template engine displays a message based on the value of this property.

In the above example, the `if` statement is an example of a default Handlebars.js helper. SuiteCommerce Advanced provides additional helpers that you can use in your templates. See [Custom Helpers](#).

Templates and the Template Context

SuiteCommerce Advanced uses templates to define the HTML code used to display features of the application. Templates perform the following primary tasks:

- Define the raw HTML code that displays the user interface of a browser-based application.
- Contain place holders that are replaced with data passed to the template.

Templates are compiled into functional JavaScript and HTML by a templating engine. SuiteCommerce Advanced uses the Handlebars.js library to define templates and the template engine. The template engine performs the following:

- Transforms the raw template file into a JavaScript function.

- Replaces place holders based on data passed to the template

Logic-less Templates

Logic-less templates ensure that the HTML code is separated from Application code. All application logic is handled in the view. For example, when the application needs to access data stored in a model, a view handles this interaction. Logic-less templates make the application more modular and ensure that the source code easier to maintain. More importantly, they ensure that when upgrading to a new version, any template customizations are not broken.

Template Compilation

The source files for each template contain the raw HTML code as well as Handlebars place holders. When you compile the application, one of the tasks the `gulp template` command performs is to pre-compile each of the template source files into a JavaScript function. The `gulp deploy` and `gulp local` commands call the `gulp template` command which in turn calls the Handlebars compiler.

The result of the `gulp template` command is that the template is transformed into a JavaScript function that can be called by passing a context object. This function is called when the view is rendered. Within the template file, the Handlebars.js place holders are transformed into JavaScript functions. For example the following `if` construct:

```
{{#if showLanguages}}
```

Is transformed into the following code when the template is compiled:

```
if(context.showLanguages){}
```

Handlebars.js placeholders that do not contain logic are also transformed into JavaScript functions. For example the following HTML code and placeholder:

```
<p>my name is {{userName}}</p>
```

Is transformed into the following when the template is compiled:

```
function(context){var s = '<p>my name is'; s += context.userName; s += '</p>'; return s; }
```

Interaction Between Templates and Views

Since templates do not contain application code and contain minimal logic, the data they require must be passed to them. When calling, the template method passes the context object. Templates only contain logic for performing loops and conditionals. Views contain all of the logic required to display the contents of a template. Views also contain all of the application logic. To display a template, the view calls the template function from the `view.render` method. For example, when the application is compiled and running, a view's `render` method would contain code similar to the following to access the values contained in context object:

```
var context = this.getContext() body.innerHTML = this.template(context)
```

In this example, the view passes the context object to the template function. The object passed to the template is the object returned by the `getContext` method. In SuiteCommerce Advanced, almost every view contains a `getContext` method. In general, this method is responsible for returning a context object that contains all of the data required by the template. This object may contain properties that contain data obtained from the model, configuration properties, or other properties the view requires.

Template Context

The template's context object returned by the `getContext` method, can be seen as a contract between the view and the template. The purpose of this contract is to ensure that any customizations you make to the template are not overridden or broken when upgrading to a new version of SuiteCommerce Advance. Any properties within the context object are not removed or modified in future versions. New properties may be added to the context object, but it will always be backward compatible with previous versions.

To ensure that this contract is maintained, the template only has access to this context object. The view is responsible for providing the context object and defining and respecting the contract in future versions.

Another area where the template context is important is within composite views. For example, you can customize a child view to use a custom template instead of the default template. When customizing an existing template or creating a new one, if you access the properties contained in the context object, these customizations will be preserved after an upgrade.

Custom Helpers

The Handlebars.js library defines helpers that perform functionality beyond that performed by the default placeholders. Handlebars.js defines multiple default helpers.

In addition to these default helpers, SuiteCommerce Advanced defines **custom** helpers that perform various tasks. These helpers are defined in the `HandlebarsExtras.js` file located in the JavaScript subdirectory of the `HandlebarsExtras` application module. The following helpers are defined:

- **translate**: returns a safe string that contains a localized version of the string by calling the `_.translate` function. Returning a safe string ensures that any HTML formatting or other text is preserved.
- **formatCurrency**: returns a formatted currency value by calling the `formatCurrency` function.
- **highlightKeyword**: returns a highlighted keyword by calling the `_.highlightKeyword` function. For example, the `site_search_item.tpl` template file uses this helper to highlight search results within a page using the following:

```
 {{highlightKeyword model._name query}}
```

- **displayMessage**: creates an instance of a `GlobalViewsMessageView` view and renders the view.
- **objectToAttributes**: returns the attributes of an object, causing them to be displayed in the template. This helper calls the `_.objectToAttributes` method.
- **each**: defines a custom each helper that supports a Backbone.js collection. This helper iterates through each item of the collection.
- **resizeImage**: builds URL images using the `imageSizeMapping` utility. This helper passes a URL and an object containing the dimensions of the image to this utility. It returns a normalized URL image based on these values.
- **fixUrl**: returns a valid URL by calling the `_.fixURL` utility.
- **trimHtml**: returns a trimmed HTML string based on the length passed to the helper. This helper calls the `jQuery.trim` method. The string returned can contain HTML elements.
- **breaklines**: places break tags (`
`) instead of new lines provided by the backend. Use this in Quotes, Returns, Case, and Review Order messages.
- **ifEquals**: creates an equality condition to a Handlebars `{{if}}` helper. The following example depicts use for a custom transaction body field:

```
 {{#ifEquals model.options.custbody_my_transaction_body_field_2 'T'}}  
 This is true
```

```
 {{else}}
  This is false
{{/ifEquals}}
```

SuiteCommerce Modules

 **Applies to:** SuiteCommerce Advanced

The source files for SuiteCommerce Advanced are organized in multiple modules. Each module defines a specific area of functionality which generally falls in one of the following categories:

- **Application modules:** define higher-level collections of features that perform similar types of functions. SuiteCommerce Advanced is composed of three separate applications:
 - Shopping
 - Checkout
 - My Account
- **Framework modules:** define the general logic and structure of SuiteCommerce Advanced and define how modules and applications work together. The application modules extend these modules to inherit the basic framework.
- **Feature modules:** define specific areas of functionality within SuiteCommerce Advanced. The Case module, for example contains the code and style sheets that implement Case Management feature.
- **Utility modules:** provide functionality that is used by multiple modules. The GlobalViews module, for example, defines views that are reused in multiple areas of SuiteCommerce Advanced.

Modules can be edited locally and stored in version control. Developer tools used to compile source files, templates, and style sheets into a deployable application.

The Modules Directory

All SuiteCommerce Advanced source files are contained in a directory called **Modules**. Within this directory, there are two subdirectories:

- **suitecommerce:** contains all of the application code for SuiteCommerce Advanced. This directory contains a subdirectory for each module.
- **third_parties:** contains all of the third party dependencies. This includes the core technologies that define the application framework.

Each module contains all source files required by a feature or utility. Each module contains some combination of the following types of files:

- **Configuration:** define the configuration metadata to build SuiteCommerce Configuration record properties related to the module.
- **JavaScript:** define the application code for the module.
- **SuiteScript:** define the services and backend models used by the module.
- **Templates:** define the raw HTML code for the module.
- **Sass:** define the style sheets used by the template files.

The source directory of a module contains subdirectories for each of these types. However, not all modules implement each of these file types. A utility module, for example, may contain only JavaScript files.

Dependencies

Every JavaScript file in the SCA source contains the same general structure that corresponds to Asynchronous Module Definition (AMD) API. RequireJS uses the AMD API to asynchronously load modules within the application. The following shows the general structure of each file:

```
define('<module_name>'  
, [  
    '<dependency_1>'  
,  
    '<dependency_2>'.  
    ...  
,  
    ...  
,  
    ...  
)  
{  
    <module code>  
}
```

All files within a module contain one define function which occurs at the beginning of the file and uses the following syntax:

```
define(module_id, [dependencies], function)
```

- **module_id** — a string defining the name of the module.
- **dependencies** — an array containing the names of the dependencies. In SCA, this is generally the name of a module, a component within a module, or a core dependency.
- **function** — defines a function that instantiates the module.

By following this structure consistently, the developer tools can compile the application into a single JavaScript file by calculating the cascading dependencies starting with the root level application modules down to each lower-level feature and utility modules. See [Core SuiteCommerce Advanced Developer Tools](#) for more information.

Application Modules

SuiteCommerce Advanced is composed of three separate applications: Checkout, My Account, and Shopping. Each of these applications contains its own top-level modules that define the dependencies and general layout. These modules are contained in the following directories:

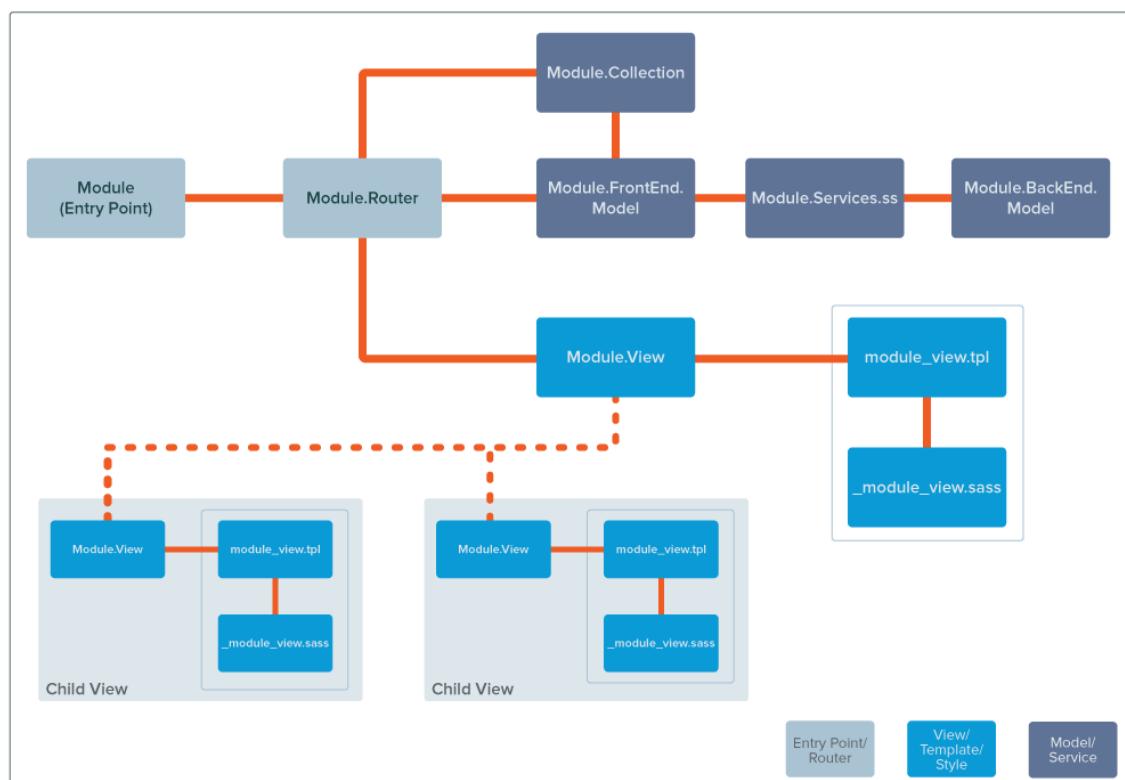
Application	Directory	Entry Point
My Account	MyAccountApplication MyAccountApplication.SCA	SC.MyAccount.Starter.js

Checkout	CheckoutApplication	SC.Checkout.Starter.js
Shopping	Shopping Application	SC.Shopping.Starter.js

Each application module contains a starter file which defines the starting node or entry point for each application. The starter file lists all of the dependencies for the application. Any dependencies not listed in this file will not be included in the application.

Note: When creating a new module, you must add it to the starter file for each application that uses the new module. See [Create a Custom Module](#) for more information.

The following image show the hierarchical relationship between the top-level application modules and the lower-level feature modules.



Module Architecture

Most of the modules within SuiteCommerce Advanced define the behavior for a specific feature or related functionality. In contrast to the application and framework modules which define the underlying infrastructure and logic of an application, these modules perform the following:

- Define the user interface for a feature.
- Define the low-level application behavior.
- Define data models.
- Handle data transfer between the application and NetSuite.

Feature modules generally conform to the Model-View-Present (MVP) design paradigm prescribed by Backbone.js. This means that they implement some combination of the following:

- Routers
- Views
- Models
- Collections

Entry Point

Most feature module contain an entry point which defines how the module interfaces with the top-level application module. The entry point of a module is defined in a file called <module_name>.js.

All modules that are dependencies of the application define a method called **mountToApp**. The ApplicationSkeleton module calls this method for each module listed as a dependency in the starter file of the application. For most modules, the **mountToApp** method initializes and returns the router for the module, making the module available to the higher-level application context.

For modules that are included in the My Account application, the entry point also returns a **MenuObject** item. This object contains information about the module that the **SC.MyAccount** node uses to create and display the menu items that appear in My Account.

Routers

Within a single-page application, routers direct URLs to client-side methods rather than passing an HTTP request to the server. All routers contain a **routes** property that maps between a partial URL and a method defined within the router. Methods within the router perform different functions including initializing views, models, and collections.

For example, the Address.Router contains the following routes property:

```
routes: {
  'addressbook': 'addressBook'
, 'addressbook/new': 'newAddress'
, 'addressbook/:id': 'addressDetailed'
}
```

When a user clicks on the Address Book menu item in the My Account application, the router of the Address module intercepts the partial URL (addressbook) and calls the **addressBook** method. The **addressBook** method is responsible for initializing the collection containing the list of addresses and initializing the view that displays them. Depending on the module, routers may initialize properties and variables required by the module. They may also contain methods that define the behavior and logic of the module.

Views

Within the Backbone.js framework, views listen to the user interface events and coordinate with the data required by the associated module. In some modules, when a router initializes a view, the router also passes a model or collection containing the data required. In other modules, the view includes all necessary models or collections as dependencies.

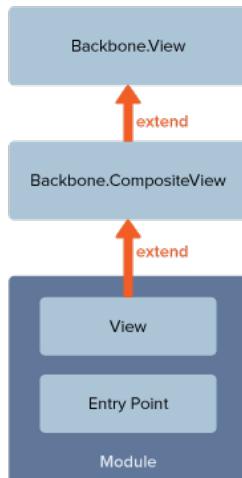
View Architecture

This section applies to the **Elbrus** release of SCA and later.

All Backbone.Views are Backbone.CompositeViews by default. Composite views are simply views that contain child views. Composite views further modularize the application, so some types of views can be used in multiple contexts. For example, some modules define Child Views that are extensions of Backbone.View. As a consequence, those Child Views are also Backbone.CompositeViews. This ensures that some types of data are displayed in a consistent way across the application.

All views extend Backbone.CompositeView by default. Any custom views should use similar code to the example below:

```
...js
var MyView = Backbone.View.extend({
  initialize: function()
  {
    //every view now is a composite view!
  }
});
...
```



A Parent View must declare its Child Views. To do this, each Parent View declares the **childViews** property, which is an object containing all the Container names from the View (the data-views). Each container has an object with the Child View's name and an object for each View with the following information:

- **childViewIndex** – this is the order in which the children render in that container.
- **childViewConstructor** – This can be either functions or Backbone.View subclasses. Each defined function must return an instance of a Backbone.View.

The following example declares the **childViews** property, containing the **Buttons** container. The Buttons container declares the **Wishlist** and **SaveForLater** objects. You can include multiple views in one container.

```
...
  childViews: {
```

```

'Buttons': {
  'Whishlist': {
    'childViewIndex': 10
    , 'childViewConstructor': function() {
      return new WhishlistView();
    }
  }
  'SaveForLater': {
    'childViewIndex': 20
    , 'childViewConstructor': SomeView
  }
}
...

```

i Note: You can add new Child Views to a View class using the `addChildViews()` method. You can also add new Child Views to an instance of a view using the `addChildViews()` method or by passing the `childViews` property in the options when initializing it.

To add a plugin before or after you initialize the View, add the plugin to the View Class and execute code in those moments (`beforeInitialize` and `afterInitialize`), as shown below:

```

...
Backbone.View.beforeInitialize.install({
  name: 'compositeView'
  , priority: 1
  , execute: function ()
  {
    var childViews = _.extend({}, this.childViews, this.constructor.childViews);

    this.childViewInstances = {};

    this.addChildViews(childViews);

    if (this.options)
    {
      if (this.options.extraChildViews)
      {
        console.warn('DEPRECATED: "options.extraChildViews" is deprecated. Use "options.childViews" instead');
        //Add extra child views from view's initialization options

        this.addChildViews(this.options.extraChildViews);
      }

      if (this.options.extraChildViews)
      {
        //Add child views from view's initialization options
        this.addChildViews(this.options.extraChildViews);
      }
    }
  });
...

```

Backwards Compatibility

To make this change backward compatible with earlier releases of SCA, Backbone.CompositeView.js includes a CompositeView.add() method as a no operation (noop) method. This prevents any errors when calling it from any custom code in implementations prior to Elbrus release. SCA supports the previous format as shown below:

```
...
childViews: {
  'Promocode': function()
  {
    return new PromocodeView({model: this.model.get('promocode')});
  }
, 'SomeWidget': SomeWidget
}
...
...
```

In this case, the container and the view name have the same, and you can combine the old and the new format and in the `childViews` property.

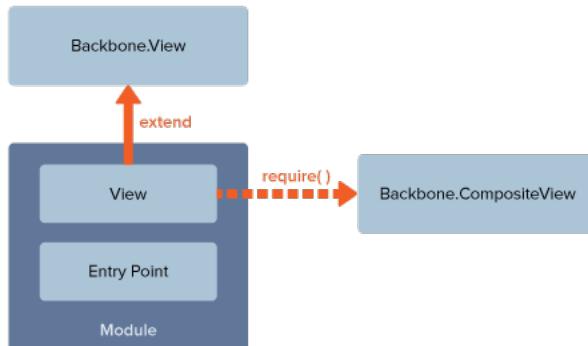
```
...
childViews: {
  Promocode:
  {
    'Promocode':
    {
      childViewIndex: 10
      , childViewConstructor: function(options)
      {
        return new PromocodeView({model: this.model.get('promocode')});
      }
    }
  }
}
...
...
```

View Architecture (Vinson Release and Earlier)

This section applies to the **Vinson** release of SCA and earlier.

In the Vinson release of SCA and earlier, each view must declare itself as a composite view using the `require()` method and call `CompositeView.add(this)` within the `initialize()` method. Any custom views should use similar code to the example below:

```
...
var CompositeView = require('Backbone.CompositeView');
var MyView = new backbone.View.extend({
  initialize: function()
  {
    CompositeView.add(this);
    //now this view is a composite view!
  }
});
...
...
```



Rendering Data From the View

The process of rendering the HTML is handled by the templating engine and the Backbone.js framework. To work within this framework, each view defines the following:

- A template file included as a dependency within the define method of the view.
- A template property defined in the body of the view.
- A `getContext` method that returns all of the data attributes required by the template.

When you compile SuiteCommerce Advance using the developer tools, Gulp.js uses these elements to compile the templates into the application. See [Logic-less Templates and Handlebars.js](#) for more information on templates and the template engine.

If implementing the Elbrus release of SCA or later, a data view acts as a container for multiple child views. Any parent view must declare a placeholder element in its associated template to render the children within it as depicted in the following example:

```

...
html
<div data-view="Buttons"></div>
...

```

Models, Collections, and Services

SuiteCommerce Advanced works with data stored in NetSuite. During a shopping session, for example, SuiteCommerce Advance may perform the following data-related actions:

- Read data from NetSuite records. For example when a user views details about an item, this information is retrieved from the NetSuite item record.
- Update data based on user input. For example when a user places an order, information about the order is stored in a NetSuite order record.
- Receive new data based on user input. For example, when a user adds a new address to the Address Book, the NetSuite address record is updated.

To retrieve, create, and update data, SuiteCommerce Advanced implements ways of handling the data on the frontend and backend.

- Frontend: defines how data is presented to the user and how the user inputs data. This data is handled by frontend models and collections.

- backend: defines how SuiteCommerce Advanced transfers data to and from NetSuite. This transfer is handled by services and backend models.

Frontend Models

Frontend models contain the data that SuiteCommerce Advanced uses. Frontend models are part of the Backbone.js framework that defines the data component within the MVC paradigm. Each frontend model is an extension of Backbone.Model which is defined in BackboneExtras. Backbone.Model defines inheritable methods that are available to all models defined in SuiteCommerce Advanced. If necessary, a model can override these core methods. For example, many modules override the `initialize` method to set values of properties that are specific to that model.

Most feature modules define models that hold the data they require. There are ways models are defined in SuiteCommerce Advanced:

- Models that are specific to features are defined by returning an object containing the model. These models are defined using the following syntax:

```
return Backbone.Model.extend
```

- Models that are used by the wider application context. These models are created and their data populated when they are initialized. These models are defined in the following manner:

```
ProfileModel = Backbone.Model.extend
```

By defining a model this way, it can be accessed from models and views that are defined outside of the current module. In this example, the ProfileModel can be accessed by other modules that need to access information stored in the user's profile. Most models that are created this way are defined as singletons. This means that there can only be one instance defined per user session.

When a router or view initializes a model, it may pass properties that the model requires. These can be properties that the frontend model uses internally or properties that are passed to the backend model.

In general, frontend models contain code that performs the following:

- Set initial values of properties.
- Define model-specific methods.
- Override default methods.
- Perform frontend validation of user-submitted data.

All frontend modules define the following properties:

- `urlRoot`: specifies the backend service that is used to handle HTTP requests. This property is a string containing a partial URL.
- `validation`: specifies an object that defines the properties that are validated when a user submits form data.

Asynchronous Data Transactions

When working with data stored in NetSuite or internally within the application, SuiteCommerce Advanced often needs to perform tasks asynchronously. Handling tasks asynchronously improves the performance and user experience of SuiteCommerce Advanced.

To handle tasks asynchronously, SuiteCommerce Advanced uses the jQuery Deferred Object API. This API provides an easy way of registering callbacks and monitoring the success or failure of data transactions. See <https://api.jquery.com/category/deferred-object/> for information on the methods provided by the Deferred Object API.



Note: Although there are other mechanisms JavaScript for performing tasks asynchronously, NetSuite recommends that you use jQuery Deferred Objects when customizing or extending SuiteCommerce Advanced.

There are two main contexts where SuiteCommerce Advanced uses the Deferred Object API:

- When performing AJAX calls when using the `save()`, `fetch()`, and `destroy()` methods of a model or collection.
- When performing tasks asynchronously using promises

Using the Deferred Object in AJAX Calls

SuiteCommerce Advanced frequently interacts with data stored in NetSuite. To perform a data transaction, code within a Router or View calls a method on the front-end model. This method then makes an AJAX call that sends an HTTP request to the service.

In general, this AJAX call needs to be asynchronous since other application code may need to continue processing while the data is loading. The primary reason for using asynchronous AJAX calls is that each browser window operates within a single thread. A synchronous AJAX call will freeze the browser until the AJAX call is complete.

One frequent use of the deferred object is to wait until the AJAX call is complete to perform other tasks.

The following example from `Case.Detail.View` shows how the `done` method is used when saving data to NetSuite:

```
this.model.save().done(function()
{
    self.showContent();
    self.showConfirmationMessage(_('Case successfully closed').translate());
    jQuery('#reply').val('');
});
```

In this example, the `done` method accepts a single function as an argument. This function calls the `showContent()` method of the view and performs other tasks. This defers calling the `showContent()` method until the data transaction initiated by the `save` method is completed.

Using Promises in Asynchronous Data Transfers

SuiteCommerce Advanced also uses the Deferred Object API in other contexts where tasks need to be performed asynchronously. In this case, a module explicitly creates an instance of the `jQuery.Deferred` object. After creating this object the module must ensure that the deferred object is ultimately resolved. This resolution is handled using promises.

Promises are part of the jQuery Deferred Object that enable asynchronous data transfers. Promises prevent other methods from interfering with the progress or status of the request. The following code example from the `ProductList.CartSaveForLater.View` module shows how a deferred object is created and ultimately resolved:

```
addItemToList: function (product)
{
    var defer = jQuery.Deferred();

    if (this.validateGiftCertificate(product))
    {
        var self = this;
```

```

this.application.ProductListModule.Utils.getSavedForLaterProductList().done(function(pl_json)
{
    if (!pl_json.internalid)
    {
        var pl_model = new ProductListModel(pl_json);

        pl_model.save().done(function (pl_json)
        {
            self.doAddItemToList(pl_json.internalid, product, defer);
        });
    }
    else
    {
        self.doAddItemToList(pl_json.internalid, product, defer);
    }
});

else
{
    defer.resolve();
}

return defer.promise();
}

```

Stopping AJAX Requests

SuiteCommerce Advanced includes an `AjaxRequestsKiller` utility module used to stop execution of AJAX requests. Performance is enhanced by stopping AJAX requests that are no longer needed. This module is primarily used in two contexts:

- **URL changes:** When a URL change is detected, this module stops execution of all pending AJAX requests that were initiated by another Router or View. Some AJAX requests are initiated without a URL change, for example, the `collection.fetch` and `model.fetch` methods. In such cases, you should pass the `killerID` property to the method as in the following:

```

model.fetch({
    killerId: AjaxRequestsKiller.getKillerId()
})

```

- **Site Search:** The `SiteSearch` module uses the `AjaxRequestsKiller` module to delete unnecessary AJAX calls when using the type ahead functionality. As a user enters a character in the search field, the `SiteSearch` module sends an AJAX request to retrieve information from the Item Search API. As the user continues entering characters, additional AJAX requests are sent. Using the `AjaxRequestsKiller` improves performance by deleting old requests.

When the value of the `killerID` property is reset, this module aborts the AJAX request associated with the old value of the `killerID` property. It also sets the `preventDefault` property of the request to `true` to avoid sending an error to the `ErrorHandling` module.

This module defines the following properties and methods:

- **getKillerId** (method): returns an unique ID that identifies the AJAX request. Each time the URL is reset, a new ID is generated.
- **getLambsToBeKilled** (method): returns an array of AJAX request IDs to be halted.

- **mountToApp** (method): loads the module into the application context. This method performs the following
 - Sets the value of the **killerID** property.
 - Pushes the **killerID** property to an array.
 - When the **killerID** property is reset, calls the abort method on the AJAX request.

Collections

Collections are sets of models. Define the data object that contains the set of models. Define methods that perform actions when the data in a model changes. They can also perform actions on all models within the collection.

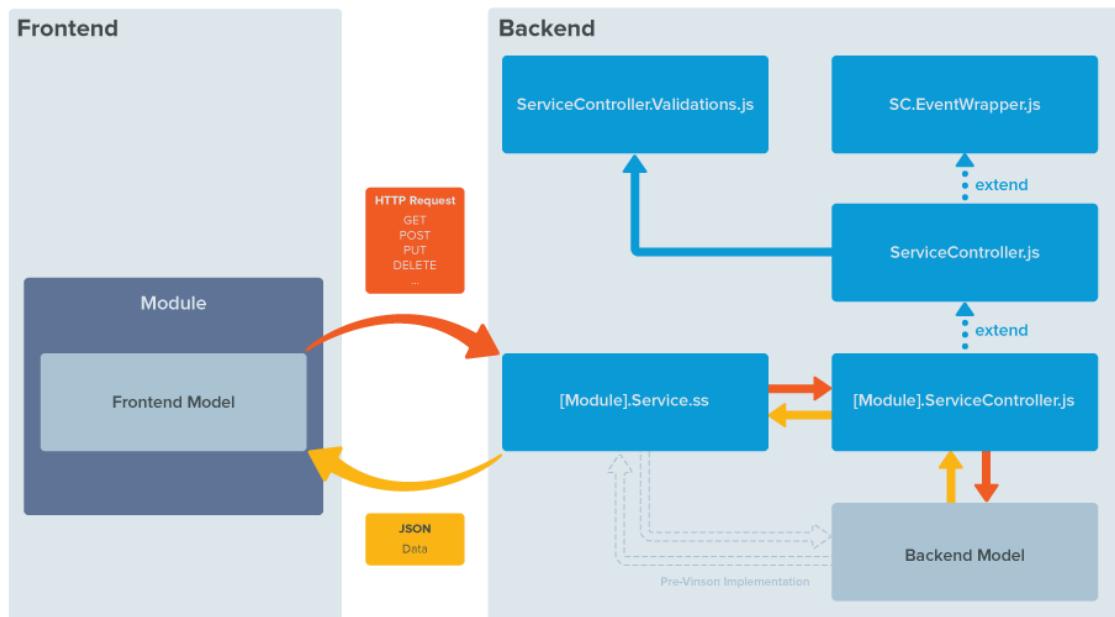
SuiteCommerce Advanced often uses collections within displaying lists of items. The Case module, for example, uses the Case.Collection to load a list of support cases assigned to a user. The Case.List.View uses the collection to display the list.

Services and Backend Models

To handle data transactions with NetSuite, SuiteCommerce Advanced provides multiple endpoints that enable access to data stored in NetSuite records. How SuiteCommerce Advanced handles these RESTful services depends on your implementation. For details on services architecture when implementing earlier versions of SCA, see [Architecture Overview \(Pre-Vinson\)](#)

This section explains the service and backend model architecture for Vinson release of SuiteCommerce Advanced and later. Services handle communication between the frontend SuiteCommerce application and the backend models that connect to NetSuite. With Vinson release, these (.ss files) are auto-generated and call Service Controllers. These Service Controllers and other files maintain code common to all services within a centralized location, as described below.

Note: The Vinson release of SuiteCommerce Advanced is backward-compatible with pre-Vinson services. This requires editing the associated module's ns.package.json files. See [Using Pre-Vinson Services with Vinson Release or Later](#) for more information.



The following components make up the backend service architecture:

SC.EventWrapper

This file contains centralized, extensible Before and After Event listening capabilities plus extension capabilities for use by all service methods. SC.EventWrapper.js is located in the SspLibraries module.

ServiceController

This file contains the common logic used to operate any service. ServiceController contains the `handle()` method, which executes the method related to the HTTP requests identified in any [Module].ServiceController. ServiceController.js is located in the SspLibraries module.

ServiceController contains the required try/catch block to ensure that any errors are passed back to the frontend application and are handled by the ErrorManagement module. The ServiceController module also extends SC.EventWrapper and references validation methods declared in ServiceController.Validations.



Note: All instances that extend ServiceController are located in the SuiteScript folder of the module during development. When you deploy to the server, the specific [Module].ServiceController code writes to the `ssp_libraries.js` file.

[Module].ServiceController

One [Module].ServiceController exists per service. This file processes requests for a specific service and extends code within ServiceController. Each [Module].ServiceController executes the proper functions based on the HTTP actions (GET, POST, PUT, DELETE) initiated by the frontend framework, which are passed through the associated .ss file. [Module].ServiceController can also call methods on any necessary backend models.

Each [Module].ServiceController can include an `options` object, which details any permissions and validations needed for the HTTP methods. Each service requires a unique [Module].ServiceController.

[Module].Service.ss

This file handles communication between the frontend SuiteCommerce application and the associated controller ([Module].ServiceController). The .ss file is automatically generated in the `services` folder of the local distribution. After you deploy your site, the .ss file exists in the `services` folder of the proper SSP Application. One .ss file exists per service.

Each [Module].Service.ss file is also responsible for passing data from the backend model to the frontend model or collection that initiated the HTTP request. This data is passed as a JSON object from the backend model to the .ss file via the associated [Module].ServiceController.

The following example shows the auto-generated .ss file for the Account Login service. This file only calls the [Model].ServiceController associated with the service as defined in the ns.package.json file.

```
//Autogenerated service function.
function service(request, response)
{
  'use strict';
  require('Account.Login.ServiceController').handle(request, response);
}
```



Note: When you deploy your site, the `autogenerated-services` object of the associated module's `ns.package.json` file signals the dev tools to auto-generate each `.ss` file and associate it with a specific service controller. Failure to include this code will result in an error and the `.ss` file will not generate. If you want to use pre-existing (preVinson) services with a Vinson implementation, you must register the service in the `services` section of the associated module's `ns.package.json` file. See [Using Pre-Vinson Services with Vinson Release or Later](#) for more information.

Backend Model

The backend model contains methods that access NetSuite records by calling the SuiteScript API. The Backend Model architecture and logic have not changed with the Vinson Release of SuiteCommerce Advanced.

ServiceController.Validations.js

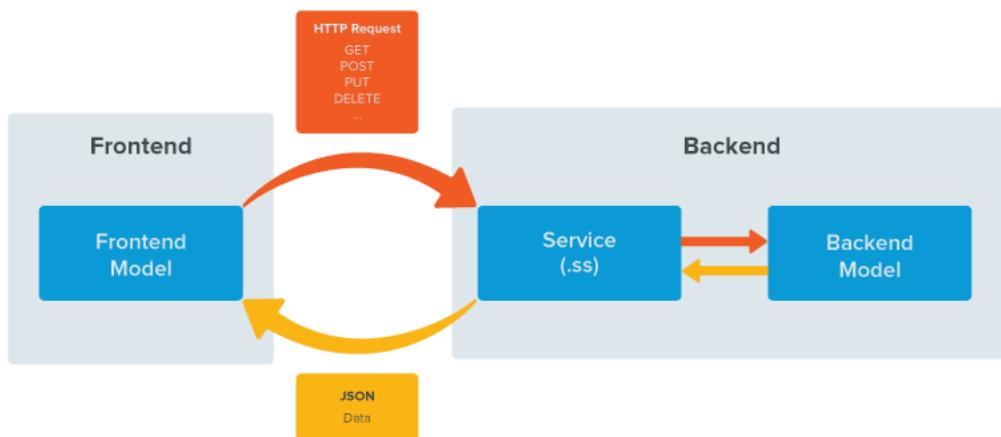
This file contains the extensible validation methods for requests common to all services. This file is located in the `SspLibraries` module.

Each feature or module in SuiteCommerce Advanced generally defines the services and backend models needed to access the required data. The Cart module, for example, defines the services and backend models for accessing items currently in the cart. Each module's `[Module].ServiceController` files and backend models are stored in the JavaScript directory of each module.

When you deploy SuiteCommerce Advanced, the gulp tasks auto-generate the required `.ss` files and deploy all of the services to the SSP application. The gulp tasks also add backend models to the `ssp_libraries.js` file. See [The `ssp_libraries.js` File](#) for more information.

Architecture Overview (Pre-Vinson)

To handle data transactions with NetSuite, SuiteCommerce Advanced provides multiple endpoints that enable access to data stored in NetSuite records. These endpoints run on NetSuite and are implemented as RESTful APIs.



Each endpoint has the following components:

Service (request handler)

The service (`.ss` file) handles communication between the frontend SuiteCommerce application and the backend models that connect to NetSuite. The service receives an HTTP request initiated by the

frontend framework, usually a model or collection. Services handle standard HTTP request methods (GET, POST, PUT, DELETE). Based on the HTTP action specified in the request, the service calls methods on the backend model.

For example, the ProductLists.Service.ss service returns information about product lists. Since this is a RESTful service, you can access product list information using HTTP methods as follows:

- **HTTP POST** — create a new product list.
- **HTTP PUT** — update a product list.
- **HTTP DELETE** — delete a product list.
- **HTTP GET** — perform a search on a product list. This is accomplished by passing search filters as parameters to the service.

Services are also responsible for passing data from the backend model to the frontend model or collection that initiated the HTTP request. This data is passed as a JSON object.

Backend Model

The backend model contains methods that access NetSuite records by calling the SuiteScript API. See the help topic [SuiteScript API](#) for more information.

Each model defines methods that correspond to the HTTP actions handled by the service. For example, if a model is handling a GET action to retrieve data from NetSuite, it must also format the corresponding JSON object that is returned to the frontend framework.

Each feature or module in SuiteCommerce Advanced generally defines the services and backend models needed to access the required data. The Cart module, for example, defines the services and backend models for accessing items currently in the cart. Services and backend models are stored in the JavaScript directory of each module.

When you deploy SuiteCommerce Advanced, the gulp tasks deploy all of the services to the SSP application. The gulp tasks also add backend models to the `ssp_libraries.js` file. See [The `ssp_libraries.js` File](#) for more information.

Services and Backend Models in Custom Modules

When creating a custom module to access NetSuite records you must read the following topics.

Define a JSON Object to Represent a NetSuite Record

To use the data in a NetSuite record, you must determine how to represent the record as a JSON object. For example, a custom record containing a set of questions and answers may have the following structure:

- **record name** —`customrecord_q_and_a`
- **custom field** —`custrecord_q_and_a_question`
- **custom field** —`custrecord_q_and_a_answer`

A JSON object representing this custom record would look like the following:

```
{
  "question": "A String with the question"
, "answer": "A String with the answer"
, "createdAt": "2016-03-31"
}
```

This example only requires name/value pairs representing each custom field and its value. However, depending on the complexity of the record you need to represent, a JSON object can be more complex. For example, a JSON object can contain arrays or nested objects.

After defining the formal structure of a JSON object, you can define the backend model that accesses the NetSuite record. See [Create a Backend Model](#) for more information.

Create a Backend Model

Backend models handle data transactions with NetSuite using the SuiteScript API. The following example shows a typical backend model that reads and modifies a custom record:

```
define(
    'QuestionsAndAnswers.Model'
, ['SC.Model', 'Application']
, function (SCModel, Application)
{
    'use strict';

    return SCModel.extend({


        name: 'QuestionsAndAnswers'

        , validation: {
            question: {required: true, msg: 'The question is required'}
        }

        , search: function(page)
        {
            var filters = [
                new nlobjSearchFilter('custrecord_q_and_a_answer', null, 'isnot', '')
            ]
            , columns = [
                new nlobjSearchColumn('custrecord_q_and_a_question')
                , new nlobjSearchColumn('custrecord_q_and_a_answer')
                , new nlobjSearchColumn('created')
            ];
            return Application.getPaginatedSearchResults({
                record_type: 'customrecord_q_and_a'
                , filters: filters
                , columns: columns
                , page: parseInt(page, 10) || 1
            });
        }

        , create: function(data)
        {

            this.validate(data);

            var question = nlapiCreateRecord('customrecord_q_and_a');
            question.setFieldValue('custrecord_q_and_a_question', data.question);
            var question_id = nlapiSubmitRecord(question);

            return data;
        }
    });
}
```

```

    }
});

});
}

```

This example demonstrates the following tasks that are common to all backend models.

Define Dependencies

Like frontend modules, backend models use RequireJS to define dependencies. See [Dependencies](#) for more information.

All backend models must include the following dependencies:

- **SC.Model** — defines the base class for backend models. All backend models should extend this module.
- **Application** — provides useful methods, including server-side search helpers.

Extend from SC.Model

All backend models must return an object that is extended from SC.Model.

```
return SCModel.extend()
```

SC.Model defines the base model which defines core functionality, including validation.

Define the Validation Object

The **validation** object defines any validation requirements for the JSON object.

```

validation: {
    question: {required: true, msg: 'The question is required'}
}

```

The Backbone.Validation module uses this object to perform validation. Any validation errors are returned to the frontend application via the service. The SC.Model module includes Backbone.Validation as a dependency.

In the above example, the **required** attribute indicates that the **question** property of the JSON object must have a value. If this property is undefined, the backend model returns the message specified by the **msg** property.

Define Methods to Handle HTTP Actions

A backend model must define a method to handle each of the HTTP actions supported by a service. These methods frequently use the SuiteScript API to handle data transactions with NetSuite. For example, the **search** method in the above example shows how a backend model retrieves data from a NetSuite record:

```

search: function(page)
{
    var filters = [
        new nlobjSearchFilter('custrecord_q_and_a_answer', null, 'isnot', '')
    ]
    , columns = [
        new nlobjSearchColumn('custrecord_q_and_a_question')
        , new nlobjSearchColumn('custrecord_q_and_a_answer')
    ]
}

```

```

        , new nlobjSearchColumn('created')
    ];

    return Application.getPaginatedSearchResults({
        record_type: 'customrecord_q_and_a'
        , filters
        , columns
        , page: parseInt(page, 10) || 1
    });
}

```

In this example, the `search` method performs the following:

- Creates an array containing search filters. Each search filter is defined by calling the `nlobjSearchFilter` method. Since this model is only retrieving data for one record, only one search filter is required.
- Creates an array for each column of the NetSuite record. In this example, each column corresponds to a property in the JSON object.
- Calls the `getPaginatedSearchResults` of the Application module. This method returns a JSON object based on the parameters it receives. This JSON object contains all of the `customrecord_q_and_a` records in NetSuite.

Create a Service to Handle HTTP Requests

Services handle HTTP requests sent by the frontend application to access backend models, which connect to NetSuite. Different releases of SuiteCommerce Advanced handle services with significant differences. This section explains how to create a service when implementing the Vinson release of SuiteCommerce Advanced or later.



Note: If using existing, customized services from previous implementations of SCA with the Vinson release, you can still use them without refactoring your custom code. Vinson offers backward compatibility with any services of previous implementations. See [Using Pre-Vinson Services with Vinson Release or Later](#)

With the Vinson release of SuiteCommerce Advanced, much of the logic required to operate a service is contained in a centralized location within the source files provided. However, you must still perform the following to create a custom service:

- Step 1: Create a custom `[Module].ServiceController`
- Step 2: Set up the Dev Tools to auto-generate the service and associate it with a service controller
- Step 3: Customize your own validations (optional)

Step 1: Create a Custom `[Module].ServiceController`

Each service requires a unique `[Module].ServiceController`. You create this file, where `[Module]` is the module using the service. To create this custom ServiceController, perform the following:

- Create a new `[Module].ServiceController` file in the SuiteScript folder of the module.
- Define the custom ServiceController and dependencies.
- Extend `ServiceController`.
- Add any custom validation options (optional).
- Add any HTTP methods as required for the service.

To create a new [Module].ServiceController file:

1. Create a new file titled [Module].ServiceController.js, where [Module] is the module associated with the service.
2. Place this file in the SuiteScript folder of the module.

For example, if you are creating a new login service for the Account module, you would title it Account.Login.ServiceController.js and place it in your local Account/SuiteScript folder.

To define your custom [Module].ServiceController and dependencies:

1. In your new [Module].ServiceController, define the specific ServiceController and any dependencies.
2. Include ServiceController as a dependency

The following code defines the Account Login ServiceController and enables the service to access methods defined in ServiceController and Account.Model.

```
// Account.Login.ServiceController.js - Service to handle the login of a user into the system
define(
  'Account.Login.ServiceController'
, [
  'ServiceController'
, 'Account.Model'
]
, function(
  ServiceController
, AccountModel
)
```

To extend ServiceController:

1. [Module].ServiceController must extend ServiceController and include the **name** property. This property is used when listening to an event is needed. The value of the name property should match the name of [Module].ServiceController.
2. Use the following example as a guide. This shows this extension with the **name** property.

```
{
  'use strict';
  return ServiceController.extend({
    name:'Account.Login.ServiceController'
```

To add custom Validation Options (Optional):

1. If you need to define any validation options for the service, include these within **return ServiceController.extend**.
2. Use the following example as a guide. This defines a requirement for the HTTP protocol to be secure before operating this service.

```
, options: {
  common: {
    requireSecure: true
  }
}
```

To add HTTP methods:

Within the object returned, define the functions required to handle the HTTP methods for the service. This code reads the HTTP method sent from the frontend (POST, DELETE, GET or PUT).

1. For your [Module].ServiceController to function, you must declare each method in lowercase and it must match the name of the HTTP method being used.
2. Use the following example as a guide. This defines what the service does on a POST request.

```
, post: function()
{
    return AccountModel.login(this.data.email, this.data.password, this.data.redirect)
}
```

Using these examples, your custom [Module].ServiceController might look something like this:

```
// Account.Login.ServiceController.js - Service to handle the login of a user into the system
define(
    'Account.Login.ServiceController',
    [
        'ServiceController',
        'Account.Model'
    ],
    function(
        ServiceController,
        AccountModel
    ) {
        'use strict';
        return ServiceController.extend({
            name:'Account.Login.ServiceController',
            options: {
                common: {
                    requireSecure: true
                }
            },
            post: function()
            {
                return AccountModel.login(this.data.email, this.data.password, this.data.redirect)
            }
        });
    }
);
```

Step 2: Set Up the Dev Tools

With the Vinson release of SuiteCommerce Advanced, the dev tools automatically generates the required .ss service file. You must perform the following to successfully deploy your new services:

- Include your new [Module].ServiceController in the ssp_libraries.js file.
- Signal the dev tools to auto-generate your service and associate it with your new Service Controller.

To set up your new service controller:

You must edit the distro.json file to include your new [Module].ServiceController in the ssp_libraries.js file when you deploy your site.

1. Open the distro.json file in the root directory of the SuiteCommerce Advanced source code.
2. Create a new dependency within the `Ssp-libraries` object for your new custom Service Controller. For example, `Account.Login.ServiceController`:

```
"ssp-libraries": {
  "entryPoint": "SCA",
  "dependencies": [
    "Account.Login.ServiceController",
    ...
  ],
}
```

3. Save the distro.json file.

To auto-generate your [Module].Service.ss file:

You must edit the ns.package.json file of the module associated with the service. This file signals the dev tools to auto-generate your [Module].Service.ss file and associate it with a specific service controller.

1. Open the ns.package.json file located in the module containing your new, custom Service Controller.
2. In the `autogenerated-services` object, define the new .ss file and associate it with the appropriate Service Controller.

```
,  "autogenerated-services":
{
  "Account.Login.Service.ss" : "Account.Login.ServiceController"
}
```



Note: The `autogenerated-services` object may not exist in your implementation. If not, create the object and add the .ss files and Service Controller associations. Each .ss file must associate with one Service Controller.

3. Save the ns.package.json file.
4. Use the dev tools to deploy your services to your site.

The dev tools will create `Account.Login.Service.ss` and associate it with `Account.Login.ServiceController.js`.

Step 3: Set Up Custom Validations (Optional)

As an option, you can extend `ServiceController.Validations` to create any custom validations for your services beyond those included by default. To set up custom validations, perform the following:

- Create a custom Validations file and add custom validation logic.
- Edit the distro.json file to include this new file as a dependency.

To create a custom Validations file:

1. In the SspLibraries folder, create a new file titled `ServiceController.Extend.Validations.js`.
2. Define this file and extend `ServiceController.Validations`.
3. Add your custom validation logic.

The following example depicts a typical custom validations file. This file extends ServiceController.Validations and contains custom logic to check validations.

```
define(
  'ServiceController.Extend.Validations'
, [
  'underscore'
, 'ServiceController.Validations'
]
, function (
  _
, ServiceControllerValidations
)
{
  'use strict';

  _.extend(ServiceControllerValidations, {
    checkValidationGeneric: function()
  {
    // validation logic here...
  }
});
});
```

- Set up the `options` object in any [Module].ServiceController using these validations. For example:

```
, options: {
  common: {
    checkValidationGeneric: true
  }
}
```

To set up the dependency in distro.json:

- Open the distro.json file in the root directory of the SuiteCommerce Advanced source code.
- Create a new dependency within the `Ssp-libraries` object for your new custom Service Controller. For example:

```
"ssp-libraries": {
  "entryPoint": "SCA",
  "dependencies": [
    "ServiceController.Extend.Validations",
    ...
  ],
}
```

- Save the distro.json file and deploy using the dev tools.

Create a Service to Handle HTTP Requests (Pre-Vinson)

The following example shows the required components of a service in pre-Vinson implementations of SuiteCommerce Advanced.

```

function service (request)
{
    'use strict';

    var Application = require('Application');

    try
    {
        var method = request.getMethod();

        var QuestionsAndAnswersModel = require('QuestionsAndAnswers.Model');

        switch (method)
        {
            case 'GET':

                var page = request.getParameter('page');
                ,   result = QuestionsAndAnswersModel.search(page);

                Application.sendContent(result,{cache: response.CACHE_DURATION_LONG});
                break;

            case 'POST':

                var data = JSON.parse(request.getBody() || '{}')
                ,   result = QuestionsAndAnswersModel.create(data)

                Application.sendContent(result, {'status': 201});

                break;

            default:

                Application.sendError(methodNotAllowedError);
        }
    }
    catch (e)
    {
        Application.sendError(e);
    }
}

```

To create a custom service for a pre-Vinson implementation of SCA, you must perform the following:

- Step 1: Create a Reference to the Application Module
- Step 2: Define a try/catch Block
- Step 3: Create a Reference to the Backend Model
- Step 4: Define a Switch Statement to Handle the HTTP Request

Step 1: Create a Reference to the Application Module

The following code enables the service to access methods defined in the Application module.

```
var Application = require('Application');
```

The Application module contains HTTP methods the service uses to send data to the frontend application and return any errors. This module is define in the `ssp_libraries.js` file. See [The `ssp_libraries.js` File](#).



Note: Services in SuiteCommerce Advanced do not include the HTTP response as a parameter. Services return a JSON object using the `sendContent` method defined in the Application module.

Step 2: Define a try/catch Block

The body of a service must be included within a try/catch block. This ensures that any errors that occur are handled correctly. These errors are passed back to the frontend application and are handled by the ErrorManagement module.

Step 3: Create a Reference to the Backend Model

```
var QuestionsAndAnswersModel = require('QuestionsAndAnswers.Model');
```

This statement enables the service to call methods defined in the backend model.

Step 4: Define a Switch Statement to Handle the HTTP Request

The main task of a service is to specify how to handle the HTTP action specified by the request. This is generally performed by a `switch` statement which contains a `case` statement for each HTTP method supported by the service.

```
switch (method)
{
    case 'GET':
        var page = request.getParameter('page');
        ,   result = QuestionsAndAnswersModel.search(page);

        Application.sendContent(result,{cache: response.CACHE_DURATION_LONG});

        break;

    case 'POST':
        var data = JSON.parse(request.getBody() || '{}')
        ,   result = ProductReview.create(data)

        Application.sendContent(result, {'status': 201});

        break;

    default:
        Application.sendError(methodNotAllowedError);
}
```

The `switch` statement also contains a `default` statement to handle errors related to the HTTP request. This statement returns an error which is sent back to the frontend application and handled by the ErrorManagement module.

Using Pre-Vinson Services with Vinson Release or Later

Services included with SuiteCommerce Advanced prior to the Vinson software release contain all the logic necessary to function with Vinson release and later.

The Vinson release of SuiteCommerce Advanced ports existing, unmodified services of previous releases. Therefore, any unmodified services using pre-Vinson architecture will function with Vinson without requiring any changes to the code. However, if you made any customizations to pre-Vinson services, you can still use your existing customizations with the Vinson release of SCA. This requires editing the appropriate ns.package.json file to use the existing service (.ss file).

With the Vinson release of SuiteCommerce Advanced, each module's ns.package.json file contains the **autogenerated-services** object. This object includes a list of .ss files to automatically generate when you deploy your site plus their associated Service Controllers. The following example displays the ns.package.json file for the Account module:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ],
    "ssp-libraries": [
      "SuiteScript/*.js"
    ],
    "autogenerated-services": {
      "Account.ForgotPassword.Service.ss" : "Account.ForgotPassword.ServiceController",
      "Account.Login.Service.ss" : "Account.Login.ServiceController",
      "Account.Register.Service.ss" : "Account.Register.ServiceController",
      "Account.RegisterAsGuest.Service.ss" : "Account.RegisterAsGuest.ServiceController",
      "Account.ResetPassword.Service.ss" : "Account.ResetPassword.ServiceController"
    }
  }
}
```

If you have customized any services using pre-Vinson architecture, you can add code to the **services** array of the appropriate ns.package.json file, forcing the dev tools to use existing .ss files of the same name instead of automatically generating ones.

To use pre-Vinson services with Vinson release or later:

1. Open the ns.package.json file located in the module containing the service file you want to use.
2. Add the following code to the file:

```
, "services": [
  "SuiteScript/*.ss"
]
```



Note: If a custom service (.ss file) exists in the module that bears the same name as one listed in the **autogenerated-services** object, this code prevents the dev tools from automatically generating that .ss file. The dev tools will display a warning, informing you that the service will not be automatically generated because of the custom service with the same name. This warning is displayed as information only and does not require any action.

3. Save the ns.package.json file.
4. Use the dev tools to deploy your services to your site.

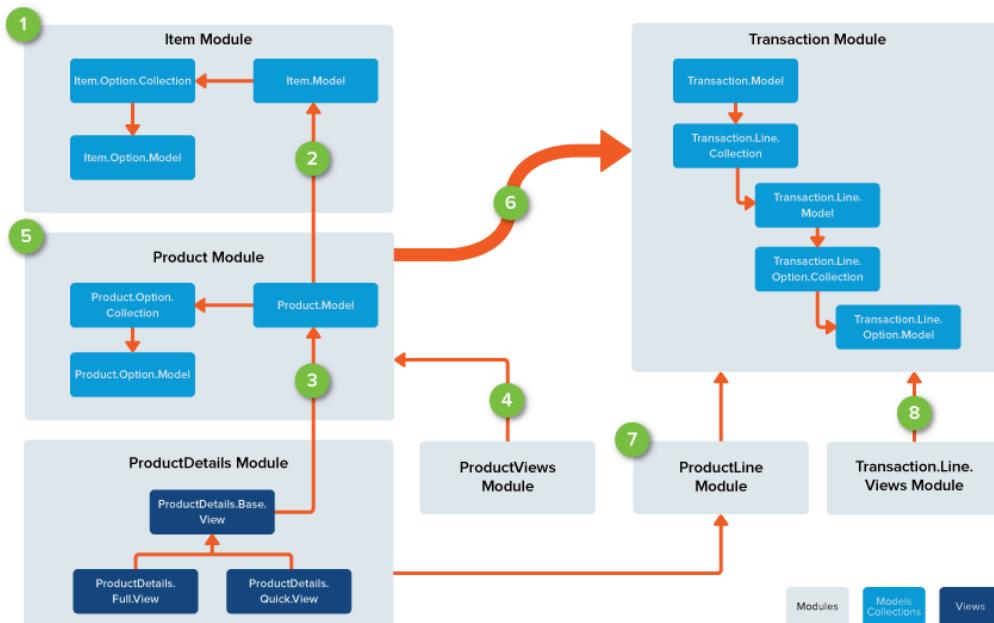
Product Details Page Architecture

ⓘ Applies to: SuiteCommerce Advanced

This section applies to the **Elbrus** release of SuiteCommerce Advanced and later. The following architecture is not backwards compatible with previous versions of SuiteCommerce Advanced.

The product details page (PDP) provides shoppers with detailed information about a product and lets shoppers add items to their cart. To ensure a positive user experience, the PDP needs to create intuitive interactions and return information from the server quickly and efficiently. The architecture of the PDP is designed to accommodate these challenges. Communication of item details is accomplished through the Items and Product modules. This unifies the data structure governing order management.

The following diagram details the architecture behind the PDP and how user selections interact with NetSuite.



- Logic in the Item module retrieves read-only data from NetSuite. Item.Model retrieves data about an item via the Search API. This is a client-side, read-only instance of an item record in NetSuite. Item.Option receives read-only information about an option associated with the item. Item.Option.Collection is a collection of these possible options.

ⓘ Note: The Item.Model is the read-only data from NetSuite used by the Product.Model to validate the user selection. Providing a read-only version of the Item ensures that the original backend data cannot be accidentally overridden.

- Product.Model contains an Item.Model associated with an item to display in the PDP. Product.Option.Model contains editable instances of the related Item.Option models, and Product.Option.Collection is a collection of these options.
- The views and child views in the ProductDetails module use the data in the Product module to render the PDP. ProductDetails.Base.View contains the abstract view from which the Full PDP and Quick View views inherit.
- The views and child views in the ProductViews module use the data in the Product module to render price and item options in the PDP.



Note: To enhance extensibility, SCA uses two different item option templates, one for the PDP and another for search results (Facets).

5. When the user interacts with the PDP to select options, Product.Model validates the data against the Item.Model that each Product.Model contains. This is why the Product.Model contains a Item.Model and why the Item.Model is a client-side representation of the backend record.
6. When the user chooses valid options and clicks Add To Cart, the following conversions occur:
 - Product.Option.Model gets converted into Transaction.Line.Option.Model
 - Product.Option.Collection gets converted to Transaction.Line.Option.Collection
 - Product.Model gets converted into a Transaction.Line.Model
 The Transaction.Model contains a collection of transaction lines, which each include an item and selected options. Transaction.Line.Option.Model contains one of the selected option, and Transaction.Line.Option.Collection is a collection of all selected options.
7. The ProductLine module contains views that are shared between the PDP, Checkout, and My Account applications.
8. The Transaction module uses views in the Transaction.Line.Views module to render transaction details in the cart and in other modules.



Note: Item.KeyMapping.js is part of the Item Module. This file contains mapping definitions of what is returned by the Search API. For example, if you want to set the name of items for data stored in a custom field instead of the default **Display Name** field, you extend the mapping in the Item.KeyMapping file, as opposed to customizing each instance across the entire code base.

Configuration

To configure the Product Details Page, you must have the SuiteCommerce Configuration bundle installed. Refer to the correct section for details on configurable properties as described below.

To configure the Product Details Page:

1. Select the domain to configure at Setup > SuiteCommerce Advanced > Configuration.
2. In the SuiteCommerce Configuration record, navigate to the Shopping Catalog tab and the appropriate subtab:
 - [Item Options Subtab](#) – configure how item options appear in the Product Details Page and in My Account transactions pages.
 - [Product Details Information Subtab](#) – configure extra fields to appear on your Product Details Page.
 - [Multi-Image Option Subtab](#) – configure multiple images in the Product Details Page based on multiple option selections.
3. Save the SuiteCommerce Configuration record to apply changes to your site.

Code Examples

Add an Item to the Cart

The following example of a custom model adds an item to the cart. Product.Model contains the item via the **fetch** method. Based on the internal ID of the item, this code then sets the quantity and item options and adds the product to the cart via the LiveOrderModel.

```
define('AddToCart',
[
    'Product.Model',
    'LiveOrder.Model'
],
function(
    ProductModel
    , LiveOrderModel
)
{
    var my_item_internal_id = '40'
    , quantity = 10
    , product = new ProductModel()
    // load the item.
    product.getItem().fetch({
        data: {id: my_item_internal_id}
    }).then(function (){
        // set quantity
        product.set('quantity', quantity);
        // set options
        product.setOption('option_id', 'value');
        // add to cart
        LiveOrderModel.getInstance().addProduct(product);
    });
});
```

Validate an Item for Purchase

All shopping, wishlist, and quote item validation is centralized in the Product.Model. You can validate per field and developers can override the default validation per field.

The following customization uses a generic `areAttributesValid` method to validate per field. In this case `options` and `quantity`, as opposed to validating the entire set. This method uses a pre-set default validation to validate the specified fields. The `generateOptionalExtraValidation` method overrides the default validation for the any fields. In this case, `quantity`. If the values are not valid, the code introduces an alert message. Otherwise, the code adds the product to the cart.

 **Note:** The default validation is set in Product.Model.js.

```
define('ValidateAddToCart'
,
[
    'LiveOrder.Model'
]
,
function (
    LiveOrderModel
)
{
    function generateOptionalExtraValidation() {
        return {
            'quantity': function(){}
        }
    }
    var product = obtainSomeProduct(/*...*/);
    var cart = LiveOrderModel.getInstance();
    if (!product.areAttributesValid(['options','quantity'], this.generateOptionalExtraValidation()))
}
```

```

    {
        alert('Invalid Product!');
    }
    else
    {
        cart.addProduct(product);
    }
});
```

Add Item to Wishlist

The following customization gets new item data from the product and the correct product list. It then adds the product line to the list and alerts the user upon successful validation.

```

define('ValidateForWishList'
,
[
    'ProductList.Item.Model'
]
,
function (
    ProductListItemModel
)
{
    var getNewItemData = function (product, productList)
    {
        var product_list_line = ProductListItemModel.createFromProduct(product);
        product_list_line.set('productList', {
            id: productList.get('internalid')
            , owner: productList.get('owner').id
        });
        return product_list_line;
    };

    var doAddProductToList = function (product, productList, dontShowMessage)
    {
        var product_list_line_to_save = getNewItemData(product, productList);
        product_list_line_to_save.save(null, {
            validate: false
            , success: function (product_list_line_to_save_saved)
            {
                alert('Product Added!');
            }
        });
    };
    var product = getSomeProduct(/*...*/);
    ,   productList = getSomeProduct(/*...*/);
    doAddProductToList(product, productList);
});
```

Get Options

The following customization returns all valid options for an item or matrix item. It then determines which item options are valid. Only valid options will render as selectable options in the browser.

```
define('ReadProductOptions'
```

```

, [
  'underscore'
]
, function (
  -
)
{
  var showFirstOptionWithValidValues = function (product)
  {
    //Collection of product.options
    var options_to_render = product.get('options');
    options_to_render.each(function (option)
    {
      //each option is a Model
      if (option.get('isMatrixDimension'))
      {
        var valid_values_for_selected_option = product.getValidValuesForOption(option);
        _.each(option.get('values'), function (value)
        {
          value.set('isAvailable', _.contains(valid_values_for_selected_option, value.get('label'))));
        });
      };
      var first_valid_option = options_to_render.find(function (option)
      {
        return _.some(option.get('values'), function (value) { return value.get('isAvailable'); });
      });
      var selected_option = product.get('options').findWhere({cartOptionId: first_valid_option.get('cartOptionId')});
      //Note that we are always backbone Model and the value of the set option contains all details of the set
      //value (label and internalid)
      alert('First Option with value values: ' + first_valid_option.get('cartOptionId') + ' set value: '
+ selected_option.get('value').label);
    });
    showFirstOptionWithValidValues(getSomeIPProduct(/*...*/));
  });
}

```

Overview

 **Applies to:** SuiteCommerce Advanced

This section explains how to upgrade your SuiteCommerce applications and apply patches to your site. See the following topics for more information:

- [Patches](#) – The patches in this section provide updates for specific issues with your SuiteCommerce web store.
- [Upgrade SuiteCommerce to SuiteCommerce Advanced](#) – This section explains how to upgrade our site from SuiteCommerce to SuiteCommerce Advanced.
- [Update SuiteCommerce Advanced](#) – This section explains how to migrate your SuiteCommerce Advanced application to the latest release.

Patches

ⓘ Applies to: SuiteCommerce Advanced

The following table lists available issue patches. Click on the issue title for information about how to implement the patch.

Patches are either provided as example JavaScript, SuiteScript, Sass, or configuration files that you must implement as custom modules or as .patch files.

Issue	Release	Description
Standard Promotion with Inline Discount and Rate as Percentage Not Updating the Amount in Checkout	Vinson Mont Blanc Denali	This patch corrects a problem in some Vinson, Mont Blanc, and Denali implementations where the amount in checkout does not update when applying a standard promotion with an inline discount and setting rate as percentage.
Incorrect Value for Shipping Estimate Occurs in Shopping Cart	Elbrus	This patch corrects a problem in some Elbrus implementations where the shipping estimate contains incorrect values before the actual value is shown.
Page With URL Fragments Redirects Too Many Times	Elbrus Kilimanjaro	This patch corrects a problem where a page with one or more URL fragments that return an ERR_TOO_MANY_REDIRECTS instead of showing a 404 error or the Product Details page.
See Complete List of Stores Link on Store Locator Page Does Not Show Store List	Kilimanjaro Vinson Elbrus	This patch corrects a problem where the See Complete List of Stores link on the Store Locator page reloads the page but does not show the complete list of stores.
Quantity Pricing Displayed in Web Store Even When "Require Login for Pricing" is Enabled	Kilimanjaro Vinson Elbrus	This patch corrects a problem where quantity pricing displays in the web store even when Require Login for Pricing is enabled.
Edited Shipping Address on the Review Your Order Page is Not Showing Changes	Aconcagua Kilimanjaro	This patch corrects a problem where the Review Your Order page is not showing changes when user edits the shipping address.
Custom Page Title in SMT Does Not Display Correctly	Aconcagua	This patch corrects a problem where creating a title for a custom facets page with Site Management Tools does not display the correct title.
Currencies Change to the Default in the Shopping Application	Kilimanjaro	This patch corrects a problem where customers using multi-subsidiaries experience currencies not changing in the webstore, which continues to use the default currency.
Order Confirmation Page Not Displayed When Using Single Page Checkout and External Payment	Elbrus Vinson	This patch corrects a problem where the order confirmation page is not displayed when using single page checkout and an external payment method.
Incorrect Discounted Amounts on Checkout Summary	Aconcagua Kilimanjaro Elbrus	This patch corrects a problem where Checkout Summary pages display incorrect discounted sub-total amounts and inconsistent strike-through and original amounts.

Issue	Release	Description
DeployDistribution Folder Does Not Include Local Files	Aconcagua R2	This patch corrects a problem preventing local files to deploy to the DeployDistribution folder.
HTML List Styles Do Not Display	Aconcagua Kilimanjaro Elbrus Vinson Some SuiteCommerce Themes	This patch corrects a problem where editing item descriptions on an Item record using and styles does not display as expected in the webstore.
Item Search Displays Incorrect Results	Elbrus	This patch corrects a problem where the application returns incorrect results when entering more than one search term in the item search criteria.
Category Product Lists Return Page Not Found	2018.2	This patch corrects a problem in some implementations that results in a Page Not Found error, instead of returning Commerce Categories based on filtered facet values.
Cannot Test an Extension on a Local Server	2018.2	This patch corrects a problem in some implementations that prevents developers from testing extensions locally.
Secure Shopping for Site Builder Implementations	Site Builder Extensions Premium Vinson Site Builder Extensions Vinson Reference Checkout 2.05 Reference Checkout 2.04 Reference My Account Premium 1.06 Reference My Account 1.06 Reference My Account Premium 1.05 Reference My Account 1.05	This patch allows some Site Builder customers to secure the shopping portion of a web store under an HTTPS domain.
Cannot Scroll Through Long Menu Lists Using iOS	Kilimanjaro, Elbrus, Vinson, Mont Blanc, and Denali	This patch corrects a potential issue where iOS users cannot scroll through a long list of menu items when the list reaches beyond the device screen.
Pages Not Indexed Using Google's Mobile-First Indexing	2018.2, Aconcagua, Kilimanjaro, Elbrus, Vinson, Mont Blanc, and Denali	This patch corrects a problem in which search engines discard content and do not render the page correctly using mobile-first indexing.
Disabling Display of SuiteCommerce Gift Wrap & Message Extension Transaction Line Fields	SuiteCommerce Site Builder Websites Reference ShopFlow Reference Checkout Reference My Account	This section provides procedures for disabling the display of Gift Wrap & Message extension transaction line item fields.

Issue	Release	Description
Users Not Redirected to External Payment System	Vinson	This patch corrects an error where SuiteCommerce Advanced does not correctly send a user to an external payment system.
Invoices Do Not Include the Request for a Return Button	Vinson	For Reference My Account v1.06, this patch adds the Request for a Return Button to Invoice records in My Account.
Incorrect Redirect URL for External Payments	Elbrus	This patch corrects an error where the page for an external payment system fails to properly redirect back to the SuiteCommerce site from which it was launched.
	Vinson	
npm Error on Implementations	Denali R2	This patch corrects an issue where <code>npm install</code> fails to run on Denali R2.
Content Appears Incorrectly in a Merchandising Zone	Elbrus Kilimanjaro	This patch corrects an error when content incorrectly shows up in a merchandizing zone.
Reference My Account Generates Error on Load	Reference My Account version 1.04	This patch corrects an error where My Account fails to load if there are 1000+ invoices associated for that account.
Error Loading Shopping Page Due to Uncaught TypeError	Elbrus	This patch corrects an error where SuiteCommerce could not correctly parse the percent sign (%) in a URL.
Users Redirected to Checkout Application Instead of Shopping Application	Elbrus and Kilimanjaro	This patch corrects an issue where users in a password protected site were redirected to the Checkout application instead of the Shopping application.
Add to Cart Button Does Not Work If Quality Field Selected	Elbrus	This patch corrects an issue where the Add to Cart button requires two clicks if the focus is on the Quality field.
URLs with Redundant Facets Generated	Elbrus	This patch corrects an issue where URLs contain redundant facets.
Content Flickers or Disappears When Browsing the Product Listing Page	Kilimanjaro and earlier	This patch corrects an error that occurs when the product listing page flickers or disappears, for content added in SMT.
Enabling Google AdWords Causes Error on Login	Vinson	This patch contains a fix for an error that occurs when Google AdWords is enabled.
URL for Commerce Categories Contains Incorrect Delimiters	Elbrus and earlier	This patch contains a fix for URLs to commerce categories that contain incorrect delimiters.
Order Summary for Item-Based Promotions	Elbrus and earlier	This patch corrects an issue where the item totals for item-based promotions in the Order Summary are incorrect.
CSS Error Hides First div Element on Product Details Page	Mont Blanc and Vinson	This patch contains a fix for a CSS error that hides the first <code>div</code> element on the product details page. This <code>div</code> typically includes the product description.
Invoice Terms Not Included In Order Details	pre-Denali	This patch corrects an issue in Reference My Account v1.05 where the order in My Account does not include the invoice terms.
Users Required to Re-enter Credit Card Payment Method Details on Payment Page	Elbrus and Kilimanjaro	This patch corrects an issue where users are required to re-enter credit card payment method details to successfully complete their order.

Issue	Release	Description
Selected Invoice Not Displayed When Making an Invoice Payment	Mont Blanc	This patch corrects an issue where an invoice selected for payment does not appear on the Make a Payment page.
Log In to See Prices Message Appears When Users are Logged In	Elbrus	This patch corrects an issue where logged-in users receive a message directing them to log in to see prices.
Item Record HTML Meta Data Not Appearing on Page Meta Data	Elbrus	This patch corrects an issue where the value of an Item record's Meta Tag HTML field does not appear in the page's meta data.
Delivery Options Not Appearing After Editing the Cart and Re-entering a Shipping Address	SiteBuilder Extensions — Elbrus	This patch corrects an issue on Site Builder Extensions where delivery methods do not appear after editing cart and re-entering the same shipping address.
Order Confirmation and Thank You Page is Blank	Mont Blanc and Vinson	This patch corrects an issue where a Thank You page is not displayed after an order is complete.
Matrix Item Options Not Displaying With Google Tag Manager Enabled	Elbrus	This patch corrects an issue where not all selected options of a matrix item appear in the Product Details Page. This issue only applies to sites implementing Google Tag Manager.
Delivery Methods Not Appearing in One Page Checkout	Mont Blanc, Vinson, Elbrus	This patch corrects an issue where not all delivery methods appear in the One Page Checkout flow after adding a zip code and checking out as a guest.
Mastercard 2-Series BIN Regex Patch	Denali, Mont Blanc, Vinson	This patch is required to include the Mastercard 2-Series BIN regex value for payment method configuration as of the Elbrus release of SuiteCommerce Advanced.
Change Email Address Patch	Denali, Mont Blanc, Vinson, Elbrus	This patch is required to take advantage of the change email address feature available as of the Kilimanjaro release of SuiteCommerce Advanced.
Auto-Apply Promotions for Elbrus	Elbrus	This patch is required for Elbrus to take advantage of the auto-apply promotions features available as of the Kilimanjaro release of SuiteCommerce Advanced.
Duplicate Product Lists in Internet Explorer 11	pre-Denali	Provides instructions on how to modify the ProductList.js file in ShopFlow 1.07 to correct an issue where IE 11 caching causes duplicate Product Lists to display in My Account.
Save for Later Item not Moved to Cart	Mont Blanc	This patch corrects an issue where an error is returned when users set an item as Save for Later and then return to move that item to the cart.
Running Gulp Commands Results in a Syntax Error	Vinson and earlier	This patch corrects an issue where running gulp commands from a command line or terminal results in a syntax error.
Missing Promo Code on Return Request	Mont Blanc	This patch corrects an issue where promo codes applied to the original sales order are not included in the calculations for a return request.
Invoices Page Displays Incorrect Date Sort (pre-Denali)	pre-Denali	This patch corrects an issue in Reference My Account v1.05, where invoices are displayed out of order.

Issue	Release	Description
PayPal Payments Cause Error at Checkout	Mont Blanc	This patch extends the <code>past()</code> method in the <code>OrderWizard.Module.PaymentMethod.PayPal.js</code> file to correct cases where an error message is displayed when orders are placed using PayPal.
Shopping Cart Not Scrolling (Mobile)	Vinson and earlier	<p>This patch corrects a situation where mobile users cannot scroll their Cart after removing an item from the Saved for Later product list.</p> <p>Follow these instructions to extend the <code>ProductList.DetailsLater.View.js</code> file and override the existing <code>Query.scPush.js</code> file with a new one (provided).</p>
Canonical Tags Populated With Relative Paths	Vinson	This patch modifies commerce category canonical URLs to use absolute paths.
Error When Adding Items to Categories in Site Management Tools	Vinson	This patch contains a fix for an error on a category or subcategory that contains more than 10 items in Site Management Tools.
Item Search API Response Data not Cached	Vinson	<p>This patch extends the <code>getSearchApiParams()</code> method in the <code>Session</code> module to include the <code>pricellevel</code> parameter. This is required to enable caching of the Item Search API Response data.</p> <p>SearchApiCdnCache.zip</p>
Secure Shopping Domains (Elbrus, Vinson, Mont Blanc, and Denali)	Elbrus, Vinson, Mont Blanc, and Denali	<p>This patch enables you to configure your site to maintain a secure browser. The patch for the Denali release of SuiteCommerce Advanced also includes a fix to maintain the identity of the user and cart contents as described in Serversync Touchpoint.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> Note: The original patches provided for this fix have been updated. If you have previously applied a patch for SSL, only the diff between the previous patch and the current one should be applied. </div> <p>Vinson-ssl-V2.patch MontBlanc-ssl-V2.patch Denali-ssl-V2.patch Elbrus-ssl.patch</p>
Secure Shopping Domain (pre-Denali)	pre-Denali	This section guides you through changes required for you to configure your pre-Denali site to maintain a secure browser.
PayPal Address not Retained in Customer Record.	Vinson and earlier	Provides instructions to retain PayPal Address details in the NetSuite customer record.
Login Email Address Appears in the Password Reset URL	Elbrus	This patch corrects a security issue in which the password reset email message includes the original login email address in the password reset URL.
Application Performance Management (APM) Backport See How to Apply .patch Files	Mont Blanc	montblanc-sensors.patch

Issue	Release	Description
Mont Blanc Release of SuiteCommerce Advanced See How to Apply .patch Files	Mont Blanc	Mont_Blanco_2.0_Patch_Files.zip
Serversync Touchpoint	Mont Blanc	This patch enables you to leverage the serversync touchpoint to maintain the identity of the user and cart contents. Serversync.zip

Standard Promotion with Inline Discount and Rate as Percentage Not Updating the Amount in Checkout

 **Applies to:** Vinson | Mont Blanc | Denali

In some implementations of the Vinson, Mont Blanc, and Denali releases of SuiteCommerce Advanced, the amount in checkout is not updating when applying a standard promotion with an inline discount and the rate as percentage.

The following patch corrects a problem in ItemViews.Cell.Navigable.View.js, which is part of the **ItemViews** module. To implement this patch, create a custom module to extend the **prototype** object for the **getContext()** method. You can download the code samples described in this procedure here: [ItemViews.Cell.Navigable.View.Extension@1.0.0-Vinson.zip](#).



Note: Before proceeding, familiarize yourself with [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend ItemViews.Cell.Navigable.View.js

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist.
2. Within this directory, create a custom module with a name similar to the module being customized. For example: `Modules/extensions/ItemViews.Cell.Navigable.View.Extension@1.0.0`.
3. Open your new ItemViews.Cell.Navigable.View.Extension@1.0.0 module and create a subfolder titled **JavaScript**.
For example: `Modules/extensions/ItemViews.Cell.Navigable.View.Extension@1.0.0/JavaScript`
4. In your new JavaScript subdirectory, create a JavaScript file to extend ItemViews.Cell.Navigable.View.js.
Name this file according to best practices.
For example: `ItemViews.Cell.Navigable.View.Extension.js`
5. Open this file and extend the **getContext()** function as shown in the following code snippet.

```
define('ItemViews.Cell.Navigable.View.Extension'
  , [  'ItemViews.SelectedOption.View'
  ]
  ,  function (
    ItemViewsSelectedOptionView
```

```

        )
{
  'use strict';
  _.extend(ItemViews.Cell.Navigable.View.prototype,
  {
    getContext: function ()
    {
      var item = this.model.get('item')
      , line = this.model;

      //@@class ItemViews.Navigable.View.Context
      return {
        //@property {String} itemId
        itemId: item.get('internalid')
        //@property {String} itemName
        , itemName: item.get('_name')
        //@property {String} cellClassName
        , cellClassName: this.options.cellClassName
        //@property {Boolean} isNavigable
        , isNavigable: !!this.options.navigable && !item.get('_isPurchasable')
        //@property {String} rateFormatted
        , rateFormatted: line.get('rate_formatted')
        //@property {String} itemSKU
        , itemSKU: item.get('_sku')
        //@property {Boolean} showOptions
        , showOptions: !(line.get('options') && line.get('options').length)
        //@property {String} itemImageURL
        , itemImageURL: item.get('_thumbnail').url
        //@property {String} itemImageAltText
        , itemImageAltText: item.get('_thumbnail').altimagetext
        //@property {String} itemURLAttributes
        , itemURLAttributes: item.get('_linkAttributes')
        //@property {Number} quantity
        , quantity: line.get('quantity')
        //@property {Boolean} showDetail2Title
        , showDetail2Title: !!this.options.detail2Title
        //@property {String} detail2Title
        , detail2Title: this.options.detail2Title
        //@property {String} detail2
        , detail2: line.get(this.options.detail2)
        //@property {Boolean} showReason
        , showBlockDetail2: !!line.get(this.options.detail2)
        //@property {Boolean} showDetail3Title
        , showDetail3Title: !!this.options.detail3Title
        //@property {String} detail3Title
        , detail3Title: this.options.detail3Title
        //@property {String} detail3
        , detail3: line.get('amount') > line.get('total') ? line.get('total_formatt
ed') : line.get(this.options.detail3)
        //@property {Boolean} showComparePrice
        , showComparePrice: line.get('amount') > line.get('total')
        //@property {String} comparePriceFormatted
        , comparePriceFormatted: line.get('amount_formatted')
      };
    }
}

```

```

    }
});
```

6. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open your new ItemViews.Cell.Navigable.View.Extension@1.0.0 module directory.
2. Create a file in this directory titled ns.package.json.
Modules/extensions/ItemViews.Cell.Navigable.View.Extension@1.0.0/ns.package.json
3. Paste the following code into your new ns.package.json file:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Open the distro.json file. This is located in your root directory.
5. Add your custom modules to the **modules** object.

Your code should look similar to:

```
{
  "name": "SuiteCommerce Advanced Vinson Release",
  "version": "2.0",
  "buildToolsVersion": "1.2.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/ItemViews.Cell.Navigable.View.Extension": "1.0.0",
    "suitecommerce/Account": "2.2.0",
    ...
  }
}
```

This step ensures that the Gulp tasks include your modules when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

6. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, the amount displays on checkout pages when applying standard promotion with inline discount and the rate set as a percentage.

Incorrect Value for Shipping Estimate Occurs in Shopping Cart

 **Applies to:** SuiteCommerce Advanced | Elbrus

In some implementations of the Elbrus release of SuiteCommerce Advanced, a value of \$0.00 appears for the shipping estimate when calculating the actual estimated value for shipping. As a result, this indicates an incorrect value of shipping while SuiteCommerce is calculating the actual shipping value. This error occurs if you use SiteBuilder Extensions to create the site.

The following patch corrects a problem in `Cart.Detailed.View.js`, which is part of the **Cart** module. To implement this patch, create a custom module to extend the **prototype** object for the **estimateTaxShip()** method. You can download the code samples described in this procedure here: [IncorrectValueforEstimateOccursinShoppingCart.zip](#).



Note: Before proceeding, familiarize yourself with [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend `Cart.Detailed.View.js`

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist.
2. Within this directory, create a custom module with a name similar to the module being customized. For example, create `Modules/extensions/Cart@1.0.0`.
3. In your new `Cart@1.0.0` directory, create a subdirectory called `JavaScript`.
For example: `Modules/extensions/Cart@1.0.0/JavaScript`
4. In your new `JavaScript` subdirectory, create a JavaScript file to extend `Cart.Detailed.View.js`. Name this file according to best practices. For example:
`Cart.Detailed.View.Extension.js`
5. Open this file and extend the `estimateTaxShip()` method as shown in the following code snippet.

```
define('Cart.Detailed.View.Extension'
  , [  'Utils'
    , 'underscore'
    , 'Backbone'
  ]
  , function (
  , Utils
  ,
  , Backbone

  , jQuery
  )
{
```

```
'use strict';

_.extend(estimateTaxShip.prototype, function estimateTaxShip(e),
{
    // Build this code in a text editor to make sure that the indents are correct.
    {
        var options = this.$(e.target).serializeObject()
        , address_internalid = options.zip + '-' + options.country + '-null'
        , self = this;

        e.preventDefault();

        var address = {
            internalid: address_internalid
            , zip: options.zip
            , country: options.country
        };

        // Render method should not fire right now, the accordion must close when the promise
        resolves.
        this.model.set('shipaddress', address_internalid, {silent: true});
        this.$('.cart-summary-button-estimate').text(_('Estimating...').translate());
        this.$('.cart-summary-button-estimate').prop('disabled', true);
        var promise = this.saveForm(e);

        if (promise)
        {
            promise.then(function()
            {
                var address_internalid = self.model.get('shipaddress');
                self.model.set('shipaddress', address_internalid);
                self.$('.cart-summary-button-estimate').text(_('Estimate').translate());
                self.$('.cart-summary-button-estimate').prop('disabled', false);
                self.swapEstimationStatus();
            });
        }
    });
});
```

- Save the file.

Step 2: Prepare the Developer Tools For Your Customization

- Open your new Cart@1.0.0 module directory.
- Create a file in this directory and name it ns.package.json.
Modules/extensions/Cart@1.0.0/ns.package.json
- Paste the following code into your new ns.package.json file:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

```
}
```

4. Open the distro.json file. This is located in your root directory.
5. Add your custom module to the **modules** object.

Your code should look similar to the following example:

```
{
    "name": "SuiteCommerce Advanced Elbrus",
    "version": "2.0",
    "buildToolsVersion": "1.3.0",
    "folders": {
        "modules": "Modules",
        "suitecommerceModules": "Modules/suitecommerce",
        "extensionsModules": "Modules/extensions",
        "thirdPartyModules": "Modules/third_parties",
        "distribution": "LocalDistribution",
        "deploy": "DeployDistribution"
    },
    "modules": {
        "extensions/Cart.Extension": "1.0.0",
        "extensions/MyExampleCartExtension1": "1.0.0",
        ...
    }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

6. Add **Cart.Extension** as a dependency to SCA entry point within the **SC.Shopping.Starter** entrypoint of the JavaScript array.

Your Distro.js file should look similar to the following:

```
"tasksConfig": {
//...
"javascript": [
//...
{
    "entryPoint": "SC.Checkout.Starter",
    "exportFile": "checkout.js",
    "dependencies": [
        //...
        "Cart.Extension",
        //...
    ],
    //...
}
]
```

7. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, the estimated shipping does not display a value until the calculation is complete.

Page With URL Fragments Redirects Too Many Times

ⓘ Applies to: SuiteCommerce Advanced | Elbrus | Kilimanjaro

In some implementations of the Elbrus and Kilimanjaro releases of SuiteCommerce Advanced, some pages return ERR_TOO_MANY_REDIRECTS instead of displaying the Product Details page or a 404 error. This error occurs when the URL contains one or more URL fragments. The patch instructions described below correct this problem.

To implement this patch, create a custom module to override `ProductDetails.Router.js`, which is part of the **ProductDetails** module. You can download the code samples described in this procedure here: [PageWithURLFragmentsRedirectsToManyTimes.zip](#).

ⓘ Note: In general, NetSuite best practice is to extend JavaScript using the JavaScript `prototype` object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify a file in a way that you cannot extend, and therefore requires you to use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1: Create the Override File

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist.
2. Within this directory, create a custom module with a name similar to the module being customized. For example, create `Modules/extensions/ProductDetailsExtension@1.0.0`.
3. In your new `ProductDetailsExtension@1.0.0` directory, create a subdirectory called **JavaScript**. For example: `Modules/extensions/ProductDetailsExtension@1.0.0/JavaScript`
4. Copy the following source file and paste into the correct location:

Copy This File:	Place a Copy Here:
<code>Modules/suitecommerce/ProductDetails@X.Y.Z/JavaScript/ ProductDetails.Router.js</code>	<code>Modules/extensions/ProductDetailsExtension@1.0.0/JavaScript</code>

In this example, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

5. Open your new copy of `ProductDetails.Router.js` and locate the following lines:

```

if (api_query.id && item.get('urlcomponent') && SC.ENVIRONMENT.jsEnvironment === 'server')
{
    nsglobal.statusCode = 301;
    nsglobal.location = product.generateURL();
}

if (data.corrections && data.corrections.length > 0)
{
    if (item.get('urlcomponent') && SC.ENVIRONMENT.jsEnvironment === 'server')
    {
        nsglobal.statusCode = 301;
    }
}

```

```

        nsglobal.location = data.corrections[0].url + product.getQuery();
    }
    else
    {
        return Backbone.history.navigate('#' + data.corrections[0].url + product.getQuery(), {trigger: true});
    }
}

```

6. Replace these lines with the following code:

```

if (api_query.id && item.get('urlcomponent') && SC.ENVIRONMENT.jsEnvironment === 'server')
{
    var productUrl = product.generateURL();
    productUrl = productUrl[0] === '/' ? productUrl : '/' + productUrl;

    nsglobal.statusCode = 301;
    nsglobal.location = productUrl;
}

if (data.corrections && data.corrections.length > 0)
{
    if (item.get('urlcomponent') && SC.ENVIRONMENT.jsEnvironment === 'server')
    {
        var url = data.corrections[0].url;
        url = url[0] === '/' ? url : '/' + url;

        nsglobal.statusCode = 301;
        nsglobal.location = url + product.getQuery();
    }
    else
    {
        return Backbone.history.navigate('#' + data.corrections[0].url + product.getQuery(),
{trigger: true});
    }
}

```

7. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open your Modules/extensions/ProductDetailsExtension@1.0.0 module directory.
2. Create a file in this directory and name it ns.package.json.
Modules/extensions/ProductDetailsExtension@1.0.0/ns.package.json
3. Paste the following code in your new ns.package.json file:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  },
  "overrides": {
    "Modules/suitecommerce/ProductDetails@X.Y.Z/JavaScript/ProductDetails.Router.js" : "JavaScript/ProductDetails.Router.js"
  }
}
```

}



Important: You must replace the string X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

4. Open the distro.json file. This file is located in your root directory.
5. Add your custom module to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Kilimanjaro",
  "version": "2.0",
  "isSCA": true,
  "buildToolsVersion": "1.3.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/ProductDetailsExtension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

6. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results. Upon successful deployment, instead of showing ERR_TOO_MANY_REDIRECTS, the browser shows a 404 error or the Product Details page.

See Complete List of Stores Link on Store Locator Page Does Not Show Store List

Applies to: SuiteCommerce Advanced | Kilimanjaro | Vinson | Elbrus

In some implementations of the Kilimanjaro, Vinson and Elbrus releases of SuiteCommerce Advanced, the See Complete List of Stores link on the Store Locator page reloads the page but does not show the complete list of stores. The patch instructions described below correct this problem.

To implement this patch, create a custom module to override the store_locator_main.tpl, store_locator_list.tpl, and store_locator_list_all_store.tpl files, which are part of the **StoreLocator@X.Y.Z** module. In this example, X.Y.Z represents the version of the StoreLocator module in your implementation

of SuiteCommerce Advanced. You can download the code samples described in this procedure here: [StoreLocator.Extension@1.0.0_ElbrusCodeSamples.zip](#)

Step 1: Create the Override Files

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist. For example, create **Modules/extensions**
2. Within this directory, create a custom module with a name similar to the module being customized. For example, create **Modules/extensions/StoreLocator.Extension@1.0.0**.
3. In your new **StoreLocator.Extension@1.0.0** directory, create a subdirectory called **Templates**. For example: **Modules/extensions/StoreLocator.Extension@1.0.0/Templates**
4. Copy the following source file and paste into the correct location:

Copy This File:	Place a Copy Here:
Modules/suitecommerce/StoreLocator@X.Y.Z/Templates/store_locator_main.tpl	Modules/extensions/StoreLocator.Extension@1.0.0/Templates
Modules/suitecommerce/StoreLocator@X.Y.X/Templates/store_locator_list.tpl	Modules/extensions/StoreLocator.Extension@1.0.0/Templates
Modules/suitecommerce/StoreLocator@X.Y.Z/Templates/store_locator_list_all_store.tpl	Modules/extensions/StoreLocator.Extension@1.0.0/Templates

In this example, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

5. Revise your new **store_locator_main.tpl** file following these steps:

- a. Locate the following line:

```
<a data-touchpoint="{{touchpoint}}" data-hashtag="stores/all" href="stores/ all" href="stores/ all">{{translate 'See complete list of stores'}}</a>
```

- b. Replace this line with the following code:

```
<a href="stores/all">{{translate 'See complete list of stores'}}</a>
```

- c. Save the file.

6. Revise your new **store_locator_list.tpl** file following these steps:

- a. Locate the following line:

```
<a data-hashtag="stores/details/{{storeId}}" data-touchpoint="{{touchpoint}}" data-toggle="show-in-pusher">
```

- b. Replace these line with the following code:

```
<a href="stores/details/{{storeId}}" data-toggle="show-in-pusher">
```

- c. Save the file.

7. Revise your new **store_locator_list_all_store.tpl** file following these steps:

- a. Locate the following lines:

```
<a class="store-locator-list-all-store-name" data-touchpoint="{{touchpoint}}" data-toggle="show-in-pusher" data-hashtag="stores/details/{{storeId}}" href="stores/details/{{storeId}}">{{name}}</a>
```

- b. Replace this line with the following code:

```
<a class="store-locator-list-all-store-name" data-toggle="show-in-pusher" href="stores/details/{{storeId}}">{{name}}</a>
```

- c. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open your Modules/extensions/StoreLocator.Extension@1.0.0 module directory.
 2. Create a file in this directory and name it ns.package.json.
- Modules/extensions/StoreLocator.Extension@1.0.0/ns.package.json**
3. Paste the following code in your new ns.package.json file:

```
{
  "gulp": {
    "templates": [
      "Templates/*"
    ],
    "overrides": {
      "suitecommerce/StoreLocator@X.Y.Z/Templates/store_locator_main.tpl": "Templates/store_locator_main.tpl",
      "suitecommerce/StoreLocator@X.Y.Z/Templates/store_locator_list.tpl": "Templates/store_locator_list.tpl",
      "suitecommerce/StoreLocator@X.Y.Z/Templates/store_locator_list_all_store.tpl": "Templates/store_locator_list_all_store.tpl"
    }
  }
}
```



Important: You must replace the string X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

4. Open the distro.json file. This file is located in your root directory.
5. Add your custom module to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "isSCA": true,
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/StoreLocator.Extension": "1.0.0",
    "suitecommerce/Account": "2.3.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

6. Save the `distro.json` file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, Store Locator reloads the page and shows the complete list of stores.

Quantity Pricing Displayed in Web Store Even When “Require Login for Pricing” is Enabled

 **Applies to:** SuiteCommerce Advanced | Kilimanjaro | Vinson | Elbrus

In some implementations of the Kilimanjaro, Vinson, and Elbrus releases of SuiteCommerce Advanced, quantity pricing displays in the web store even when Require Login for Pricing is enabled.

The following patch corrects a problem in `QuantityPricing.View.js`, which is part of the **QuantityPricing** module. To implement this patch, create a custom module to extend the `prototype` object for the `initialize` function. You can download the code samples described in this procedure here: [QuantityPricingAconcaguaCodeSamples..](#)

 **Note:** Before proceeding, familiarize yourself with [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend `QuantityPricing.View.js`

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist.
2. Within this directory, create a custom module with a name similar to the module being customized. For example, create `Modules/extensions/QuantityPricing.Extension@1.0.0`.
3. In your new `QuantityPricing.Extension@1.0.0` directory, create a subdirectory called `JavaScript`. For example: `Modules/extensions/QuantityPricing.Extension@1.0.0/JavaScript`
4. In your new `JavaScript` subdirectory, create a JavaScript file to extend `QuantityPricing.View.js`. Name this file according to best practices.
`QuantityPricing.View.Extension.js`
5. Open this file and extend the `QuantityPricing.View` method as shown in the following code snippet.

```
define(
    'QuantityPricing.View.Extension'
  , [
      'QuantityPricing.View'
    , 'QuantityPricing.Utils'
    , 'SC.Configuration'
```

```

        , 'Profile.Model'
        , 'Backbone'
        , 'underscore'
    ]
    function (
        QuantityPricingView
        , QuantityPricingUtils
        , Configuration
        , ProfileModel
        , Backbone
        ,
        -
    )
{
    'use strict';

    _.extend(QuantityPricingView.prototype,
    {
        initialize: function ()
        {

            this.profileModel = ProfileModel.getInstance();

            this._isEnabled = !this.profileModel.hidePrices();

            this.price_schedule = QuantityPricingUtils.rearrangeQuantitySchedule(this.model.
get('item'), _.isFunction(this.model.getSelectedMatrixChilds) ? this.model.getSelectedMatrixChilds() :
[]);

            this.model.on('change', function ()
            {
                var new_price_schedule =  QuantityPricingUtils.rearrangeQuantitySchedule(this.model.
get('item'), _.isFunction(this.model.getSelectedMatrixChilds) ? this.model.getSelectedMatrixChilds() :
[]);

                if (!_.isEqual(this.price_schedule, new_price_schedule))
                {
                    this.price_schedule = new_price_schedule;
                    this.render();
                }
            }, this);

            this.item_key = this.model.get('item').id + '' + (new Date()).getMilliseconds();
        }
    });
}
);
});

```

6. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open your new QuantityPricing.Extension@1.0.0 module directory.
2. Create a file in this directory and name it ns.package.json.
Modules/extensions/QuantityPricing.Extension@1.0.0/ns.package.json
3. Paste the following code into your new ns.package.json file:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Open the distro.json file. This is located in your root directory.
5. Add your custom module to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Aconcagua",
  "version": "2.0",
  "isSCA": true,
  "buildToolsVersion": "sco-2018.1.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/QuantityPricing.Extension": "1.0.0",
    "suitecommerce/Account": "sco-2018.1.0",
    ...
  }
},
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

6. Add **QuantityPricing.Extension** as a dependency to SCA entry point within the **SC.Shopping.Starter** and **SC.MyAccount.Starter** entrypoints of the JavaScript array.

Your Distro.json file should look similar to the following:

```
"tasksConfig": {
//...
"javascript": [
//...
{
  "entryPoint": "SC.Shopping.Starter",
  "exportFile": "shopping.js",
  "dependencies": [
    //...
    "ProductDetailToQuote",
    "StoreLocator",
    "SC.CCT.Html",
    "QuantityPricing.View.Extension"
  ],
  ...
}
},
```

```

    "entryPoint": "SC.MyAccount.Starter",
    "exportFile": "myaccount.js",
    "dependencies": [
        //...
        "GoogleMap.Configuration",
        "StoreLocatorAccessPoints",
        "SC.CCT.Html",
        "QuantityPricing.View.Extension"
    ],
    //...

```

7. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, quantity pricing will not display in the web store when Require Login for Pricing is enabled.

Edited Shipping Address on the Review Your Order Page is Not Showing Changes

① Applies to: SuiteCommerce Advanced | Aconcagua | Kilimanjaro

In some implementations of the Aconcagua and Kilimanjaro releases of SuiteCommerce Advanced, the Review Your Order page is not showing changes when user edits the shipping address.

To implement this patch, create a custom module to override the OrderWizard.Module.ShowShipments.js file, which is part of the **OrderWizard.Module.Shipmethod@X.Y.Z** module. In this example, X.Y.Z represents the version of the OrderWizard.Module.Shipment module in your implementation of SuiteCommerce Advanced. You can download the code samples described in this procedure here: [OrderWizard.Module.Shipmethod.AconcaguaSampleCode.zip](#)

In general, NetSuite best practice is to extend JavaScript using the JavaScript **prototype** object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify a file in a way that you cannot extend, and therefore requires you to use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1: Create the Override File

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist. For example, create **Modules/extensions**
2. Within this directory, create a custom module with a name similar to the module being customized. For example, create **Modules/extensions/
OrderWizard.Module.Shipmethod.Extension@1.0.0**.
3. In your new **StoreLocator.Extension@1.0.0** directory, create a subdirectory called **JavaScript**. For example: **Modules/extensions/
OrderWizard.Module.Shipmethod.Extension@1.0.0/JavaScript**

- Copy the following source file and paste into the correct location:

Copy This File:	Place a Copy Here:
Modules/suitecommerce/OrderWizard.Module.Shipment@X.Y.Z/OrderWizard.Module.ShowShipments.js	Modules/extensions/OrderWizard.Module.Shipmethod.Extension@1.0.0/JavaScript

In this example, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

- Revise your new OrderWizard.Module.ShowShipments.js file following these steps:

- Replace the existing **initialize** function with this code:

```
, initialize: function () {
    WizardModule.prototype.initialize.apply(this, arguments);

    this.application = this.wizard.application;
    this.options.application = this.wizard.application;

    this.addressSource = this.options.useModelAddresses ?
        this.model.get('addresses') : this.wizard.options.profile.get('addresses');

    BackboneCompositeView.add(this);

    this.wizard.model.on('ismultishiptoUpdated', this.render, this);
    this.wizard.model.on('promocodeUpdated', this.render, this);
    this.address = this.addressSource.get(this.model.get('shipaddress'));
    this.address && this.address.on('change', this.render, this);
}
```

- Add this new **destroy** function after the **initialize** function:

```
, destroy: function destroy()
{
    this.address && this.address.off('change', this.render, this);
    return this._destroy();
}
```

- Save the file.

Step 2: Prepare the Developer Tools For Your Customization

- Open your Modules/extensions/StoreLocator.Extension@1.0.0 module directory.
 - Create a file in this directory and name it ns.package.json.
- Modules/extensions/OrderWizard.Module.Shipmethod.Extension@1.0.0/ns.package.json**
- Paste the following code in your new ns.package.json file:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  },
  "overrides": {
```

```

    "suitecommerce/OrderWizard.Module.Shipmethod@X.Y.Z/JavaScript/OrderWizard.Module.ShowShipments.
js": "JavaScript/OrderWizard.Module.ShowShipments.js"
}
}

```



Important: You must replace the string X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

4. Open the distro.json file. This file is located in your root directory.
5. Add your custom module to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Aconcagua",
  "version": "2.0",
  "isSCA": true,
  "buildToolsVersion": "sc0-2018.1.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/OrderWizard.Module.Shipmethod.Extension": "1.0.0",
    "suitecommerce/Account": "sc0-2018.1.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

6. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
 2. Confirm your results.
- Upon successful deployment, edited shipping address information displays on the Review Your Order page.

Custom Page Title in SMT Does Not Display Correctly

Applies to: SuiteCommerce Advanced | Aconcagua

In some implementations of SuiteCommerce Advanced, creating a title for a custom facets page using Site Management Tools does not display the correct title. In these cases, editing the page name on a custom page results in the page title inheriting the facet path instead of the custom title.

The following patch corrects a problem in CMSAdapter.Page.Router.js, which is part of the **CMSAdapter** module. To implement this patch, create a custom module to extend the **prototype** object for the **getPageForFragment** method. You can download the code samples described in this procedure here: [CustomPageTitlesNotDisplayingCorrectly.zip](#).



Note: Before proceeding, familiarize yourself with [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend CMSAdapter.Page.Router.js

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist.
2. Within this directory, create a custom module with a name similar to the module being customized.
For example: **Modules/extensions/CMSAdapterExtension@1.0.0**.
3. Open your new CMSAdapterExtension@1.0.0 module and create a subfolder titled **JavaScript**.
For example: **Modules/extensions/CMSAdapterExtension@1.0.0/JavaScript**
4. In your new JavaScript subdirectory, create a JavaScript file to extend CMSAdapter.Page.Router.Extend.js.
Name this file according to best practices.
For example: **CMSAdapter.Page.Router.Extend.js**
5. Open this file and extend the **getPageForFragment()** method as shown in the following code snippet.

```
define(
  'CMSAdapter.Page.Router.Extend'
, [
  'CMSAdapter.Page.Router'
, 'underscore'
, 'Backbone'
]
, function (
  CMSAdapterPageRouter

  , Backbone
)
{
  'use strict';

  _.extend(CMSAdapterPageRouter.prototype,
  {
    getPageForFragment: function getPageForFragment(fragment, allPages)
    {
      fragment = fragment || Backbone.history.fragment || '/';
      fragment = fragment.split('?')[0]; //remove options

      var collection = allPages ? this.allPages : this.landingPages;
      return collection.find(function(page)
      {
        return ((page.get('url') === fragment) || (page.get('url') === '/' + fragment));
      });
    }
  }
})
```

```
    });
});
```

6. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open your new CMSAdapterExtension@1.0.0 module directory.

2. Create a file in this directory titled ns.package.json.

Modules/extensions/CMSAdapterExtension@1.0.0/ns.package.json

3. Paste the following code into your new ns.package.json file:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Open the distro.json file. This is located in your root directory.

5. Add your custom module to the **modules** object.

Your code should look similar to:

```
{
  "name": "SuiteCommerce Advanced Kilimanjaro",
  "version": "2.0",
  "isSCA": true,
  "buildToolsVersion": "1.3.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/CMSAdapterExtension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    ...
  }
}
```

This step ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

6. Add **CMSAdapter.Page.Router.Extend** as a dependency to SCA entry point within the **SC.Shopping.Starter**, **SC.Checkout.Starter**, and **SC.MyAccount.Starter** entrypoints of the JavaScript array.

Your distro.js file should look similar to:

```
"tasksConfig": {
//...
"javascript": [
```

```

//...
{
    "entryPoint": "SC.Shopping.Starter",
    "exportFile": "shopping.js",
    "dependencies": [
        //...
        "ProductDetailToQuote",
        "StoreLocator"
        "CMSadapter.Page.Router.Extend"
    ],
    //...

    "entryPoint": "SC.MyAccount.Starter",
    "exportFile": "myaccount.js",
    "dependencies": [
        //...
        "StoreLocatorAccessPoints",
        "StoreLocator",
        "SC.CCT.Html",
        "CMSadapter.Page.Router.Extend"
    ],
    //...

    "entryPoint": "SC.Checkout.Starter",
    "exportFile": "checkout.js",
    "dependencies": [
        //...
        "StoreLocatorAccessPoints",
        "StoreLocator",
        "SC.CCT.Html",
        "CMSadapter.Page.Router.Extend"
    ],
    //...
}

```

7. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, custom page titles should display as expected.

Currencies Change to the Default in the Shopping Application

ⓘ Applies to: SuiteCommerce Advanced | Kilimanjaro

In some implementations of SuiteCommerce Advanced using multi-subsidiaries, users cannot change the currency on the site. The webstore continues to use the default currency.

The following patch corrects the GlobalViews.CurrencySelector.View.js, which is part of the **GlobalViews** module. To implement this patch, create a custom module to extend the **prototype** object for the **getContext()** method. You can download the code samples described in this procedure here: [CurrenciesChangeToDefault-Kilimanjaro.zip](#).

Step 1: Extend Transaction.Line.Views.Cell.Navigable.View.js

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist.
2. Within this directory, create a custom module with a name that is unique but similar to the module being customized.
For example: **Modules/extensions/GlobalViewsExtension@1.0.0**.
3. Open your new GlobalViewsExtension@1.0.0 module and create a subfolder titled **JavaScript**.
For example: **Modules/extensions/GlobalViewsExtension@1.0.0/JavaScript**
4. In your new JavaScript subdirectory, create a JavaScript file to extend **GlobalViews.CurrencySelector.View.js**.
Name this file according to best practices.
For example: **GlobalViews.CurrencySelector.View.Extend.js**
5. Open this file and extend the **getContext()** method as shown in the following code snippet.

```

define(
    'GlobalViews.CurrencySelector.View.Extend'
  [
    'GlobalViews.CurrencySelector.View'
    , 'Backbone'
    , 'underscore'
  ]
  function(
    GlobalViewsCurrencySelectorView
    , Backbone
    , _
  )
{
  'use strict';

  _.extend(GlobalViewsCurrencySelectorView.prototype,
  {
    getContext: function()
    {
      var available_currencies = _.map(SC.ENVIRONMENT.availableCurrencies, function(currency)
      {
        // @class GlobalViews.CurrencySelector.View.Context.Currency
        return {
          // @property {String} code
          code: currency.code
          // @property {String} internalId
          , internalId: currency.internalid
          // @property {String} isDefault
          , isDefault: currency.isdefault
          // @property {String} symbol
          , symbol: currency.symbol
        }
      })
      // @property {Object} currencies
      currencies: available_currencies
    }
  })
}

```

```
// @property {Boolean} symbolPlacement
, symbolPlacement: currency.symbolplacement
// @property {String} displayName
, displayName: currency.title || currency.name
// @property {Boolean} isSelected
, isSelected: SC.getSessionInfo('currency').code === currency.code
};

});

// @class GlobalViews.CurrencySelector.View.Context
return {
    // @property {Boolean} showCurrencySelector
    showCurrencySelector: !(SC.ENVIRONMENT.availableCurrencies && SC.ENVIRONMENT.availableCurrencies.length > 1)
        // @property {Array<GlobalViews.CurrencySelector.View.Context.Currency>} availableCurrencies
        , availableCurrencies: available_currencies || []
        // @property {String} currentCurrencyCode
        , currentCurrencyCode: SC.getSessionInfo('currency').code
        // @property {String} currentCurrencySymbol
        , currentCurrencySymbol: SC.getSessionInfo('currency').symbol
    };
}
);
});
```

- ## 6. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open your new Transaction.Line.Views.Extension@1.0.0 module directory.
 2. Create a file in this directory titled ns.package.json.

Modules/extensions/Transaction.Line.Views.Extension@1.0.0/ns.package.json

3. Paste the following code into your new ns.package.json file:

```
{  
  "gulp": {  
    "javascript": [  
      "JavaScript/*.js"  
    ]  
  }  
}
```

4. Open the distro.json file. This is located in your root directory.
 5. Add your custom module to the `modules` object.

Your code should look similar to:

```
{  
    "name": "SuiteCommerce Advanced Kilimanjaro",  
    "version": "2.0",  
    "isSCA": true,  
    "buildToolsVersion": "1.3.1",  
    "folders": {  
        "modules": "Modules".
```

```

    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/GlobalViewsExtension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    ...
  }
}

```

This step ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

6. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, Checkout Summary pages should display correct original and discounted sub-total amounts. Strike-through amounts also display correctly.

Order Confirmation Page Not Displayed When Using Single Page Checkout and External Payment

 **Applies to:** Vinson | Elbrus

In some implementations of the Vinson and Elbrus releases of SuiteCommerce Advanced, the Order Confirmation Page is not displayed when using one page checkout with an external payment system.

To implement this patch, create a custom module to override SC.Checkout.Configuration.Steps.OPC.js, which is part of the **CheckoutApplication@2.2.0** module. You can download the code samples described in this procedure here: [CheckoutApplication.Extension@1.0.0 VinsonCodeSamples.zip](#)

 **Note:** In general, NetSuite best practice is to extend JavaScript using the JavaScript **prototype** object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify a file in a way that you cannot extend, and therefore requires you to use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1: Create the Override File

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist.
2. Within the extension directory, create a directory for a new custom module with a name similar to the module being customized.

For example, create `Modules/extensions/CheckoutApplication.Extension@1.0.0`.

3. In your new `CheckoutApplicationExtension@1.0.0` module directory, create a directory called `JavaScript`.

For example: `Modules/extensions/CheckoutApplication.Extension@1.0.0/JavaScript`

4. Copy the following source file and paste into the correct location:

Copy This File:	Place a Copy Here:
<code>Modules/suitecommerce/CheckoutApplication@X.Y.Z/JavaScript/ SC.Checkout.Configuration.Steps.OPC.js</code>	<code>Modules/extensions/CheckoutApplication.Extension@1.0.0/JavaScript</code>

In this example, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

5. Open your new copy of `SC.Checkout.Configuration.Steps.OPC.js` and locate the following line:

```
, [OrderWizardModulePaymentMethodSelector, {record_type:'salesorder', preventDefault: true}]
```

6. Replace this line with the following code:

```
, [OrderWizardModulePaymentMethodSelector, {record_type:'salesorder', prevent_default: true}]
```

7. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open your `Modules/extensions/CheckoutApplication.Extension@1.0.0` directory.
2. Create a file in this directory and name it `ns.package.json`.

`Modules/extensions/CheckoutApplication.Extension@1.0.0/ns.package.json`

3. Paste the following code in your new `ns.package.json` file:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  },
  "overrides": {
    "suitecommerce/CheckoutApplication@2.2.0/JavaScript/SC.Checkout.Configuration.Steps.OPC.js" :
    "JavaScript/SC.Checkout.Configuration.Steps.OPC.js"
  }
}
```



Important: You must replace the string X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

4. Open the `distro.json` file. This file is located in your root directory.
5. Add your custom module to the `modules` object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Vinson Release",
```

```

    "version": "2.0",
    "isSCA": true,
    "buildToolsVersion": "1.2.1",
    "folders": {
        "modules": "Modules",
        "suitecommerceModules": "Modules/suitecommerce",
        "extensionsModules": "Modules/extensions",
        "thirdPartyModules": "Modules/third_parties",
        "distribution": "LocalDistribution",
        "deploy": "DeployDistribution"
    },
    "modules": {
        "extensions/CheckoutApplication.Extension": "1.0.0",
        "suitecommerce/Account": "2.2.0",
        ...
    }
}

```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

6. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, the Order Confirmation Page is displayed after completing an order when using one page checkout with an external payment system.

Incorrect Discounted Amounts on Checkout Summary

Applies to: SuiteCommerce Advanced | Aconcagua | Kilimanjaro | Elbrus

In some implementations of SuiteCommerce Advanced, Checkout Summary pages display incorrect discounted sub-total amounts and inconsistent strike-through and original amounts. These issues occur on Subtotal and Items to Ship sections of the checkout flow. The patch instructions described below correct this problem.

The following patch corrects a problem in Transaction.Line.Views.Cell.Navigable.View.js, which is part of the **Transaction.Line.Views** module. To implement this patch, create a custom module to extend the **prototype** object for the **getContext()** method. You can download the code samples described in this procedure here: [IncorrectDiscountedAmountsOnCheckout.zip](#).

Step 1: Extend **Transaction.Line.Views.Cell.Navigable.View.js**

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist.
2. Within this directory, create a custom module with a name that is unique but similar to the module being customized.

For example: `Modules/extensions/Transaction.Line.Views.Extension@1.0.0`.

3. Open your new `Transaction.Line.Views.Extension@1.0.0` module and create a subfolder titled `JavaScript`.

For example: `Modules/extensions/Transaction.Line.Views.Extension@1.0.0/JavaScript`

4. In your new JavaScript subdirectory, create a JavaScript file to extend `Transaction.Line.Views.Cell.Navigable.View.js`.

Name this file according to best practices.

For example: `Transaction.Line.Views.Cell.Navigable.View.Extend.js`

5. Open this file and extend the `getContext()` method as shown in the following code snippet.

```
define('Transaction.Line.Views.Cell.Navigable.View.Extend'
, [
    'Transaction.Line.Views.Cell.Navigable.View'
, 'underscore'
, 'Backbone'
]
, function (
    TransactionLineViewsCellNavigableView
)
{
    'use strict';

    _.extend(TransactionLineViewsOptionsSelectedView.prototype,
    {
        getContext: function ()
        {
            var item = this.model.get('item')
            , line = this.model;

            //@@class Transaction.Line.Views.Navigable.View.Context
            return {
                //@@property {Transaction.Line.Model} model
                model: this.model
                //@@property {String} itemId
                , itemId: item.get('internalid')
                //@@property {String} itemName
                , itemName: item.get('_name')
                //@@property {String} cellClassName
                , cellClassName: this.options.cellClassName
                //@@property {Boolean} isNavigable
                , isNavigable: !!this.options.navigable && !item.get('_isPurchasable')
                //@@property {String} rateFormatted
                , rateFormatted: line.get('rate_formatted')
                //@@property {Boolean} showOptions
                , showOptions: !(line.get('options') && line.get('options').length)
                //@@property {String} itemURLAttributes
                , itemURLAttributes: line.getFullLink({quantity:null,location:null,fulfillment
Choice:null})
                //@@property {Number} quantity
                , quantity: line.get('quantity')
                //@@property {Boolean} showDetail2Title
            }
        }
    }
});
```

```

        , showDetail2Title: !!this.options.detail2Title
        //@property {String} detail2Title
        , detail2Title: this.options.detail2Title
        //@property {String} detail2
        , detail2: line.get(this.options.detail2)
        //@property {Boolean} showBlockDetail2
        , showBlockDetail2: !!line.get(this.options.detail2)
        //@property {Boolean} showDetail3Title
        , showDetail3Title: !!this.options.detail3Title
        //@property {String} detail3Title
        , detail3Title: this.options.detail3Title
        //@property {String} detail3
        , detail3: line.get('amount') > line.get('total') ? line.get('total_formatted')
        : line.get(this.options.detail3)
        //@property {Boolean} showComparePrice
        , showComparePrice: line.get('amount') > line.get('total')
        //@property {String} comparePriceFormatted
        , comparePriceFormatted: line.get('amount_formatted')
        // @property {ImageContainer} thumbnail
        , thumbnail: this.model.getThumbnail()
        // @property {Boolean} isFreeGift
        , isFreeGift: line.get('free_gift') === true
    );
}
});
);
});

```

- Save the file.

Step 2: Prepare the Developer Tools For Your Customization

- Open your new Transaction.Line.Views.Extension@1.0.0 module directory.
- Create a file in this directory titled ns.package.json.
Modules/extensions/Transaction.Line.Views.Extension@1.0.0/ns.package.json
- Paste the following code into your new ns.package.json file:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

- Open the distro.json file. This is located in your root directory.
- Add your custom module to the **modules** object.

Your code should look similar to:

```
{
  "name": "SuiteCommerce Advanced Aconcagua",
  "version": "2.0",
  "isSCA": true,
  "buildToolsVersion": "aconcaguaR2",
  "folders": {
    "src": {
      "files": [
        "index.html"
      ],
      "folders": [
        "components"
      ]
    }
  }
}
```

```

    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
},
"modules": {
    "extensions/Transaction.Line.Views.Extension": "1.0.0",
    "suitecommerce/Account": "aconcaguaR2",
    ...
}

```

This step ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

- Add `Transaction.Line.Views.Cell.Navigable.View.Extend` as a dependency to SCA entry point within the `SC.Checkout.Starter` entrypoint of the JavaScript array.

Your distro.js file should look similar to:

```

"tasksConfig": {
//...
"javascript": [
//...
{
    "entryPoint": "SC.Checkout.Starter",
    "exportFile": "checkout.js",
    "dependencies": [
//...
        "StoreLocator",
        "SC.CCT.Html",
        "Transaction.Line.Views.Cell.Navigable.View.Extend"
    ],
    ...
}
]
}

```

- Save the distro.json file.

Step 3: Test and Deploy Your Customization

- Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
- Confirm your results.

Upon successful deployment, Checkout Summary pages should display correct original and discounted sub-total amounts. Strike-through amounts also display correctly.

DeployDistribution Folder Does Not Include Local Files

ⓘ Applies to: SuiteCommerce Advanced | Aconcagua

In some implementations of the Aconcagua R2 release of SuiteCommerce Advanced, local files are not deploying to the DeployDistribution folder after running the gulp deploy command using the developer

tools. As a consequence, accessing Local SSP pages result in a Page Not Found error. This problem occurs because the R2 release of Aconcagua did not include three important files. This patch adds these files to your implementation.

To implement this patch, you must add the following files to your source and deploy:

- CheckoutApplication@aconcaguaR2\Internal\ index-local.ssp
- MyAccountApplication@aconcaguaR2\Internal\ index-local.ssp
- ShoppingApplication@aconcaguaR2\Internal\ index-local.ssp

Because these files do not exist, you cannot use traditional best practices, such as extending or overriding. This instruction simply requires adding three files, renaming them to match the applicable module, and deploying.

Step 1: Download the Patch Files

1. Download the .zip file containing the three local.ssp files here: [LocalFilesNotDeploying-AconcaguaR2.zip](#).

This zip file includes the following files:

- checkout-local.ssp
- my_account-local.ssp
- shopping-local.ssp

2. Extract the contents of this file to any location locally.

3. For the Checkout application:

- a. Copy checkout-local.ssp to the CheckoutApplication@aconcaguaR2/Internal folder.

If the Internal folder does not exist, create it.

- b. Rename checkout-local.ssp to **index-local.ssp**.

4. For the MyAccount application:

- a. Copy myaccount-local.ssp to the MyAccountApplication@aconcaguaR2/Internal folder.

If the Internal folder does not exist, create it.

- b. Rename myaccount-local.ssp to **index-local.ssp**.

5. For the Shopping application:

- a. Copy shopping-local.ssp to the ShoppingApplication@aconcaguaR2/Internal folder.

If the Internal folder does not exist, create it.

- b. Rename shopping-local.ssp to **index-local.ssp**.

Your source directory structure should look like this:

```
...
CheckoutApplication@aconcaguaR2/
    Internal/
        index-local.ssp
...
MyAccountApplication@aconcaguaR2/
    Internal/
        index-local.ssp
...
ShoppingApplication@aconcaguaR2/
```

```

Internal/
index-local.ssp
....
```

Step 2: Deploy Your Customization

1. Deploy your source code customizations to your NetSuite account and test the functionality. See [Deploy to NetSuite](#) for details.



Note: Since this patch adds SuiteScript files, changes are not visible in your local environment. SuiteScript files run on the server and must be deployed to NetSuite to take effect.

2. Confirm your results.

Upon successful deployment, local files should exist in your DeployDistribution folder.

HTML List Styles Do Not Display

ⓘ Applies to: SuiteCommerce Advanced | SuiteCommerce Web Stores | Aconcagua | Kilimanjaro | Elbrus | Vinson

In some implementations of SuiteCommerce and SuiteCommerce Advanced, editing item descriptions on an Item record (**Web Store** tab, **Detailed Description** field) using `` and `` styles does not display as expected in the webstore. This patch corrects this issue. However, the correct patching instructions depend on your implementation, as described below:

Implementation	Patching Instructions
SCA Kilimanjaro	HTML List Styles Do Not Display (Kilimanjaro)
SCA Vinson, Elbrus	HTML List Styles Do Not Display (Elbrus and Vinson)
Any theme created using the 2018.1 release of the SuiteCommerce Base Theme bundle	HTML List Style Does Not Display (Theme)

HTML List Styles Do Not Display (Kilimanjaro)

ⓘ Applies to: SuiteCommerce Advanced | Kilimanjaro

Apply the changes in this patch using the override method, as described in this section. The following table lists the files to be overridden and their locations in the source code:

Module Location	File
BaseSassStyles	_base.scss
ProductDetails	_product-details-information.scss

You can download the code samples described in this procedure here: [HtmlListStyleNotDisplaying-Kilimanjaro.zip](#)

Step 1: Create the Override File

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist.
2. Within this directory, create a custom module with a name that is unique, but similar to each module being customized. Give these new modules a version of **1.0.0**.

Your implementation should look like this:

- **Modules/extensions/BaseSassStylesExtension@1.0.0**
- **Modules/extensions/ProductDetailsExtension@1.0.0**

3. Open your new BaseSassStylesExtension@1.0.0 module and create nested subfolders titled **Sass** and **base**.

For example: **Modules/extensions/BaseSassStylesExtension@1.0.0/Sass/base**

4. Open your new ProductDetailsExtension@1.0.0 module and create a subfolder titled **Sass**.

For example: **Modules/extensions/ProductDetailsExtension@1.0.0/Sass**

5. Copy the following source files and paste into the correct location, where X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced:

Copy This File:	Place a Copy Here:
Modules/suitecommerce/BaseSassStyles@X.Y.Z/Sass/base/_base.scss	Modules/extensions/BaseSassStyles@1.0.0/Sass/base
Modules/suitecommerce/ProductDetails@X.Y.Z/Sass/_product-details-information.scss	Modules/extensions/ProductDetails@1.0.0/Sass

6. Open your new copy of **_base.scss** and add the following lines to the end of file:

```
.cms-content ul, .cms-content ol {
    margin-left: $sc-margin-lv4;
}

.cms-content ul li {
    list-style: circle;
}

.cms-content ol li {
    list-style: decimal;
}
```

7. Save the file.
8. Open your new copy of **_product-details-information.scss** and add the following lines to the end of file:

```
.product-details-information-tab-content-container ul, .product-details-information-tab-content-container ol {
    margin-left: $sc-margin-lv4;
}

.product-details-information-tab-content-container ul li {
    list-style: outside;
```

```

    }

.product-details-information-tab-content-container ol li {
    list-style: decimal;
}

```

- Save the file.

Step 2: Prepare the Developer Tools For Your Customization

- Open your Modules/extensions/BaseSassStyles@1.0.0 module.
- Create a file in this directory titled **ns.package.json**.
For example: `Modules/extensions/BaseSassStyles@1.0.0/ns.package.json`
- Paste the following code in your new ns.package.json file:

 **Important:** You must replace the string X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

```
{
  "gulp": {
    "sass": [
      "Sass/**/*.scss"
    ]
  },
  "overrides": {
    "suitecommerce/BaseSassStyles@X.Y.Z/Sass/base/_base.scss" : "Sass/base/_base.scss"
  }
}
```

- Save the file.
- Open your Modules/extensions/ProductDetails@1.0.0 module.
- Create a file in this directory titled **ns.package.json**.
For example: `Modules/extensions/ProductDetails@1.0.0/ns.package.json`
- Paste the following code in your new ns.package.json file:

 **Important:** You must replace the string X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

```
{
  "gulp": {
    "sass": [
      "Sass/**/*.scss"
    ]
  },
  "overrides": {
    "suitecommerce/ProductDetails@X.Y.Z/Sass/_product-details-information.scss" : "Sass/_product-details-information.scss"
  }
}
```

- Open the distro.json file. This file is located in your root directory.

- Add both custom modules to the `modules` object.

Your code should look similar to:

```
{
  "name": "SuiteCommerce Advanced Kilimanjaro",
  "version": "2.0",
  "isSCA": true,
  "buildToolsVersion": "1.3.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/BaseSassStylesExtension": "1.0.0",
    "extensions/ProductDetailsExtension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    ...
  }
}
```

This step ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

- Save the `distro.json` file.

Step 3: Test and Deploy Your Customization

- Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
- Confirm your results.

Upon successful deployment, the HTML list (`` and ``) styles should display on your webstore.

HTML List Styles Do Not Display (Elbrus and Vinson)

i Applies to: SuiteCommerce Advanced | Elbrus | Vinson

Apply the changes in this patch using the override method, as described in this section. The following table lists the files to be overridden and their locations in the source code:

Module Location	File
BaseSassStyles	_base.scss
ProductDetails	_product-details-information.scss

You can download the code samples described in this procedure here: [HtmlListStylesNotDisplaying-Elbrus-Vinson.zip](#)

Step 1: Create the Override File

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist.
2. Within this directory, create a custom module with a name that is unique, but similar to each module being customized. Give these new modules a version of **1.0.0**.

Your implementation should look like this:

- **Modules/extensions/BaseSassStylesExtension@1.0.0**
- **Modules/extensions/ProductDetailsExtension@1.0.0**

3. Create a subfolder for each new module titled **Sass**.

Your file structure should look like:

- **Modules/extensions/BaseSassStylesExtension@1.0.0/Sass**
- **Modules/extensions/ProductDetailsExtension@1.0.0/Sass**

4. Copy the applicable source files and paste into the correct location, where X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced:

Copy This File:	Place a Copy Here:
Modules/suitecommerce/BaseSassStyles@X.Y.Z/Sass/_base.scss	Modules/extensions/BaseSassStyles@1.0.0/Sass
Modules/suitecommerce/ProductDetails@X.Y.Z/Sass/_product-details-information.scss	Modules/extensions/ProductDetails@1.0.0/Sass

5. Open your new copy of _base.scss and add the following lines to the end of file:

```
.cms-content ul, .cms-content ol {
    margin-left: $sc-margin-lv4;
}

.cms-content ul li {
    list-style: circle;
}

.cms-content ol li {
    list-style: decimal;
}
```

6. Save the file.

7. Open your new copy of _product-details-information.scss and add the following lines to the end of file:

```
.product-details-information-tab-content-container ul, .product-details-information-tab-content-container ol {
    margin-left: $sc-margin-lv4;
}

.product-details-information-tab-content-container ul li {
    list-style: outside;
}

.product-details-information-tab-content-container ol li {
    list-style: decimal;
}
```

8. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Create an ns.package.json file for each custom module.
 - a. Open your Modules/extensions/BaseSassStyles@1.0.0 module.
 - b. Create a file in this directory titled **ns.package.json**.
For example: **Modules/extensions/BaseSassStyles@1.0.0/ns.package.json**
 - c. Paste the following code in your new ns.package.json file:

 **Important:** You must replace the string X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

```
{
  "gulp": {
    "sass": [
      "Sass/**/*.scss"
    ]
  },
  "overrides": {
    "suitecommerce/BaseSassStyles@X.Y.Z/Sass/_base.scss" : "Sass/_base.scss"
  }
}
```

- d. Save the file.
- e. Open your Modules/extensions/ProductDetails@1.0.0 module.
- f. Create a file in this directory titled **ns.package.json**.
For example: **Modules/extensions/ProductDetails@1.0.0/ns.package.json**
- g. Paste the following code in your new ns.package.json file:

 **Important:** You must replace the string X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

```
{
  "gulp": {
    "sass": [
      "Sass/**/*.scss"
    ]
  },
  "overrides": {
    "suitecommerce/ProductDetails@X.Y.Z/Sass/_product-details-information.scss" : "Sass/_product
-details-information.scss"
  }
}
```

- h. Save the file.
2. Edit your distro.json file.

This step ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

- a. Open the distro.json file. This file is located in your root directory.

- b. Add both custom modules to the `modules` object.

Your code should look similar to:

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/BaseSassStylesExtension": "1.0.0",
    "extensions/ProductDetailsExtension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    ...
  }
}
```

- c. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, the HTML list (`` and ``) styles should display on your webstore.

HTML List Style Does Not Display (Theme)

 **Applies to:** SuiteCommerce Advanced | Aconcagua | SuiteCommerce Web Stores

The instructions in the section apply to you if you are implementing themes and you are implementing a theme created using the SuiteCommerce Base Theme (2018.1 release) as a baseline.

This topic explains how to correct the theme manually using the theme developer tools. Because you cannot patch a theme and you cannot make changes to a published theme, you have three options, depending on your implementation.

- If the theme exhibiting this behavior is a published theme and you want to make the changes yourself, you must create a new theme using the theme developer tools, applying the new code as described in this section.
- If the theme is owned by you, you can make changes to your theme using the theme developer tools as described in this section.
- If this is a published theme and you do not want to make this change manually, you must wait until the theme publisher provides an update or use a different theme.

This fix requires adding some code to the theme's existing Sass files. This step is the same if you own the theme or not. If you do not own the theme, the developer tools prompt you to create a new one when you deploy your changes.

You can download the code samples described in this procedure here: [HtmlListStyleNotDisplaying-Themes.zip](#)



Note: Any themes created using the SuiteCommerce Base Theme (2018.2 release and later) already include this fix.

Step 1: Fetch the Active Theme

To make the change, the first step is to fetch the theme files from NetSuite and place them in your local developer environment. This step assumes that you have already set up your local developer environment and are using the applicable theme developer tools. See [Developer Environment](#) for more details.

To fetch the theme:

1. Ensure that the theme exhibiting this behavior is activated on a domain associated with your site. See the help topic [Manage Themes and Extensions](#) for details.
2. Open a command line or terminal.
3. Access the top-level theme development directory you created when you downloaded the developer tools.
4. Enter the following command:
`gulp theme:fetch`
5. Follow the prompts to access your account, website, and the domain that currently has the theme activated.

Step 2: Customize the Theme

The next step is to customize your theme by editing two .scss files.

To customize the theme:

1. In your theme development source code, navigate to the following location:

```
<THEME_DEV_TOOLS_DIRECTORY>/Workspace/<THEME_NAME>/Modules/
BaseSassStyles@X.Y.Z/Sass/base
```

In this example, X.Y.Z represents the version of the BaseSassStyles module in your theme.

2. Open _base.scss and add the following lines to the end of file:

```
.cms-content ul, .cms-content ol {
    margin-left: $sc-margin-lv4;
}

.cms-content ul li {
    list-style: circle;
}
```

```
.cms-content ol li {
    list-style: decimal;
}
```

3. Save the file.
4. Navigate to the following location:

`.. /Modules/ProductDetails@X.Y.Z/Sass`

In this example, X.Y.Z represents the version of the ProductDetails module in your theme.

5. Open `_product-details-information.scss` and add the following lines to the end of file:

```
.product-details-information-tab-content-container ul,
.product-details-information-tab-content-container ol {
    margin-left: $sc-margin-lv4;
}

.product-details-information-tab-content-container ul li {
    list-style: outside;
}

.product-details-information-tab-content-container ol li {
    list-style: decimal;
}
```

6. Save the file.

Step 3: Deploy Your Changes

This step involves deploying your changes to a NetSuite account and activating on a domain. The steps are the same, regardless of the state of your theme, but the developer tools perform different actions, as described below:

- If you own the theme, the developer tools prompt you to include a new revision number. You can later update any bundles that include the theme for publication.
- If you do not own the theme, the developer tools prompt you to create a new one. This process results in a new theme that is based on the original, but includes your customizations. These changes do not affect the existing theme.



Important: After you deploy the theme, you must activate the update to apply your changes to a domain.

To deploy changes:

1. In your local developer environment, open a command line or terminal and access the top-level theme directory.
 2. Run the following command:
`gulp theme:deploy`
 3. Follow the prompts to confirm your account, website, and other information about your theme, as required.
- If you are customizing a published theme, the developer tools prompt you to create a new theme.

4. After deployment is completed, activate the theme on a domain. See the help topic [Manage Themes and Extensions](#) for details.
5. Confirm your results.

Upon successful deployment, the HTML list (`` and ``) styles should display on your webstore.

For more information:

- To use theme developer tools, see [Theme Developer Tools](#).
- To create bundles for a theme, see [Theme and Extension SuiteApps](#).

Item Search Displays Incorrect Results

 **Applies to:** SuiteCommerce Advanced | Elbrus

In some implementations of the Elbrus release of SuiteCommerce Advanced, the application returns incorrect results when entering more than one term in the search criteria.

The following patch corrects a problem in `ItemsSearcher.View.js`, which is part of the **ItemsSearcher** module. To implement this patch, create a custom module to change the `defaultOptions.highlightFirst` property. You can download the code samples described in this procedure here: [ElbrusItemSearchResultsIncorrect.zip](#).



Note: Before proceeding, familiarize yourself with [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend `ItemsSearcher.View.js`

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist.
2. Within this directory, create a custom module with a name similar to the module being customized. For example: `Modules/extensions/ItemsSearcherExtension@1.0.0`.
3. In your new `ItemsSearcherExtension@1.0.0` module, create a subdirectory called **JavaScript**. For example: `Modules/extensions/ItemsSearcherExtension/JavaScript`
4. In your new JavaScript subdirectory, create a JavaScript file. Name this file according to best practices. For example: `ItemsSearcher.View.Extend.js`
5. Open this file and extend `ItemsSearcher.View.js` to customize the `highlightFirst` property from `true` to `false`, as shown in the following code snippet.

```
define(
  'ItemsSearcher.View.Extend'
, [
  'ItemsSearcher.View'
]
, function (
  ItemsSearcherView
)
{
}
```

```
'use strict';

ItemSearcherView.prototype.defaultOptions.highlightFirst = false

});
```

6. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open your new ItemsSearcherExtension@1.0.0 module directory.
2. Create a file in this directory and name it ns.package.json.

Modules/extensions/ItemsSearcherExtension@1.0.0/ns.package.json

3. Paste the following code into your new ns.package.json file:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Open the distro.json file. This is located in your root directory.
5. Add your custom module to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/ItemsSearcherExtension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

6. Add **ItemsSearcher.View.Extend** as a dependency to SCA entry point within the **SC.Shopping.Starter** entrypoint of the JavaScript array.

Your distro.js file should look similar to the following:

```
"tasksConfig": {
```

```
//...
"javascript": [
    //...
    {
        "entryPoint": "SC.Shopping.Starter",
        "exportFile": "shopping.js",
        "dependencies": [
            //...
            "StoreLocator",
            "SC.CCT.Html",
            "ItemsSearcher.View.Extend"
        ],
        //...
    }
]
```

7. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, item search strings should display correct results.

Category Product Lists Return Page Not Found

 **Applies to:** SuiteCommerce Advanced | 2018.2

In some implementations of the 2018.2 release of SuiteCommerce Advanced, category product lists return Page Not Found when viewing a table or list.

The following patch corrects a problem in Facets.Translator.js, which is part of the **Facets** module. To implement this patch, create a custom module to extend the **prototype** object for the **FacetsTranslator** function. You can download the code samples described in this procedure here: [CategoryPageNotFound.zip](#).

 **Note:** Before proceeding, familiarize yourself with [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend the `Facets.Translator.js` File

1. Create an **extensions** directory to store your custom module. Depending on your implementation, this directory might already exist.
2. Create a custom module with a name similar to the module being customized.
For example, `Modules/extensions/FacetsExtension@1.0.0`.
3. In your new `FacetsExtension@1.0.0` directory, create a subdirectory called **JavaScript**.
For example: `Modules/extensions/FacetsExtension@1.0.0/JavaScript`
4. In your new `JavaScript` subdirectory, create a `JavaScript` file to extend `Facets.Translator.js`.

Name this file according to best practices.

For example:

`Facets.Translator.Extension.js`

5. Open this file and extend the `FacetsTranslator()` method as shown in the following code snippet:

```
define('Facets.Translator.Extension'
, [ 'Facets.Translator'
, 'underscore'
, 'jQuery'
, 'SC.Configuration'
, 'Utils'
, 'UrlHelper'
]
, function (
  FacetsTranslator
,
  -
,
  jQuery
,
  Configuration
)
{
  'use strict';

  _.extend(FacetsTranslator.prototype,
  {
    FacetsTranslator (facets, options, configuration, category)
    {
      // Enforces new
      if (!(this instanceof FacetsTranslator))
      {
        return new FacetsTranslator(facets, options, configuration, category);
      }

      // Facets go Here
      this.facets = [];

      // Other options like page, view, etc. goes here
      this.options = {};

      // This is an object that must contain a fallbackUrl and a lists of facet configurations
      this.configuration = configuration || default_config;

      // Get the facets that are in the sitesettings but not in the config.
      // These facets will get a default config (max, behavior, etc.) - Facets.Translator
      // Include facet aliases to be considered as a possible route
      var facets_data = Configuration.get('siteSettings.facetfield')
      , facets_to_include = [];

      _.each(facets_data, function(facet)
      {
        if (facet.facetfieldid !== 'commercecategory')
        {
          facets_to_include.push(facet.facetfieldid);
        }
      });
    }
  });
}
```

```

// If the facet has an urlcomponent defined, then add it to the possible values list.
facet.urlcomponent && facets_to_include.push(facet.urlcomponent);

// Finally, include URL Component Aliases...
_.each(facet.urlcomponentaliases, function(facet_alias)
{
    facets_to_include.push(facet_alias.urlcomponent);
});
});

facets_to_include = _.union(facets_to_include, _.pluck(Configuration.get('facets'), 'id'));
facets_to_include = _.uniq(facets_to_include);

this.facetsToInclude = facets_to_include;

this.isCategoryPage = !!category;

if (_.isBoolean(category) && category)
{
    var index = facets.length
    , facetsToInclude = this.facetsToInclude.slice(0)
    , facets_delimiter = this.configuration.facetDelimiters.betweenDifferentFacets
    , facet_value_delimiter = this.configuration.facetDelimiters.betweenFacetNameAndValue;

    facetsToInclude.push(this.configuration.facetDelimiters.betweenFacetsAndOptions);

    _.each(facetsToInclude, function(facetname)
    {
        facetname = facets_delimiter + facetname + facet_value_delimiter;
        var i = facets.lastIndexOf(facetname);

        if (i !== -1 && i < index)
        {
            index = i;
        }
    });
}

var categoryUrl = facets.substring(0, index);

facets = facets.substring(index);

if (categoryUrl[0] !== '/')
{
    categoryUrl = '/' + categoryUrl;
}

if (categoryUrl[categoryUrl.length - 1] === '/')
{
    categoryUrl = categoryUrl.substring(0, categoryUrl.length - 1);
}

if (facets && facets[0] === facets_delimiter)
{
    facets = facets.substring(1, facets.length);
}

```

```

        }

        this.categoryUrl = categoryUrl;
    }
    else if (_isString(category))
    {
        this.categoryUrl = category;
    }

    // We cast on top of the passed in parameters.
    if (facets && options)
    {
        this.facets = facets;
        this.options = options;
    }
    else if (_isString(facets))
    {
        // It's a url
        this.parseUrl(facets);
    }
    else if (facets)
    {
        // It's an API option object
        this.parseOptions(facets);
    }
}
});
});
});

```

6. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open your new FacetsExtension@1.0.0 module directory.
2. Create a file in this directory and name it ns.package.json.
Modules/extensions/FacetsExtension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Open the distro.json file. This is located in your root directory.
5. Add your custom modules to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced 2018.2.0 (sc-2018.2.0)",
  "version": "2.0",
  "isSCA": true,
```

```

"buildToolsVersion": "sc-2018.2.0",
"folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
},
"modules": {
    "extensions/FacetsExtension": "1.0.0",
    "suitecommerce/Account": "sc-2018.2.0",
    // ...
}

```

This step ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

- Add `Facets.Translator.Extension` as a dependency to SCA entry point within the `SC.Shopping.Starter` entrypoint of the JavaScript array.

Your distro.json file should look similar to:

```

"tasksConfig": {
//...
"javascript": [
    //...
    {
        "entryPoint": "SC.Shopping.Starter",
        "exportFile": "shopping.js",
        "dependencies": [
            //...
            "Instrumentation.Cart",
            "SiteSearch",
            "Facets.Translator.Extension"
        ],
        //...
    }
]
}

```

- Save the distro.json file.

Step 3: Test and Deploy Your Customization

- Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
- Confirm your results.

Upon successful deployment, Commerce Categories filter based on the selected facet values.

Cannot Test an Extension on a Local Server

ⓘ Applies to: SuiteCommerce Advanced | 2018.2

In some 2018.2 implementations of SuiteCommerce Advanced, developers might experience the following console error when testing an extension on a local server: **TypeError**:

`SC.addExtensionModule` is not a function. The patch instructions described below correct this error.

To implement this patch, you must create two custom modules to override different instances of index-local.ssp. These files are located in the **Internals** directory of the following modules (where X.Y.Z represents the version of the module in your implementation):

- CheckoutApplication@X.Y.Z
- MyAccountApplication@X.Y.Z

You can download the code samples described in this procedure here: [RunExtensionsOnALocalServer.zip](#).

i Note: In general, NetSuite best practice is to extend JavaScript using the JavaScript **prototype** object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify a file in a way that you cannot extend, and therefore requires you to use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1: Create the Override Files

1. Create an **extensions** directory to store each of your new custom modules. Depending on your implementation, this directory might already exist.

Create two custom modules and give them names similar to the modules being customized. For example, create the following files:

`Modules/extensions/CheckoutApplicationExtension@1.0.0`

`Modules/extensions/MyAccountApplicationExtension@1.0.0`

2. In each module, create a subdirectory called **Internal**.

For example:

■ `Modules/extensions/CheckoutApplicationExtension@1.0.0/Internal`

■ `Modules/extensions/MyAccountApplicationExtension@1.0.0/Internal`

3. Copy the following source files and paste into the correct location:

Copy This File:	Place a Copy Here:
<code>Modules/suitecommerce/CheckoutApplication@X.Y.Z/Internal/index-local.ssp</code>	<code>Modules/extensions/CheckoutApplicationExtension@1.0.0/Internal</code>
<code>Modules/suitecommerce/MyAccountApplication@X.Y.Z/Internal/index-local.ssp</code>	<code>Modules/extensions/MyAccountApplicationExtension@1.0.0/Internal</code>

In this example, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

4. For each new copy of index-local.ssp, add the following lines after the `loadscript()` function:

```
SC.extensionModules = [];
SC.addExtensionModule = function addExtensionModule(appModuleName) {
    SC.extensionModules.push(appModuleName);
};
```

Each file should look similar to the following example:

```
//...
```

```

function loadScript (url)
{
    'use strict';
    var reference_tag = document.getElementsByTagName("script")[0];
    var new_script_tag = document.createElement("script");
    new_script_tag.src = url;
    new_script_tag.type = "text/javascript";
    new_script_tag.async = false;
    reference_tag.parentNode.insertBefore(new_script_tag, reference_tag);
}

SC.extensionModules = [];
SC.addExtensionModule = function addExtensionModule(appModuleName) {
    SC.extensionModules.push(appModuleName);
};
//...

```

5. Save each file.

Step 2: Prepare the Developer Tools For Your Customization

1. Create a file called ns.package.json in each of your custom modules.

For example:

- Modules/extensions/CheckoutApplicationExtension@1.0.0/ns.package.json
- Modules/extensions/MyAccountApplicationExtension@1.0.0/ns.package.json

2. Paste the following code in each ns.package.json file, as applicable:

CheckoutApplication

```
{
  "gulp": {
    "ssp-files": [
      "Internal/*.ssp"
    ]
  },
  "overrides": {
    "suitecommerce/CheckoutApplication@X.Y.Z/Internal/index-local.ssp" : "Internal/index-local.ssp"
  }
}
```

MyAccountApplication

```
{
  "gulp": {
    "ssp-files": [
      "Internal/*.ssp"
    ]
  },
  "overrides": {
    "suitecommerce/MyAccountApplication@X.Y.Z/Internal/index-local.ssp" : "Internal/index-local.ssp"
  }
}
```



Important: You must replace the string X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

3. Open the distro.json file. This file is located in your root directory.
4. Add both custom modules to the `modules` object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced 2018.2.0",
  "version": "2.0",
  "isSCA": true,
  "buildToolsVersion": "1.3.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/CheckoutApplicationExtension": "1.0.0",
    "extensions/MyAccountApplicationExtension": "1.0.0",
    "suitecommerce/Account": "sc-2018.2.0",
    ...
  }
}
```

This step ensures that the Gulp tasks include your module when you deploy your code to NetSuite.

In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

5. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Deploy your changes to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.



Note: Because this patch modifies an SSP library file, changes are not visible in your local environment until you first deploy the customizations to NetSuite.

2. Confirm your results.

Upon successful deployment, you should be able to test your extensions on a local server.

Secure Shopping for Site Builder Implementations

Applies to: Site Builder

A secure shopping domain creates an encrypted connection between your web server and your customer's web browser. You can use an SSL certificate to secure the shopping portion of your web store under an HTTPS domain on your Site Builder sites. HTTPS is currently the industry standard for ecommerce services and consumers prefer seeing the secure icon in their browser address bar.



Important: If setting up Secure Shopping on a Site Builder site, you must set up the shopping portion of your site to use an HTTPS domain. You cannot set up a single secure domain for Shopping and Checkout.

Some Site Builder implementations must be patched to take advantage of this functionality. Refer to the following table to determine if you need this patch:

Implementation	Instructions
Site Builder	No patch required. See the help topic Secure Web Store for instructions on setting up Secure Shopping for your site.
<ul style="list-style-type: none"> ■ Site Builder Extensions Premium Kilimanjaro ■ Site Builder Extensions Kilimanjaro ■ Site Builder Extensions Premium Elbrus ■ Site Builder Extensions Elbrus 	No patch required. See the help topic Secure Web Store for instructions on setting up Secure Shopping for your site.
<ul style="list-style-type: none"> ■ Site Builder Extensions Premium Vinson ■ Site Builder Extensions Vinson 	Patch required. See Secure Shopping for Site Builder Extensions (Vinson) for instructions.
<ul style="list-style-type: none"> ■ Reference Checkout 2.05 ■ Reference Checkout 2.04 ■ Reference My Account Premium 1.06 ■ Reference My Account 1.06 ■ Reference My Account Premium 1.05 ■ Reference My Account 1.05 	Patch required. See Secure Shopping for Site Builder (Pre-Denali) for instructions.

Secure Shopping for Site Builder Extensions (Vinson)

Applies to: Site Builder

This topic explains how to patch the following implementations of Site Builder to take advantage of Secure Shopping domains:

- SiteBuilder Extensions Vinson
- SiteBuilder Extensions Premium Vinson



Note: Site Builder sites implementing the Kilimanjaro or Elbrus releases of Site Builder Extensions and Extensions Premium already include the ability to create a secure Shopping domain by default and do not require a patch. Denali and Mont Blanc Site Builder implementations cannot set up secure Shopping domains.



Note: To patch pre-Denali reference implementations for Site Builder, see [Secure Shopping for Site Builder \(Pre-Denali\)](#).

Follow these steps to set up a secure Shopping domain for your Site Builder Extensions implementation:

1. Configure a Secure Shopping Domain

2. Apply Patch to Source Files
3. Deploy Files

To leverage a secure shopping domain in your implementation of Site Builder Extensions, you need to introduce these changes as custom modules and deploy using the Core SCA developer tools. This patch is highly technical and involves customizing multiple files.

Configure a Secure Shopping Domain

Before you can use an SSL certificate to secure the shopping portion of your web store under an HTTPS domain, you must set up a secure shopping domain in NetSuite. See the help topic [Secure Web Store](#) for more information.

When setting up a secure domain, be aware of the following:

- Set the **Hosted As** field on the Set Up Domains record to **Secured Web Store**.
- Sites implementing Site Builder can only secure the shopping portion of the web store under a separate HTTPS domain. Site Builder sites cannot implement a single secure domain for both Shopping and Checkout applications.

Sites implementing Site Builder can only secure the **Shopping** portion of the web store as a HTTPS domain. Site Builder sites cannot implement secure shopping as part of one single domain for both Shopping and Checkout.

When you have configured your domain, continue with the next step: [Apply Patch to Source Files](#).

Apply Patch to Source Files

Apply the changes in this patch using the Override method, as described in this section. The following table lists the files to be overridden and their locations in the Vinson source code, where X.Y.Z represents the version of the module in your implementation:

Module	Subdirectory	File
CheckoutApplication@X.Y.Z	/Internal	/index-local.ssp
	/JavaScript	/SC.Checkout.Configuration.js
	/SuiteScript	/checkout.environment.ssp
		/SuiteScript/checkout.ssp
CheckoutSkipLogin@X.Y.Z	/JavaScript	/CheckoutSkipLogin.js
GoogleUniversalAnalytics@X.Y.Z	/JavaScript	/GoogleUniversalAnalytics.js
LiveOrder@X.Y.Z	/SuiteScript	/LiveOrder.Model.js
MyAccountApplication@X.Y.Z	/Internal	/index-local.ssp
	/SuiteScript	/my_account.ssp
		/myaccount.environment.ssp
NavigationHelper@X.Y.Z	/JavaScript	/NavigationHelper.Plugins.DataTouchPoint.js
ProductList@X.Y.Z	/SuiteScript	/ProductList.Model.js

Module	Subdirectory	File
Profile@X.Y.Z	/SuiteScript	/ProductList.Model.js
		/Profile.Model.js
SiteSettings@X.Y.Z	/SuiteScript	/SiteSettings.Model.js
SspLibraries@X.Y.Z	/SuiteScript	/Application.js
		/ServiceController.Validations.js
		/Utils.js
Utilities@X.Y.Z	/JavaScript	/Utils.js
jQueryExtras@X.Y.Z	/JavaScript	/jQuery.ajaxSetup.js
	/JavaScript	/jQuery.ajaxSetup.noLoader.js

Note: In general, NetSuite best practice is to extend JavaScript using the JavaScript **prototype** object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify files in a way that you cannot extend, and therefore requires you to use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Create Override Files

The first step is to create a space to contain your customized files. You set up your new files to override the existing code when you deploy to NetSuite.

To create override files:

1. Open the Modules directory and create a subdirectory titled extensions to maintain your customizations.
2. In your extensions directory, create a new directory for each module according to the table above. Give these new modules a version of **1.0.0**.

For example:

```
Modules/extensions/CheckoutApplication@1.0.0/
Modules/extensions/CheckoutSkipLogin@1.0.0/
Modules/extensions/GoogleUniversalAnalytics@1.0.0/
...
```

3. In each new module directory, create the required subdirectories according to the table above.

For example, the directory structure for your custom CheckoutApplication@1.0.0 module should look similar to the following:

```
Modules/extensions/CheckoutApplication@1.0.0
  /Internal
  /JavaScript
  /SuiteScript
```

4. Locate the files in your current implementation to be overridden according to the table.

5. Make a copy of each file and paste the copy into the applicable subdirectory.

For example, your directory structure for your custom CheckoutApplication@1.0.0 module should look similar to the following:

```
Modules/extensions/CheckoutApplication@1.0.0/
  Internal/
    index-local.ssp
  JavaScript/
    SC.Checkout.Configuration.js
  SuiteScript/
    checkout.environment.ssp
```

6. Repeat this procedure for each file you need to override.

Implement the Patch Changes

The next step is to implement the patch in each new override file.

To implement the patch:

1. Download the .patch file associated with this patch guide here: HTTPS_SiteBuilder-Vinson.zip. This file includes all changes described in this procedure.
2. Open each new file within your extensions directory and add or remove lines (as applicable) according to the .patch file.
See the help topic [How to Apply .patch Files](#) for instructions on how to understand .patch files and apply changes. The following example describes one scenario.

Example

The following code snippet comes from the **Vinson Site Builder Extensions Premium** .patch file:

```
...
diff --git a/Modules/suitecommerce/CheckoutApplication@2.2.0/JavaScript/SC.Checkout.Configuration.js b/Modules/
suitecommerce/CheckoutApplication@2.2.0/JavaScript/SC.Checkout.Configuration.js
index 48cd87d..8dbc7eb 100644
--- a/Modules/suitecommerce/CheckoutApplication@2.2.0/JavaScript/SC.Checkout.Configuration.js
+++ b/Modules/suitecommerce/CheckoutApplication@2.2.0/JavaScript/SC.Checkout.Configuration.js
@@ -62,15 +62,15 @@ define(
  /*
  */

-  currentTouchpoint: _.isSecureDomain() ? 'checkout' : 'viewcart'
+  currentTouchpoint: Utils.isInShopping() ? 'viewcart' : 'checkout'

  , modulesConfig: {
    'ItemDetails': {startRouter: true}
  ...
}
```

In this example, these lines specify that the file you need to customize is **SC.Checkout.Configuration.js**:

```
diff --git a/Modules/suitecommerce/CheckoutApplication@2.2.0/JavaScript/SC.Checkout.Configuration.js b/Modules/
suitecommerce/CheckoutApplication@2.2.0/JavaScript/SC.Checkout.Configuration.js
```

The next lines in the example provide a reference point to locate the code in the file that needs updated. Two sets of @@ symbols specify that the code to be customized is located on line 62 of the file and that 15 lines of code are displayed. The code after this reference is included as a visual reference to help locate the code affected by this change:

```
@@ -62,15 +62,15 @@ define(
  ,*/
```



Important: The line reference (@@) specifies the line in the original source code. If you have made any preexisting modifications to the file prior to this patch, the code needing customized code may be located on a different line.

Any lines prepended with a - symbol indicate code to delete/remove as part of this patch. In this example, you remove the following line from SC.Checkout.Configuration.js:

```
- currentTouchpoint: _.isSecureDomain() ? 'checkout' : 'viewcart'
```

Lines prepended with a + symbol indicate code to add. In this example, you add the following line:

```
+ currentTouchpoint: Utils.isInShopping() ? 'viewcart' : 'checkout'
```

Lines not prepended with a - or + symbol do not require change. See [How to Apply .patch Files](#) for more information on how to apply these changes manually.

Deploy Files

When all customizations are complete, you are ready to deploy your custom files to your Site Builder SSP Application in NetSuite. The following topics explain how to deploy custom files to the Custom implementation:

- [Prepare the Developer Tools For Your Customizations](#)
- [Deploy Your Customizations to NetSuite](#)



Important: The customizations introduced by this patch are extensive. Always test changes in a sandbox account or other test site before deploying to a production account.

Prepare the Developer Tools For Your Customizations

Before you deploy your changes, you must prepare your Vinson developer tools to include your customizations. To do this:

1. Create an ns.package.json file for each new module in your extensions subdirectory.
2. Edit the distro.json file.

To create an ns.package.json file for each new module:

Perform this procedure for each custom module in your extensions directory.

1. Create a new file for each custom module and title it: ns.package.json.

You should end up with one unique ns.package.json file for each custom module.

- Open each ns.package.json file and paste the following code:

The following example includes <placeholders> that you must change within each ns.package.json file.

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ],
    "overrides": {
      "suitecommerce/<MODULE>@<X.Y.Z>/<SUBDIRECTORY>/<EXISTING_FILE>" : "<NEW_SUBDIRECTORY>/<NEW_FILE>.js"
    }
  }
}
```

- Edit the placeholders in the **overrides** object as described below:

- <MODULE> represents the name of each module being changed.
- <X.Y.Z> represents the module version in your implementation.
- <SUBDIRECTORY> represents the subdirectory containing the original file you are overriding.
- <EXISTING_FILE> represents the original file you are overriding.
- <NEW_SUBDIRECTORY> represents the subdirectory according to the table in [Apply Patch to Source Files](#).
- <NEW_FILE> represents the new file according to the table in [Apply Patch to Source Files](#).

- Add the following as applicable:

- Add an **Internal**, **JavaScript**, or **SuiteScript** array (as applicable) to the **gulp** object for each file type in the module.

The table

- Add a path to the **overrides** object for each file in the module.

For example, the ns.package.json file for your custom CheckoutApplication@1.0.0 module should look similar to the following:

```
{
  "gulp": {
    "internal": [
      "Internal/*.ssp"
    ],
    "javascript": [
      "JavaScript/*.js"
    ],
    "ssp-libraries": [
      "SuiteScript/*.ssp"
      "SuiteScript/*.js"
    ]
  },
  "overrides": {
    "suitecommerce/CheckoutApplication@2.2.0>/Internal/index-local.ssp" : "Internal/index-local.ssp",
    "suitecommerce/CheckoutApplication@2.2.0>/JavaScript/SC.Checkout.Configuration.js" : "JavaScript/SC.Checkout.Configuration.js",
    "suitecommerce/CheckoutApplication@2.2.0>/SuiteScript/checkout.environment.ssp" : "SuiteScript/checkout.environment.ssp"
  }
}
```

```

    }
}

}

```

5. Repeat this procedure for each module, following the same basic structure.

To edit the distro.json file:

1. Open the distro.json file. This is located in your root directory.
2. Add each custom module to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Site Builder 2.0",
  "version": "2.0",
  "buildToolsVersion": "1.2.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "suitecommerce/CheckoutApplication": "1.0.0",
    "suitecommerce/CheckoutSkipLogin": "1.0.0",
    "suitecommerce/GoogleUniversalAnalytics": "1.0.0",
    "suitecommerce/LiveOrder": "1.0.0",
    "suitecommerce/MyAccountApplication": "1.0.0",
    "suitecommerce/NavigationHelper": "1.0.0",
    "suitecommerce/ProductList": "1.0.0",
    "suitecommerce/Profile": "1.0.0",
    "suitecommerce/SiteSettings": "1.0.0",
    "suitecommerce/Ssplibraries": "1.0.0",
    "suitecommerce/Utilities": "1.0.0",
    "suitecommerce/jQueryExtras": "1.0.0",
    "suitecommerce/Account": "2.2.0",
    // ...
  }
}
```

This ensures that the Gulp tasks include your modules when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

3. Save the distro.json file.

Deploy Your Customizations to NetSuite

The last step is to deploy your customizations.

To deploy your customizations:

1. Deploy your source code customizations to your NetSuite account (see [Deploy to NetSuite](#)).
2. Confirm your results.

Upon successful deployment, you should be able to take advantage of secure shopping domains on your Site Builder site.

Secure Shopping for Site Builder (Pre-Denali)

 **Applies to:** Site Builder

This topic explains how to patch the following implementations of Site Builder to take advantage of Secure Shopping domains:

- Site Builder Reference Checkout 2.05
- Site Builder Reference Checkout 2.04
- Site Builder Reference My Account 1.06
- Site Builder Reference My Account Premium 1.06
- Site Builder Reference My Account 1.05
- Site Builder Reference My Account Premium 1.05

Follow these steps to allow for secure Shopping domains on your Site Builder site:

1. [Configure a Secure Shopping Domain](#)
2. [Apply the Patch to Source Files](#)
3. [Deploy Files and Configure the Implementation](#)

Configure a Secure Shopping Domain

Before you can use an SSL certificate to secure the shopping portion of your web store under an HTTPS domain, you must set up a secure shopping domain in NetSuite. See the help topic [Secure Web Store](#) for more information.

When setting up a secure domain for a site implementing Site Builder, be aware of the following:

- You must set the **Hosted As** field on the Set Up Domains record to **Secured Web Store**.
- Sites implementing Site Builder can only implement SSL for the Shopping area of their site using a separate, secure shopping domain. Site Builder sites cannot implement a single secure domain for both Shopping and Checkout applications.

When you have configured your domain, continue with the next step: [Apply the Patch to Source Files](#).

Apply the Patch to Source Files

To leverage a secure shopping domain in your pre-Denali implementation of Site Builder, you need to introduce these changes as custom files and deploy to your Custom implementation.

This patch is highly technical and involves customizing multiple files. Before you begin making changes, back up any custom files that already exist. Each .zip file listed in the procedure below includes a .patch file for a specific implementation. Launch the .patch file and introduce changes manually.

For each file that needs updating, go to the SSP Application in NetSuite and copy the Reference files to the same location in the Custom folder:

Documents > Files > File Cabinet > Web Site Hosting Files > Live Hosting Files > SSP Applications

To implement the patch:

1. Download the correct patch .zip file, depending on your implementation.

These files include all changes described in this procedure:

Implementation	Download
Site Builder Reference Checkout 2.0.4 and 2.0.5	HTTPS_SiteBuilder-Reference-Checkout-2-0-4_and_2-0-5.zip
Site Builder Reference My Account and My Account Premium 1.0.5	HTTPS_SiteBuilder-Reference-My-Account.1-0-5.zip
Site Builder Reference My Account and My Account Premium 1.0.6	HTTPS_SiteBuilder-Reference-My-Account-1-0-6.zip

2. Extract the .zip file and open the .patch file you find there.

The .patch file describes each file and subsequent code to be customized.

3. If your implementation includes previous customizations, ensure that each file being patched is located in the custom folder and downloaded to your local environment.
4. If your implementation does not include any previous customizations, perform the following steps:
 - a. Copy the corresponding source file located in the Reference implementation in NetSuite. If a preexisting file exists in the Custom implementation, then a customization has already been implemented on that file in the past. Back up this file and customize it instead.
 - b. Create a local copy for development.
5. Access the root of your local development environment for this implementation.
6. Manually modify your files as instructed in the patch files to add or remove lines, as required.

See [How to Apply .patch Files](#) for instructions on how to understand .patch files and apply changes. The following example describes one scenario.

When you have successfully uploaded your customizations, continue with the next step: [Deploy Files and Configure the Implementation](#).

Patch Update Example

The following code snippet comes from the Site Builder Reference Checkout 2.0.4 .patch file:

```
diff --git a/index-local.ssp b/index-local.ssp
index 3f974de..ac20076 100644
--- a/index-local.ssp
+++ b/index-local.ssp
@@ -30,11 +30,11 @@
     login = true;
 }
else if (
-    (SiteSettings.registration.registrationoptional !== 'T' && !session.isLoggedIn())
-    || (!SC.Configuration.checkout_skip_login && !session.isLoggedIn())
-    || (parameters.is && (parameters.is === 'login' || parameters.is === 'register') && !
session.isLoggedIn())
+    (SiteSettings.registration.registrationoptional !== 'T' && !session.isLoggedIn2())
+    || (!SC.Configuration.checkout_skip_login && !session.isLoggedIn2())
+    || (parameters.is && (parameters.is === 'login' || parameters.is === 'register') && !
session.isLoggedIn2())
```

```

        || (parameters.is && (parameters.is === 'login' || parameters.is === 'register') &&
session.getCustomer().isGuest())
-        || (SC.Configuration.checkout_skip_login && !session.isLoggedIn() && session.isRecognized())
+        || (SC.Configuration.checkout_skip_login && !session.isLoggedIn2() && session.isRecognized())
    )
{
    delete parameters.sitepath;
}

```

In this example, these lines specify that the file you need to customize is **index-local.ssp**:

```
diff --git a/index-local.ssp b/index-local.ssp
```

The next lines in the example provide a reference point to locate the code in the file that needs updated. Two sets of @@ symbols specify that the code to be customized is located on line 30 of the file and that 11 lines of code are displayed. The code after this line reference provides a reference to help locate the area in the file affected by this change:

```

@@ -30,11 +30,11 @@
    login = true;
}
else if (

```



Important: The line reference (@@) specifies the line in the original source code. If you have made any preexisting modifications to the file prior to this patch, the code needing customized code may be located on a different line.

Any lines prepended with a - symbol indicate code to delete/remove as part of this patch. In this example, you remove the following lines from index-local.ssp:

```

-        (SiteSettings.registration.registrationoptional !== 'T' && !session.isLoggedIn())
-        || (!SC.Configuration.checkout_skip_login && !session.isLoggedIn())
-        || (parameters.is && (parameters.is === 'login' || parameters.is === 'register') && !
session.isLoggedIn())

```

Lines prepended with a + symbol indicate code to add:

```

+        (SiteSettings.registration.registrationoptional !== 'T' && !session.isLoggedIn2())
+        || (!SC.Configuration.checkout_skip_login && !session.isLoggedIn2())
+        || (parameters.is && (parameters.is === 'login' || parameters.is === 'register') && !
session.isLoggedIn2())

```

Lines not prepended with a - or + symbol do not require change. See [How to Apply .patch Files](#) for more information on how to apply these changes manually.

Deploy Files and Configure the Implementation

When all customizations are complete, you are ready to deploy your custom files to your Site Builder SSP Application in NetSuite. The following topics explain how to deploy custom files to the Custom implementation:

1. Deploy Your Custom Files
2. Refresh Your Backend Environment
3. Configure the Models.js Script Path



Important: The customizations introduced by this patch are extensive. Always test changes in a sandbox account or other test site before deploying to a production account.

Deploy Your Custom Files

The first task is to upload your changes to the correct location in the Custom implementation of your Site Builder SSP Application.

Site Builder SSP Applications each have a locked **Reference** implementation that contains all source files and an unlocked **Custom** implementation, where you place your customizations.

The Custom implementation directory (in the File Cabinet) is organized to match the structure of the Reference implementation. Upload your customized files to the Custom implementation in the location mirroring the original source location.

Upload your customizations as a .zip file using the Advanced Add feature of the File Cabinet. Assuming that the location of each customized file in the Custom implementation mirrors the location in Reference, your customizations overwrite the source when your site compiles for runtime.

For example, if you upload a custom version of index-local.ssp to a My Account Premium 1.05.0 implementation, the correct location for deployment is the root:

Web Site Hosting Files > Live Hosting Files > SSP Applications > NetSuite Inc. - My Account Premium 1.05 > Custom My Account Premium

To deploy customized files:

1. In your local developer environment, open your root implementation folder.
2. Select the entire contents of the folder and include all files within a local .zip file.



Important: Include the root folder contents only. Do not include the root folder itself.

3. In NetSuite, navigate to the SSP Application folder for your implementation.

For example:

Web Site Hosting Files > Live Hosting Files > SSP Applications > NetSuite Inc. - My Account Premium 1.05 > Custom My Account Premium

4. Add your zip file to the File Cabinet using the Advanced Add feature. See the help topic [Uploading Files to the File Cabinet](#) for details.

Refresh Your Backend Environment

The next task is to update two configuration files in the NetSuite backend. These two files are:

- Application.js
- Templates.js

These files must be automatically regenerated. You can force this action by editing two configuration files in your File Cabinet.

To reset your environment:

Perform this procedure for both combiner.config and templates.config.

1. In your NetSuite File Cabinet, navigate to Web Site Hosting Files > Live Hosting Files > SSP Applications > [APPLICATION] > [CUSTOM IMPLEMENTATION FOLDER] > js.

2. Click **Edit** next to combiner.config.
3. Locate the **Media Item** section and click **Edit** to preview the file.
4. Place your cursor at the very end of the code in this file.
5. Add a single space to the end of the code and click **Save**.
This change forces a background process to generate a new, concatenated Application.js file.
6. Confirm that a new Application.js file is generated.
The date in the **Last Modified** column should reflect an updated date and time when the process is complete.
7. Repeat this procedure for the templates.config file, located here:
Web Site Hosting Files > Live Hosting Files > SSP Applications > NetSuite Inc. - My Account Premium 1.05 > Custom My Account Premium > templates.
This action updates the Templates.js file.

Configure the Models.js Script Path

The last task to complete is updating the Models.js library script path and touch points. This file is generated when the Models.js script runs. Before this happens, you must update the script path manually to point to the Custom folder.

To configure the Models.js script path:

1. In NetSuite, go to Setup > SuiteCommerce Advanced > SSP Applications.
2. Click **Edit** to modify the SSP Application associated with your implementation.
3. On the **Library** subtab under **Scripts**, locate the following library script:
Web Site Hosting Files/Live Hosting Files/SSP Applications/[APPLICATION]/ssp_libraries/Models.js
4. Manually change this path to point to the Custom Files location:
Web Site Hosting Files/**Custom Files**/SSP Applications/[APPLICATION]/ssp_libraries/Models.js
In the above examples, [APPLICATION] equals the SSP Application of your current implementation.
5. Click **Save**.
6. Update touch points. See the help topic [Select Supported Touch Points](#)

Upon successful deployment, you should be able to take advantage of secure shopping domains on your Site Builder site.

Cannot Scroll Through Long Menu Lists Using iOS

ⓘ Applies to: SuiteCommerce Advanced | Denali | Mont Blanc | Vinson | Elbrus | Kilimanjaro

In some cases, iOS users might encounter an issue where they cannot scroll through a long list of menu items if the list reaches beyond the device screen. If you experience this issue on your SuiteCommerce Advanced site, perform the updates described in the section corresponding to your implementation:

- [Elbrus Release and Earlier – Menu List Scrolling Patch \(iOS\)](#)
- [Kilimanjaro – Menu List Scrolling Patch \(iOS\)](#)

In addition to making these changes, you must create an ns.package.json file and update your distro.json file for any custom modules you include.

Elbrus Release and Earlier – Menu List Scrolling Patch (iOS)

ⓘ Applies to: SuiteCommerce Advanced | Denali | Mont Blanc | Vinson | Elbrus

This section explains how to override the _mixins.scss to correct the iOS menu item scrolling issue for any Elbrus implementations of SuiteCommerce Advanced and earlier. Due to the structure of dependencies within these implementations, you cannot divide the entry point within the distro.json file. Therefore, you must override this file. You can download the code samples described in this procedure here: [MixinsSassOverride-ElbrusAndEarlierSample.zip](#).



Important: Be aware of any existing extensions or customizations to the existing file and familiarize yourself with the [Best Practices for Customizing SuiteCommerce Advanced](#) before proceeding with any customizations.

Step 1: Override the _mixins.scss File

1. Create a directory to store your custom module.

Following best practices, name this directory **extensions** and place it in your Modules directory. Depending on your implementation and customizations, this directory might already exist.

2. Open your extensions directory and create a custom module to maintain your customizations.

Give this directory a unique name that is similar to the module being customized. For example:

Modules/extensions/MixinsSassOverride@1.0.0/

3. In your new module, create a subdirectory named Sass.

4. Copy the _mixins.scss file located in Modules > suitecommerce > BaseSassStyles@x.y.z and paste into your new Sass directory (where x.y.z represents the version of the module in your implementation of SuiteCommerce Advanced).

For example, paste your copy of this into the following location:

Modules/extensions/MixinsSassOverride@1.0.0/Sass/_mixins.scss

5. Open this file and remove the following line:

```
-webkit-overflow-scrolling: touch;
```

When you are finished, your file should match the following code snippet:

```
/*
  © 2017 NetSuite Inc.
  User may not copy, modify, distribute, or re-bundle or otherwise make available this code;
  provided, however, if you are an authorized user with a NetSuite account or log-in, you
  may use this code subject to the terms that govern your access and use.
*/

@mixin border-radius($radius) {
  -webkit-border-radius: $radius;
  -moz-border-radius: $radius;
  -ms-border-radius: $radius;
  border-radius: $radius;
}

%scroll-y{
  overflow-y: auto;
}
```

```

// avoid using BS mixin cause wont generate "-webkit-transform"
@mixin transition-transform($duration...) {

    -webkit-transition: -webkit-transform $duration;
    -moz-transition: -moz-transform $duration;
    -o-transition: -o-transform $duration;
    transition: transform $duration;
}

@mixin appearance($appearance) {
    -webkit-appearance: $appearance;
    -moz-appearance: $appearance;
    appearance: $appearance;
}

//used for select arrow down
@mixin angle-down-background($color){
    $color-local: remove-hash-from-color($color);
    background-image:url("data:image/svg+xml;utf8,<svg width='2000px' height='2000px' fill='%23{$color-local}' xmlns='http://www.w3.org/2000/svg'><path d='M1075 352q0 -13 -10 -23l-50 -50q-10 -10 -23 -10t-23 10l-393 393l-393 -393q-10 -10 -23 -10t-23 10l-50 50q-10 10 -10 23t10 23l466 466q10 10 23 10t23 -10l466 -466q10 -10 10 -23z'/'></svg>");
}

@function remove-hash-from-color($color) {
    @return str-slice(ie-hex-str($color), 4);
}

//All placeholders equal
@mixin placeholder {
    ::-webkit-input-placeholder {@content}
    :-moz-placeholder           {@content}
    ::-moz-placeholder          {@content}
    :-ms-input-placeholder     {@content}
}

```

- Save the file.

Step 2: Prepare the Developer Tools for Your Override

- Open the MixinsSassOverride@1.0.0 module.
 - Create a file in this module and name it **ns.package.json**.
- Modules/extensions/MixinsSassOverride@1.0.0/ns.package.json**
- Build the ns.package.json file using the following code

```
{
    "gulp": {
        "sass": [
            "Sass/**/*.scss"
        ]
    },
    "overrides": {
        "suitecommerce/BaseSassStyles@2.0.0/Sass/_mixins.scss" : "Sass/_mixins.scss"
    }
}
```

```

    }
}
```

Note: Replace the string `x.y.z` in the above example with the version of the module in your version of SuiteCommerce Advanced.

4. Save the `ns.package.json` file.
5. Open the `distro.json` file.
This file is located in the top-level directory of your SuiteCommerce Advanced source code.
6. Add your module to the `modules` object to ensure that the Gulp tasks include it when you deploy.
Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/MixinsSassOverride": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    // ...
  }
}
```

7. Save the file.

Step 3: Deploy Your Override

1. Deploy your source code customizations to your NetSuite account and test the functionality using a phone running iOS. See [Deploy to NetSuite](#) for details.
2. Confirm your results.

Upon successful deployment, iOS mobile users should be able to scroll through long menu lists.

Kilimanjaro – Menu List Scrolling Patch (iOS)

Applies to: SuiteCommerce Advanced | Kilimanjaro

In your Kilimanjaro implementation of SuiteCommerce Advanced, you can extend the `_mixins.scss` file to overwrite the `%scroll-y` selector as described in this section. You can download the code samples described in this procedure here: [MixinsSassExtension-KilimanjaroSample.zip](#)

Step 1: Extend the `_mixins.scss` File

1. If you have not done so already, create a directory to store your custom module.
Following best practices, name this directory **extensions** and place it in your `Modules` directory. Depending on your implementation and customizations, this directory might already exist.

2. Open your extensions directory and create a custom module to maintain your customizations.

Give this directory a unique name that is similar to the module being customized. For example:

Modules/extensions/MixinsSassExtension@1.0.0/

3. In your new module, create a subdirectory named Sass.

4. In your Sass subdirectory, create a new .scss file.

Give this file a unique name that is similar to the file being modified. For example:

Modules/extensions/mixinsSassExtension@1.0.0/Sass/_mixinsExtension.scss

5. Open this file and overwrite the **%scroll-y** selector.

Your file should match the following code snippet:

```
%scroll-y{
    overflow-y: auto;
}
```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Customizations

1. Open the MixinsSassExtension@1.0.0 module.

2. Create a file in this module and name it **ns.package.json**.

Modules/extensions/MixinSassExtension@1.0.0/ns.package.json

3. Build the ns.package.json file using the following code

```
{
  "gulp": {
    "sass": [
      "Sass/**/*.scss"
    ]
  }
}
```

4. Save the ns.package.json file.

5. Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

6. Add your custom module to the **modules** object to ensure that the Gulp tasks include it when you deploy.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Kilimanjaro",
  "version": "2.0",
  "isSCA": true,
  "buildToolsVersion": "1.3.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  }
}
```

```

},
"modules": {
    "extensions/MixinsSassExtension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    //...
}

```

- Split the **BaseSassStyle** entry in the **sass** object of each application (**Shopping**, **MyAccount**, and **Checkout**) and add your custom module (**MixinsSassExtension**) between them. This ensures that the Sass variables are loaded in the correct order.



Note: You do not need to wrap your custom module in an **include** array. The compiler adds all files within your new module by default.

```

//...
"sass": {
//...
"applications": [
{
    "name": "Shopping",
    "exportFile": "shopping.css",
    "dependencies": [
//...
{
        "module": "BaseSassStyles",
        "include": [
            "main-variables",
            "main-base"
        ]
},
        "MixinsSassExtension",

{
        "module": "BaseSassStyles",
        "include": [
            "main-atoms",
            "main-molecules",
            "molecules/datepicker"
        ]
},
        //...
    ]
},
{
    "name": "MyAccount",
    "exportFile": "myaccount.css",
    "dependencies": [
//...
{
        "module": "BaseSassStyles",
        "include": [
            "main-variables",
            "main-base"
        ]
},
        //...
    ]
},
{
    "name": "Checkout"
}
]
}

```

```

    "MixinsSassExtension",

    {
        "module": "BaseSassStyles",
        "include": [
            "main-atoms",
            "main-molecules",
            "molecules/datepicker"
        ]
    },
    // ...
    ],
},
{
    "name": "Checkout",
    "exportFile": "checkout.css",
    "dependencies": [
    // ...
    {
        "module": "BaseSassStyles",
        "include": [
            "main-variables",
            "main-base"
        ]
    },
    "MixinsSassExtension",

    {
        "module": "BaseSassStyles",
        "include": [
            "main-atoms",
            "main-molecules",
            "molecules/datepicker"
        ]
    },
    // ...
    ],
},
//...
]
}

```

8. Save the distro.json file.

Step 3: Deploy Your Extension

1. Deploy your source code customizations to your NetSuite account and test the functionality using a phone running iOS. See [Deploy to NetSuite](#) for details.
2. Confirm your results.

Upon successful deployment, iOS mobile users should be able to scroll through long menu lists.

Pages Not Indexed Using Google's Mobile-First Indexing

ⓘ Applies to: SuiteCommerce Advanced | 2018.2 | Aconcagua | Kilimanjaro | Elbrus | Vinson | Mont Blanc | Denali

In some cases, search engine crawlers discard content from the page generator and do not load all the required resources to render pages correctly. This results in SuiteCommerce pages not rendering content. Applying this patch corrects this issue by maintaining the SEO pre-render result.

This patch uses the override method to replace shopping.ssp, located in the **ShoppingApplication** module. You can download the code samples described in this procedure here: [MobileFirstIndexingPatch.zip](#).

ⓘ Note: In general, NetSuite best practice is to extend JavaScript using the JavaScript **prototype** object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify an SSP file, which requires that you use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1: Create the shopping.ssp Override File

1. Open the Modules/extensions directory and create a subdirectory to maintain your customizations and store your custom module.

Give this subdirectory a name similar to the module being customized. For example: **Modules/extensions/ShoppingApplication.Extension@1.0.0**.

ⓘ Note: If you are implementing the Elbrus release of SCA or later, the **extensions** directory already exists in your source code. If implementing Vinson release or earlier, you must add it manually.

2. In your new ShoppingApplication.Extension@1.0.0 directory, create a subdirectory called **SuiteScript**.

For example: **Modules/extensions/ShoppingApplication.Extension@1.0.0/SuiteScript**

3. Find the following file in your existing source:

Modules/suitecommerce/ShoppingApplication@X.Y.Z/shopping.ssp

In this example, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

4. Copy this entire file and paste a copy in your new module's SuiteScript directory.

For example: **Modules/extensions/ShoppingApplication.Extension@1.0.0/SuiteScript/shopping.ssp**

5. Open your new copy of shopping.ssp and locate the following line:

```
document.getElementById("main").innerHTML = '';
```

6. Replace this line with the following code:

```
if (!navigator.userAgent.match(/googlebot/i))
```

```
{
  document.getElementById('main').innerHTML="";
}
```

- Save the file.

Step 2: Prepare the Developer Tools For Your Customization

- Open your ShoppingApplication.Extension@1.0.0 module directory.
- Create a file in this directory and name it ns.package.json.

For example: `Modules/extensions/ShoppingApplication.Extension@1.0.0/ns.package.json`

- Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "ssp-files": {
      "SuiteScript/shopping.ssp": "shopping.ssp"
    }
  },
  "overrides": {
    "suitecommerce/ShoppingApplication@X.Y.Z/SuiteScript/shopping.ssp" : "SuiteScript/shopping.ssp"
  }
}
```

In your code, replace X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

- Open the distro.json file. This is located in your root directory.
- Add your custom module to the `modules` object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced 2018.2.0 (sc-2018.2.0)",
  "version": "2.0",
  "isSCA": true,
  "buildToolsVersion": "sc-2018.2.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/ShoppingApplication.Extension": "1.0.0",
    "suitecommerce/Account": "sc-2018.2.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

- Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations by deploying them to your NetSuite account (see [Deploy to NetSuite](#)).



Note: Since this patch modifies a SuiteScript file, changes are not visible in your local environment. SuiteScript files run on the server and must be deployed to NetSuite to take effect.

2. Confirm your results.

Disabling Display of SuiteCommerce Gift Wrap & Message Extension Transaction Line Fields

Applies to: SuiteCommerce Advanced

After installing the SuiteCommerce Gift Wrap & Message extension, the application adds the following transaction line item fields to all items in your account:

- Gift Wrap Item
- Gift Wrap Item ID
- Gift Wrap Message From
- Gift Wrap Message To
- Gift Wrap Message Text

After installing the extension, these added fields may display on your site, even if you do not activate the extension. Follow these instructions, depending on your implementation, to prevent these fields from displaying on your site:

- SuiteCommerce and SuiteCommerce Advanced Aconcagua and Later
- SuiteCommerce Advanced Denali through Kilimanjaro
- SuiteCommerce Advanced Pre-Denali
- Site Builder Websites

SuiteCommerce and SuiteCommerce Advanced Aconcagua and Later

Follow the steps below to disable the display of SuiteCommerce Gift Wrap & Message extension transaction line item fields on product display pages.

Step 1: Activate the SuiteCommerce Gift Wrap & Message Extension.

1. If you have not done so already, install the SuiteCommerce Gift Wrap & Message extension into your NetSuite account. See the help topic [Install Theme and Extension SuiteApps](#).
2. Activate the SuiteCommerce Gift Wrap & Message extension. See see the help topic [Manage Themes and Extensions](#).

Step 2: Update the Configuration Record for the Web Site and Domain

1. Go to Setup > SuiteCommerce Advanced > Configuration.

2. Select the web site and domain.
3. Click **Configure**.
4. Navigate to the **My Extensions** tab.
5. On the **Gift Wrap & Message** subtab, clear the **Enable Gift Wrap** box.
6. Click **Save**.

Step 3: Clear the Domain Cache and Confirm Results

1. Go to Setup > SuiteCommerce Advanced > Cache Invalidation Requests.
2. Click **New Invalidation Request**.
3. Double-click on the domain for which you want to clear the cache.
4. Enable the **Clear cache for the whole domain(s)** option.
5. Click **Submit**.
6. Click **OK**.
7. Confirm your results.

After a successful configuration, your site will not display the gift wrap transaction line item fields.

SuiteCommerce Advanced Denali through Kilimanjaro

This patch disables the display of the SuiteCommerce Gift Wrap & Message extension transaction line item fields on product display pages.

You can download the code samples described in this procedure here:
[GiftWrap.itemOptionsExtension@1.0.0—KilimanjaroSample.zip](#)

Step 1: Create the Directory Structure to Store Your Custom Module

1. Open the Modules/extensions directory and create a subdirectory to maintain your customizations.

Note: If you are implementing the Vinson release of SuiteCommerce Advanced or earlier, you might need to create the extensions directory manually.

2. In the extensions directory, create a subdirectory called GiftWrap.ItemOptionsExtension@1.0.0.

When creating a new custom module, the module name must be unique. You must not have duplicate module names, even if those modules reside in different folders.

3. In your new module, create a subdirectory called Sass.
4. In your new Sass subdirectory, create a text file.

Give this file a unique name that is similar to the module directory. For example:

`_giftWrap-item-options-extension.scss`

Step 2: Create a Sass File

1. In the Sass directory, create a new file called `_giftWrap-item-options-extension.scss`.
2. Add the following text to the `_giftWrap-item-options-extension.scss` file:

```
[id*='custcol_ns_sc_ext_gw_item'],
[name*='custcol_ns_sc_ext_gw_item'],
[id*='custcol_ns_sc_ext_gw_id'],
```

```
[name*='custcol_ns_sc_ext_gw_id'],
[id*='custcol_ns_sc_ext_gw_message_from'],
[name*='custcol_ns_sc_ext_gw_message_from'],
[id*='custcol_ns_sc_ext_gw_message_to'],
[name*='custcol_ns_sc_ext_gw_message_to'],
[id*='custcol_ns_sc_ext_gw_message_text'],
[name*='custcol_ns_sc_ext_gw_message_text'] {
    display: none;
}
```

Step 3: Create the ns.package.json File

1. Open the GiftWrap.ItemOptionsExtension@1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.
Modules/extensions/GiftWrap.ItemOptionsExtension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "sass": [
      "Sass/**/*.scss"
    ]
  }
}
```

4. Save the ns.package.json file.

Step 4: Update the distro.json File and Deploy Your Customizations to Your NetSuite Account

1. Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce source code.

2. Add your custom module to the modules object to ensure that the Gulp tasks include it when you deploy.

In this example, the `extensions/GiftWrap.ItemOptionsExtension@1.0.0`, module is added at the beginning of the list of the other extensions modules. However, you can add the module anywhere in the modules object. The order of precedence in this list does not matter.

```
{
  "name": "SuiteCommerce Advanced Kilimanjaro",
  "version": "2.0",
  "isSCA": true,
  "buildToolsVersion": "1.3.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/GiftWrap.ItemOptionsExtension": "1.0.0",
  }
}
```

```

    "extensions/QuoteToSalesOrderExtension": "1.0.0",
    "extensions/TransactionExtention": "1.0.0",
    "extensions/OrderWizardPaymentMethodExtension": "1.0.0",
    ...

```

3. Include the module definition (`GiftWrap.ItemOptionsExtension`) in the dependencies/applications array of the `sass` object within applications/dependencies/ for each application (Shopping, MyAccount, and Checkout). You must include this file after the module you are overriding. Your distro.json file should look similar to the following:

```

//...
"tasksConfig": {
//...
"sass": {
//...
"applications": [
{
    "name": "Shopping",
    "exportFile": "shopping.css",
    "dependencies": [
//...
        "Location.SCA",
        "GiftWrap.ItemOptionsExtension"
    ]
}
//...
{
    "name": "MyAccount",
    "exportFile": "myaccount.css",
    "dependencies": [
//...
        "Location.SCA",
        "GiftWrap.ItemOptionsExtension"
    ]
}
//...
{
    "name": "Checkout",
    "exportFile": "checkout.css",
    "dependencies": [
//...
        "Location.SCA",
        "GiftWrap.ItemOptionsExtension"
    ]
}
]
}
//...

```

4. Save the distro.json file.
5. Deploy your source code customizations to your NetSuite account. See [Deploy to NetSuite](#) for details.

Step 5: Clear Your Domain Cache and Confirm Results

1. Go to Setup > SuiteCommerce Advanced > Cache Invalidation Requests.

2. Click **New Invalidation Request**.
3. Double-click on the domain for which you want to clear the cache.
4. Enable the **Clear cache for the whole domain(s)** option.
5. Click **Submit**.
6. Click **OK**.
7. Confirm your results.

After a successful deployment, your site will not display the gift wrap transaction line item fields.

SuiteCommerce Advanced Pre-Denali

Follow the steps below to disable the display of SuiteCommerce Gift Wrap & Message extension transaction line item fields on product display pages.

Step 1: Add code to your application stylesheet

1. Open any of the SuiteCommerce Advanced instance stylesheets located in your local development files. For example:
Custom ShopFlow 1.07.0/skins/standard/styles.css
2. Paste the following code at the bottom of the file:

```
[id*='custcol_ns_sc_ext_gw_item'],
[name*='custcol_ns_sc_ext_gw_item'],
[id*='custcol_ns_sc_ext_gw_id'],
[name*='custcol_ns_sc_ext_gw_id'],
[id*='custcol_ns_sc_ext_gw_message_from'],
[name*='custcol_ns_sc_ext_gw_message_from'],
[id*='custcol_ns_sc_ext_gw_message_to'],
[name*='custcol_ns_sc_ext_gw_message_to'],
[id*='custcol_ns_sc_ext_gw_message_text'],
[name*='custcol_ns_sc_ext_gw_message_text'] {
    display: none;
}
```

Step 2: Upload the stylesheet to the File Cabinet and Compile the Change

1. Upload the file to the File Cabinet in the appropriate location. Example:
CustomShopFlow 1.07.0/skins/standard/styles.css
2. Trigger the style's combiner.config file to compile the change by editing and saving the styles.css file without making any changes.

Step 3: Clear Your Domain Cache and Confirm Results

1. Go to Setup > SuiteCommerce Advanced > Cache Invalidation Requests.
2. Click **New Invalidation Request**.
3. Double-click on the domain for which you want to clear the cache.
4. Enable the **Clear cache for the whole domain(s)** option.
5. Click **Submit**.

6. Click **OK**.
7. Confirm your results.

After a successful deployment, your site will not display the gift wrap transaction line item fields.

Site Builder Websites

For Site Builder web sites, you only need to add some code to your Web Site Setup record and save the changes. After a successful configuration, your site no longer displays the gift wrap transaction line item fields.

Adding Code to Your Web Site Setup Record

1. In NetSuite, go to Setup > SiteBuilder > Web Site Set Up.
2. Click **Edit** for the website you want to patch.
3. Go to the **Analytics** subtab and paste the following code into the **Additional to <head>** field:

```
<style type="text/css">
    [id*='custcol_ns_sc_ext_gw_item'],
    [name*='custcol_ns_sc_ext_gw_item'],
    [id*='custcol_ns_sc_ext_gw_id'],
    [name*='custcol_ns_sc_ext_gw_id'],
    [id*='custcol_ns_sc_ext_gw_message_from'],
    [name*='custcol_ns_sc_ext_gw_message_from'],
    [id*='custcol_ns_sc_ext_gw_message_to'],
    [name*='custcol_ns_sc_ext_gw_message_to'],
    [id*='custcol_ns_sc_ext_gw_message_text'],
    [name*='custcol_ns_sc_ext_gw_message_text'] {
        display: none;
    }
</style>
```

4. Click **Save**.

Users Not Redirected to External Payment System

 **Applies to:** SuiteCommerce Advanced | Vinson

In Vinson releases of SCA, if a user chooses an external payment method during checkout, SuiteCommerce Advanced does not correctly open the third-party payment system.

When a user clicks the **Continue to External Payment** button during checkout, instead of sending the user to the external payment system, SCA redirects the user to the beginning of the checkout process.

This error occurs because SuiteCommerce Advanced deprecated the existing payment processing status result, **HOLD**, in 17.1. Instead, SCA uses a new status result, **PENDING**. As a result, any SCA sites implementing the Vinson release requires this patch to use third-party payment systems.

To implement this patch, you must extend the **prototype** object for the **ExternalPaymentModel** function that includes the **_isDone()**, **_validateRecordStatus()**, and **_validateStatusFromRequest()** methods. This function is located in **ExternalPayment.Model.js**.

for the **ExternalPayment** module. You can download the code samples described in this procedure here: [UsersNotRedirectedExternalPaymentSystem.zip](#).

Step 1: Extend the ExternalPayment.Model.js File

1. If you have not done so already, create a directory to store your custom module. Give this directory a name similar to the module being customized. For example, create **Modules/extensions/ExternalPaymentExtension@1.0.0**
2. In your new ExternalPaymentExtension@1.0.0 directory, create a subdirectory called **JavaScript**. For example: **Modules/extensions/ExternalPaymentExtension@1.0.0/SuiteScript**
3. In your new JavaScript subdirectory, create a JavaScript file to extend ExternalPayment.Model.js. Name this file according to best practices. For example:
ExternalPayment.Model.Extension.js
4. Open this file and extend the **ExternalPaymentModel()** function as shown in the following code snippet.

```
define('ExternalPayment.Model.Extension'
  [
    'ExternalPayment.Model',
    'underscore'
  ]
  , function (
    ExternalPaymentModel
    '-
  )
{
  'use strict';

  _.extend(ExternalPaymentModel.prototype,
  {
    _isDone: function (request, record_type)
    {
      var status = this._getStatusFromRequest(request, record_type)
      , status_accept_value = this._getConfiguration(record_type, 'statusAcceptValue', 'ACCEPT')
      , status_hold_value = this._getConfiguration(record_type, 'statusHoldValue', 'HOLD')

      //Added by payment
      , status_pending_value = this._getConfiguration(record_type, 'statusPendingValue', 'PENDING')
      return status === status_accept_value || status === status_hold_value || status ===
status_pending_value;
    }

    , _validateRecordStatus: function (record)
    {
      //Modified by Payment
      return record.getFieldValue('paymenteventholdreason') === 'FORWARD_REQUESTED';
    }

    , _validateStatusFromRequest: function (request, record_type)
    {
      var status = this._getStatusFromRequest(request, record_type)
      , status_accept_value = this._getConfiguration(record_type, 'statusAcceptValue' , 'ACCEPT')
      , status_hold_value = this._getConfiguration(record_type, 'statusHoldValue' , 'HOLD')
```

```

    //Added by Payment
    , status_pending_value = this._getConfiguration(record_type, 'statusPendingValue', 'PENDING')
    , status_reject_value = this._getConfiguration(record_type, 'statusRejectValue' , 'REJECT');

    //Modified by Payment
    return status === status_accept_value || status === status_pending_value || status ===
    status_hold_value || status === status_reject_value;
}
});
}
);

```

- Save the file.

Step 2: Prepare the Developer Tools For Your Customization

- Open your new ExternalPaymentExtension@1.0.0 module directory.

- Create a file in this directory and name it ns.package.json.

Modules/extensions/ExternalPaymentExtension@1.0.0/ns.package.json

- Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "ssp-libraries": [
      "SuiteScript/*.js"
    ]
  }
}
```

- Open the distro.json file. This is located in your root directory.

- Add your custom modules to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Vinson Release",
  "version": "2.0",
  "buildToolsVersion": "1.2.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/ExternalPaymentExtension": "1.0.0",
    "suitecommerce/Account": "2.2.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

- Add **ExternalPaymentExtension** as a dependency to SCA entry point within the **SC.Shopping.Starter** entrypoint of the JavaScript array.

Your distro.js file should look similar to the following:

```
"tasksConfig": {
// ...
"javascript": [
// ...
{
"entryPoint": "SC.Checkout.Starter",
"exportFile": "checkout.js",
"dependencies": [
"ExternalPaymentExtension",
"Backbone.View.Plugins",
"jQuery.html",
// ...
],
// ...
]
}
}
```

7. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Deploy your source code customizations to your NetSuite account and test the functionality. See [Deploy to NetSuite](#) for details.



Note: Since this patch modifies a SuiteScript file, changes are not visible in your local environment. SuiteScript files run on the server and must be deployed to NetSuite to take effect.

2. Confirm your results.

Upon successful deployment, the checkout process redirects to an external payment system when you click **Continue to External Payment**.

Incorrect Redirect URL for External Payments

Applies to: SuiteCommerce Advanced | Elbrus | Vinson

On releases of Elbrus and Vinson, the checkout process that uses an external payment system does not correctly cancel the process and return to the SuiteCommerce site. Instead, when you click **Cancel** while using an external payment system, a 404 error occurs. This error occurs due to an incorrect URL variable value in the **OrderWizard.Module.PaymentMethod** module.

To implement this patch, you must extend the **prototype** object for the **render()** method in **OrderWizard.Module.PaymentMethod.External.js** for the **OrderWizard.Module.PaymentMethod** module. You can download the code samples described in this procedure here: [IncorrectRedirectURLforExternalPaymentsElbrus.zip](#) or [IncorrectRedirectURLforExternalPaymentsVinson.zip](#).



Note: Before proceeding, familiarize yourself with [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend the **OrderWizard.Module.PaymentMethod.External.js** File

1. If you have not done so already, create a directory to store your custom module.

Give this directory a name similar to the module being customized. For example, create **Modules/extensions/OrderWizard.Module.PaymentMethod.Extension@1.0.0**.

2. In your new OrderWizard.Module.PaymentMethod.Extension@1.0.0 directory, create a subdirectory called **JavaScript**.

For example: **Modules/extensions/OrderWizard.Module.PaymentMethod.Extension@1.0.0/JavaScript**

3. In your new JavaScript subdirectory, create a JavaScript file to extend OrderWizard.Module.PaymentMethod.External.js.

Name this file according to best practices. For example:

OrderWizard.Module.PaymentMethod.External.Extension.js

4. Open this file and extend the **render()** method as shown in the following code snippet.

```
define(
    'OrderWizard.Module.PaymentMethod.External.Extension',
    [
        'OrderWizard.Module.PaymentMethod',
        'Session',
        'OrderPaymentmethod.Model'

        , 'order_wizard_paymentmethod_external_module_tpl'

        , 'Backbone'
        , 'underscore'
        , 'SC.Configuration'
        , 'Utils'
    ]
    ,
    function (
        OrderWizardModulePaymentMethod
        , Session
        , OrderPaymentmethodModel

        , order_wizard_paymentmethod_external_module_tpl

        , Backbone
        , Configuration
        , Utils
    )
{
    'use strict';

    _.extend(OrderWizard.Module.PaymentMethod.prototype, {
        render: function ()
        {
            var n = Configuration.get('siteSettings.id')
            , status_accept_value = Configuration.get('siteSettings.externalCheckout.' + this.options.record_type.toUpperCase() + '.statusAcceptValue' , 'ACCEPT')
            , status_reject_value = Configuration.get('siteSettings.externalCheckout.' + this.options.record_type.toUpperCase() + '.statusRejectValue' , 'REJECT')
            , status_parameter_name = Configuration.get('siteSettings.externalCheckout.' + this.options.record_type.toUpperCase() + '.statusParameterName' , 'status')
            , origin = window.location.origin ? window.location.origin : (window.location.protocol + '//' +
            window.location.hostname + (window.location.port ? (':' + window.location.port) : ''))

        }
    })
}
```

```

        , url = origin + _.getAbsoluteUrl('external_payment.ssp')
        , current_touchpoint = Configuration.get('currentTouchpoint')
        , thankyouurl_parameters = {n: n, externalPaymentDone: 'T', touchpoint: current_touchpoint,
recordType: this.options.record_type || 'salesorder' }
        , errorurl_parameters = {n: n, externalPaymentDone: 'T', touchpoint: current_touchpoint,
recordType: this.options.record_type || 'salesorder' }
        , returnurl_parameters = {n: n, externalPaymentDone: 'T', touchpoint: current_touchpoint,
recordType: this.options.record_type || 'salesorder'};

thankyouurl_parameters[status_parameter_name] = status_accept_value;
errorurl_parameters[status_parameter_name] = status_reject_value;

if (this.options.prevent_default)
{
    var prevent_default_parameter_name = Configuration.get('siteSettings.externalCheckout.' +
+ this.options.record_type.toUpperCase() + '.preventDefaultParameterName', 'preventDefault')
    , prevent_default_value = Configuration.get('siteSettings.externalCheckout.' + this.options.
record_type.toUpperCase() + '.preventDefaultValue', 'T');

    thankyouurl_parameters[prevent_default_parameter_name] = prevent_default_value;
    errorurl_parameters[prevent_default_parameter_name] = prevent_default_value;
    returnurl_parameters[prevent_default_parameter_name] = prevent_default_value;
}

this.paymentMethod = new OrderPaymentmethodModel({
    type: 'external_checkout_' + this.options.paymentmethod.key
    , isexternal: 'T'
    , internalid: this.options.paymentmethod.internalid
    , merchantid: this.options.paymentmethod.merchantid
    , key: this.options.paymentmethod.key
    , thankyouurl: Utils.addParamsToUrl(url, thankyouurl_parameters) // Commerce API
    , errorurl: Utils.addParamsToUrl(url, errorurl_parameters) // Commerce API
    , returnurl: Utils.addParamsToUrl(url, returnurl_parameters) // SuiteScript
});

this._render();

}
}

});

```

5. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open your new OrderWizard.Module.PaymentMethod.Extension@1.0.0 module directory.
2. Create a file in this directory and name it ns.package.json.
Modules/extensions/OrderWizard.Module.PaymentMethod.Extension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Open the distro.json file. This is located in your root directory.
5. Add your custom modules to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Vinson Release",
  "version": "2.0",
  "buildToolsVersion": "1.2.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/OrderWizard.Module.PaymentMethod.Extension": "1.0.0",
    "suitecommerce/Account": "2.2.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

6. Add **OrderWizard.Module.PaymentMethod.Extension** as a dependency to SCA entry point within the **SC.Checkout.Starter** entrypoint of the JavaScript array.

Your distro.js file should look similar to the following:

```
"tasksConfig": {
//...
"javascript": [
//...
{
  "entryPoint": "SC.Checkout.Starter",
  "exportFile": "checkout.js",
  "dependencies": [
    "Backbone.View.Plugins",
    "OrderWizard.Module.PaymentMethod.Extension",
    // ...
  ],
//...
}
]
```

7. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, the **Cancel** button for an external payment system redirects back to the SuiteCommerce site from where it was launched.

Invoices Do Not Include the Request for a Return Button

ⓘ Applies to: SuiteCommerce Advanced | Vinson

In Reference My Account v1.06, invoices do not include the **Request for a Return** button. As a result, a customer cannot request to return items on an invoice in NetSuite. However, this button does appear on invoices for Reference My Account Premium v1.06.

To add this button, this patch overrides `invoice_details.tpl`, `Invoice.Router.js`, and `Invoice.Details.View.js` in the **Invoice** module. You can download the code samples described in this procedure here: [RequestForAReturnButtonOnInvoices.zip](#)

ⓘ Note: In general, NetSuite best practice is to extend JavaScript using the JavaScript **prototype** object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify files that you cannot extend, and therefore requires you to use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1. Override the JavaScript and Template Files

1. If you have not done so already, create a directory to store your custom module. Give this directory a name similar to the module being customized. For example, create **Modules/extensions/Invoice.Extension@1.0.0**
2. In your new `Invoice.Extension@1.0.0` directory, create subdirectories called **JavaScript** and **Templates**. For example: `Modules/extensions/Invoice.Extension@1.0.0/JavaScript` and `Modules/extensions/Invoice.Extension@1.0.0/Templates`.
3. Copy the following files:

```
Modules/suitecommerce/Invoice@X.Y.Z/JavaScript/Invoice.Router.js
Modules/suitecommerce/Invoice@X.Y.Z/JavaScript/Invoice.Details.View.js
Modules/suitecommerce/Invoice@X.Y.Z/Templates/invoice_details.tpl
```

In this example, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

4. Paste a copy in your new module's `JavaScript` and `Templates` directories. For example: `Modules/extensions/Invoice.Extension@1.0.0/JavaScript/Invoice.Router.js` or `Modules/extensions/Invoice.Extension@1.0.0/Templates/invoice_details.tpl`
5. Open your new copy of `Invoice.Router.js` and locate the `showInvoice()` method.

Replace the existing `showInvoice()` method with the following code:

```
showInvoice: function (invoice_id, referer)
{
    var invoice = new InvoiceModel({internalid: invoice_id})
        , view = new DetailsView({
            application: this.application
            , model: invoice
            , referer: referer
        });
    invoice
        .fetch({
            killerId: AjaxRequestsKiller.getKillerId()
        })
        .done(function() {
            view.isReturnable()
                .done(function() {
                    view.showContent();
                })
        });
}
```

6. Save the file.
7. Open your new copy of `Invoice.Details.View.js` and make the following changes.

- a. Find the following line:

```
, 'underscore'
```

Replace this line with the following code:

```
, 'underscore'
, 'ReturnAuthorization.GetReturnableLines'
, 'OrderHistory.Model'
, 'AjaxRequestsKiller'
, 'jQuery'
```

- b. Find the following line:

```
' -
```

Replace this line with the following code:

```
' -
, ReturnLinesCalculator
, OrderHistoryModel
, AjaxRequestsKiller
, JQuery
```

- c. Find the following lines:

```
, makeAPayment: function ()
{
    LivePaymentModel.getInstance().selectInvoice(this.model.id);
}
```

Replace these lines with the following code:

```

, makeAPayment: function ()
{
    LivePaymentModel.getInstance().selectInvoice(this.model.id);
}

, makeAPayment: function ()
{
    LivePaymentModel.getInstance().selectInvoice(this.model.id);
}

//@method isReturnable calculates a value based on the fulfillments and returns made,
// if the invoice accepts returns
, isReturnable: function ()
{
    var deferred = JQuery.Deferred();

    var self = this;
    var model = new OrderHistoryModel();
    model
        .fetch({
            data: {
                internalid: this.model.get('createdfrom').internalid
                , recordtype: this.model.get('createdfrom').recordtype
            }
            , killerId: AjaxRequestsKiller.getKillerId()
        })
        .done(function() {
            var ret_calculator = new ReturnLinesCalculator(model, {notConsiderFulfillments: true});
            self.validLines = ret_calculator.calculateLines().validLines.length;
            deferred.resolve();
        });
}

return deferred.promise();
}

```

- d. Find the following line:

```
, showOpenedAccordion: _.isTabletDevice() || _.isDesktopDevice()
```

Replace this line with the following code:

```

, showOpenedAccordion: _.isTabletDevice() || _.isDesktopDevice()
//@property {Boolean} showRequestReturnButton
, showRequestReturnButton: !!this.validLines

```

- e. Find the following line inside the `initialize()` function:

```
this.navSource = options.referer === 'paidinvoices' ? '/paid-invoices' : '/invoices';
```

Replace this line with the following code:

```

this.navSource = options.referer === 'paidinvoices' ? '/paid-invoices' : '/invoices';
this.model = options.model;

```

```
this.validLines = 0;
```

8. Save the file.
9. Open your new copy of invoice_details.tpl and make the following change.

Find the following lines:

```
<a target="_blank" class="invoice-details-button-download-as-pdf" href="{{downloadPdfUrl}}>
    {{translate 'Download as PDF'}}
</a>
```

Replace these lines with the following code:

```
<a target="_blank" class="invoice-details-button-download-as-pdf" href="{{downloadPdfUrl}}>
    {{translate 'Download as PDF'}}
</a>

{{#if showRequestReturnButton}}

<a href="/returns/new/invoice/{{model.internalid}}" class="invoice-details-button-request" data-permissions="transactions.tranRtnAuth.2">
    {{translate 'Request a Return'}}
</a>

{{/if}}
```

10. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open your Modules/extensions/ module directory.
2. Create a file in this directory and name it ns.package.json.
3. Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ],
    "overrides": {
      "suitecommerce/Invoice@X.Y.Z/Javascript/Invoice.Router.js" : "JavaScript/Invoice.Router.js",
      "suitecommerce/Invoice@X.Y.Z/Javascript/Invoice.Details.View.js" : "JavaScript/Invoice.Details.View.js",
      "suitecommerce/Invoice@X.Y.Z/Templates/invoice_details.tpl" : "Templates/invoice_details.tpl"
    }
  }
}
```



Important: You must replace the string X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

4. Open the distro.json file. This is located in your root directory.
5. Add your custom modules to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Vinson Release",
  "version": "2.0",
  "buildToolsVersion": "1.2.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/Invoice.Extension": "1.0.0",
    "suitecommerce/Account": "2.2.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

6. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.
Upon successful deployment, Invoice records under My Account contain a **Request a Return** button .

npm Error on Implementations

① Applies to: SuiteCommerce Advanced | Denali

In some implementations of Denali R2, the `npm install` command returns an error. As a result, the installation cannot continue. This error occurs due to a version mismatch with the request HTTP client. You must change the version of the `request` client to correct the error.

Step 1: Modify package.json

1. If you have not already done so, create a backup of package.json, located in the root of your project directory.
2. Open package.json and make the following changes.

Locate the line:

```
"request": "2.39.0",
```

Replace it with the following line:

```
"request": "2.81.0",
```

3. Save the file.

Step 2: Test Your Changes

1. Confirm your results by running the `npm install` command again.
The `npm install` command successfully installs the required Node.js packages.

Content Appears Incorrectly in a Merchandising Zone

ⓘ Applies to: SuiteCommerce Advanced | Elbrus | Kilimanjaro

In some implementations of Elbrus and Kilimanjaro, items may appear in a merchandising zone that do not meet the criteria for the zone. This error applies to merchandising zones that are added in Site Management Tools (SMT) and appear on either the product detail or cart pages.

The process to implement the fix in this patch is the same whether you update Elbrus or Kilimanjaro. To implement this patch, you create a new JavaScript file and override CMSadapter.js in the **CMSadapter** module. The new JavaScript file adds a new merchandising implementation in the module.

ⓘ Note: This fix only applies if you are using CMS Adapter Version 3 on Kilimanjaro or CMS Adapter Version 2 on Elbrus.

For an example of the changes needed for this patch, see one of the following links for the source files, depending on your implementation:

- [MerchandisingZoneContentAppearsIncorrectlyElbrus.zip](#)
- [MerchandisingZoneContentAppearsIncorrectlyKilimanjaro.zip](#)

ⓘ Note: In general, NetSuite best practice is to extend JavaScript using the JavaScript **prototype** object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify a file in a way that you cannot extend, and therefore requires you to use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1: Create the CMSadapterImpl.Merchandising.js File and the CMSadapter.js Override File

This step shows how to override the CMSadapter.js file and create the CMSadapterImpl.Merchandising.js file. Both files are located in the **CMSadapter** module.

1. Open the Modules/extensions directory and create a subdirectory to maintain your customizations.
For example, create **Modules/extensions/CMSadapter.Extension@1.0.0**
2. In your new CMSadapter.Extension@1.0.0 directory, create a subdirectory called **JavaScript**.
For example, create **Modules/extensions/CMSadapter.Extension@1.0.0/JavaScript**
3. Copy the following file:
Modules/suitecommerce/CMSadapter@X.Y.Z/JavaScript/CMSadapter.js
In this example, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.
4. Paste a copy in your new JavaScript directory in the new module.
For example: **Modules/extensions/CMSadapter.Extension@1.0.0/JavaScript/CMSadapter.js**

- Open your new copy of the CMSadapter.js file and locate the following lines:

```
, 'CMSadapterImpl.Enhanced'
```

Replace this line with the following code:

```
, 'CMSadapterImpl.Enhanced'  
, 'CMSadapterImpl.Merchandising'
```

- In the new CMSadapter.js, add this line at the end of the `initAdapter()` method:

```
this.adapterMerchandising = new CMSadapterImplMerchandising(application, CMS);
```

Your code should look similar to the following:

```
, initAdapter: function(application, landingRouter)  
{  
    if (typeof CMS === 'undefined')  
    {  
        console.log('Error: CMS global variable not found - CMS adapter not initialized');  
        return;  
    }  
  
    this.adapterCore = new CMSadapterImplCore(application, CMS);  
    this.adapterLanding = new CMSadapterImplLanding(application, CMS, landingRouter);  
    this.adapterCategories = new CMSadapterImplCategories(application, CMS);  
  
    this.adapterMerchandising = new CMSadapterImplMerchandising(application, CMS);  
}
```

- Save the file.
- In the new JavaScript folder, build out the new CMSadapterImpl.Merchandising.js file with the following code:

```
/*  
@module CMSadapter  
@class CMSadapterImpl.Merchandising  
*/  
define('CMSadapterImpl.Merchandising'  
, [  
    'LiveOrder.Model'  
, 'underscore'  
,  
    function ()  
    {  
        LiveOrderModel  
        '  
    }  
]  
{  
    'use strict';  
    function AdapterMerchandising(application, CMS)  
    {  
        this.CMS = CMS;  
        this.application = application;  
        this.listenForCMS();  
    }  
}
```

```

_.extend(AdapterMerchandising.prototype, {
  listenForCMS: function () {
    var self = this;
    self.CMS.on('cart:items:get', function (promise) {
      var linesId = [];
      , cartInstance = LiveOrderModel.getInstance();
      cartInstance.cartLoad.then(function() {
        var lines = cartInstance.get('lines');
        lines.each(function(line) {
          var item = line.get('item')
          , internalId = item && item.get('internalid');
          if (internalId) {
            linesId.push(internalId);
          }
        });
        promise.resolve(linesId);
      });
    });
    self.CMS.on('page:items:get', function (promise) {
      var linesId = []
      , PDP_FULL_VIEW = 'ProductDetails.Full.View'
      , PDP_QUICK_VIEW = 'ProductDetails.QuickView.View'
      , view = self.application.getLayout().getCurrentView()
      , view_identifier = view && view.attributes &&
view.attributes.id
      , view_prototype_id = view && view.prototype &&
view.prototype.attributes && view.prototype.attributes.id;
      if (view && (view_identifier === PDP_FULL_VIEW ||
view_identifier === PDP_QUICK_VIEW ||
view_prototype_id === PDP_FULL_VIEW || view_prototype_id
=== PDP_QUICK_VIEW)) {
        var internalId = view.model && view.model.get('item') &&
view.model.get('item').get('internalid');
        if (internalId) {
          linesId.push(internalId);
        }
      }
      promise.resolve(linesId);
    });
  });
  return AdapterMerchandising;
});

```

9. Save CMSadapterImpl.Merchandising.js.

Step 2: Prepare the Developer Tools For Your Customization

This step shows how to create the ns.package.json file for the custom module and shows how to modify the distro.json file to make sure that the Gulp tasks include your modules when you deploy.

1. Open your new Modules/extensions/CMSadapter.Extension@1.0.0 module.

2. Create a file in this directory and name it ns.package.json.

`Modules/extensions/CMSadapterImpl.Extension@1.0.0/ns.package.json`

3. Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  },
  "suitecommerce/CMSadapter@X.Y.Z/JavaScript/CMSadapter.js" : "JavaScript/CMSadapter.js"
}
```



Important: You must replace the string X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

4. Open the distro.json file. This is located in your root directory.

5. Add your custom modules to the `modules` object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    ...
    "extensions/CMSadapter.Extension": "1.0.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

6. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.

2. Confirm your results.

Upon successful deployment, the merchandising zones for products contain only items that meet the criteria for the zone.

Reference My Account Generates Error on Load

ⓘ Applies to: SuiteCommerce Advanced | pre-Denali

This section applies to the **Reference My Account v1.04** bundle.

In some pre-Denali releases of SuiteCommerce Advanced , viewing the home page of My Account results in a blank page and SuiteCommerce returns the following error:

Uncaught TypeError: Cannot read property 'is_logged_in' of undefined.

This error can occur because of the number of invoices returned for My Account is too large for the browser to load. This patch limits the number of returned invoices at 1000. To implement this patch, you must update the **Application.getModel** function in Models.js.

To modify Models.js:

1. If you have not done so already, copy the Models.js file from the reference folder to your custom folder in the NetSuite File Cabinet.

Models.js is located in the File Cabinet at My Account 1.04 > Reference My Account > ssp_libraries.

2. Update the **setPaymentMethod** function in the Models.js file from your custom folder:

Locate the following lines in Models.js:

```
var invoices_info = Application.getModel('Receipts').list({
    type: 'invoice'
    , internalid: _.pluck(result.invoices, 'internalid')
});
```

Replace these lines with the following code:

```
var invoices_id = _.pluck(result.invoices, 'internalid');
invoices_id.sort(function(a,b){ return parseInt(b) - parseInt(a) });

var invoices_info = Application.getModel('Receipts').list({
    type: 'invoice'
    , internalid: invoices_id.slice(0, 1000)
});
```

3. Save Models.js.

4. Configure your SSP Application to use the custom Models.js file.

1. Open the My Account SSP application record at Setup > SuiteCommerce Advanced > SSP Applications.
2. On the **Scripts** subtab, update the **Libraries** list to include the Models.js file from your custom folder.

This file must maintain the correct position at the bottom of the list of library files.

3. Click Save.
5. Verify that the customization was successful.

Log in to the server and go to My Account > Overview. The home page for My Account loads successfully.

Error Loading Shopping Page Due to Uncaught TypeError

ⓘ Applies to: SuiteCommerce Advanced | Elbrus

In some implementations of Elbrus, the shopping page returns a blank page and the following error is returned in the log for the page:

Uncaught TypeError: Cannot read property 'indexOf' of undefined

This error can occur because the URL includes the percent sign (%) and SuiteCommerce does not correctly handle the symbol. This patch overrides Utils.js in the **Utilities** module to add the **getDecodedURLParameter** function and correctly handle the symbol. You can download the code samples described in this procedure here: [ShoppingPageErrorUncaughtTypeError.zip](#).

ⓘ Note: In general, NetSuite best practice is to extend JavaScript using the JavaScript **prototype** object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify a file that you cannot extend, and therefore requires you to use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1: Create the Utils.js Override File

This step shows how to override the Utils.js file, located in the **Utilities** module.

1. Open the Modules/extensions directory and create a subdirectory to maintain your customizations. Give this directory a name similar to the module being customized. For example, create **Modules/extensions/UtilitiesExtension@1.0.0**
2. In your new UtilitiesExtension@1.0.0 directory, create a subdirectory called **JavaScript**. For example: **Modules/extensions/UtilitiesExtension@1.0.0/JavaScript**
3. Copy the following file:
Modules/suitecommerce/Utilities@X.Y.Z/JavaScript/Utils.js
In this example, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.
4. Paste a copy in the JavaScript directory for the new module.
For example: **Modules/extensions/Utilities@X.Y.Z/JavaScript/Utils.js**
5. Open your new copy of Utils.js and make the following change.
Add the following **getDecodedURLParameter** function before the existing **parseUrlOptions** function:

```

function getDecodedURLParameter (url_parameter)
{
    url_parameter = url_parameter || '';
    var position;

    for(var temporal = ''; (position = url_parameter.indexOf('%')) >= 0; url_parameter =
        url_parameter.substring(position + 3))
    {
        temporal += url_parameter.substring(0, position);
        var x = url_parameter.substring(position, position + 3);

        try
        {
            temporal += decodeURIComponent(x);
        }
        catch (e)
        {
            temporal += x;
        }
    }
    return temporal + url_parameter;
}

```

Your final result should look similar to the following:

```

...
    return baseUrl;
}

function getDecodedURLParameter (url_parameter)
{
// new function
}

function parseUrlOptions (options_string)
{
...

```

- Add the call to the `getDecodedURLParameter` function in `SC.Utils`:

```

var Utils = SC.Utils = {
    ...
    , deepCopy: deepCopy
    , getDecodedURLParameter: getDecodedURLParameter
};

```

- Locate the following line in the `parseUrlOptions` function:

```
options[current_token[0]] = decodeURIComponent(current_token[1]);
```

Replace this line with the following line:

```
options[current_token[0]] = Utils.getDecodedURLParameter(current_token[1]);
```

- Save the file.

Step 2: Prepare the Developer Tools For Your Customization

This step shows how to set up the ns.package.json file for your custom module and modify distro.json to make sure that the Gulp tasks include your modules when you deploy.

1. Open the Modules/extensions/UtilitiesExtension@1.0.0 module directory.

2. Create a file in this directory and name it ns.package.json:

Modules/extensions/UtilitiesExtension@1.0.0/ns.package.json

3. Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  },
  "overrides": {
    "suitecommerce/Utilities@X.Y.Z/JavaScript/Utils.js" : "JavaScript/Utils.js"
  }
}
```

4. Open the distro.json file. This is located in your root directory.

5. Add your custom modules to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/UtilitiesExtension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

6. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, URLs that contain the percent sign symbol load properly.

Users Redirected to Checkout Application Instead of Shopping Application

i Applies to: SuiteCommerce Advanced | Elbrus | Kilimanjaro

In some implementations of Elbrus and Kilimanjaro, users who access the Shopping application are redirected to the login page for the Checkout application. The users stay in the Checkout application after they log in. Users should be able to access the Shopping application, log in, and stay in the Shopping application. This error only occurs for websites that have the **Password-protect Entire Site** field in the Configuration record for the web site enabled.

To implement this patch, you must override **Header.Menu.MyAccount()** in the **Header.Profile.View.js** file of the **Header** module. For an example of the changes needed for this patch, see [UsersRedirectedtoCheckoutApp.zip](#).

i Note: In general, NetSuite best practice is to extend JavaScript using the JavaScript **prototype** object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify a file in a way that you cannot extend, and therefore requires you to use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

i Note: The instructions to install the patch are the same for either the Elbrus or Kilimanjaro releases.

Step 1: Create the Header.Profile.View.js Override File

1. If you have not done so already, create a directory to store your custom module.
2. Open this directory and create a subdirectory to maintain your customizations.

Give this directory a name similar to the module being customized. For example:

Modules/extensions/Header.Extension@1.0.0

3. In your new **Header.Extension@1.0.0** directory, create a subdirectory called **JavaScript**:

Modules/extensions/Header.Extension@1.0.0/JavaScript

4. Copy the following file:

Modules/suitecommerce/Header@X.Y.Z/JavaScript/Header.Profile.View.js

In this example, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

5. Paste a copy in your new module's **JavaScript** directory.

For example: **Modules/extensions/Header.Extension@1.0.0/JavaScript/Header.Profile.View.js**

6. Open your new copy of **Header.Profile.View.js** and locate the following lines in the new file:

```
'Header.Menu.MyAccount': function ()
{
    return new HeaderMenuMyAccountView(this.options);
}
```

```
}
```

7. Replace the lines with the following code:

```
'Header.Menu.MyAccount': function () {
{
    var password_protected_site =
        SC.ENVIRONMENT.siteSettings.siteloginrequired === 'T';
    var profile = ProfileModel.getInstance();
    var isLoggedIn = profile.get('isLoggedIn') === 'T';

    if (!password_protected_site || isLoggedIn) {
        return new HeaderMenuMyAccountView(this.options);
    } else {
        return null;
    }
}
```

8. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open the Modules/extensions/Header.Extension@1.0.0 module directory.
2. Create a file in this directory and name it ns.package.json:
Modules/extensions/Header.Extension@1.0.0/ns.package.json
3. Create a file in this directory and name it ns.package.json.
4. Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ],
    "overrides": {
      "suitecommerce/Header@X.Y.Z/JavaScript/Header.Profile.View.js" : "JavaScript/Header.Profile.View.js"
    }
}
```



Important: You must replace the string X.Y.Z with the version of the module in your implementation of SuiteCommerce Advanced.

5. In distro.json, add your custom modules to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Vinson Release",
  "version": "2.0",
  "buildToolsVersion": "1.2.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
```

```

    "deploy": "DeployDistribution"
},
"modules": {
    "extensions/Header.Extension": "1.0.0",
    "suitecommerce/Account": "2.2.0",
    ...

```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

6. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, users can access the Shopping application and stay there after they log in.

Add to Cart Button Does Not Work If Quality Field Selected

ⓘ Applies to: SuiteCommerce Advanced | Elbrus

In implementations of Elbrus, clicking the **Add to Cart** button once does not actually add the chosen items to the cart. Instead, users must click this button twice. This error only occurs if the focus is on the **Quantity** field.

To implement this patch, you must override the `events` object of the `Cart.AddToCart.Button.View.js` file, located in the `Cart` module. Because of the type of change that you need to make to the View file, you must override the view instead of extending it. You can download the code samples described in this procedure here: [AddtoCartButtonDoesNotWork.zip](#).

ⓘ Note: In general, NetSuite best practice is to extend JavaScript using the JavaScript `prototype` object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify a file in a way that you cannot extend, and therefore requires you to use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1: Create the `Cart.AddToCart.Button.View.js` Override File

This step shows how to override the `Cart.AddToCart.Button.View.js` file, located in the `Cart` module.

1. If you have not done so already, create a directory to store your custom module.
2. Open this directory and create a subdirectory to maintain your customizations.

Give this directory a name similar to the module being customized. For example:

`Modules/extensions/Cart.Extension@1.0.0`

- In your new `Cart.Extension@1.0.0` directory, create a subdirectory called **JavaScript**.

`Modules/extensions/Cart.Extension@1.0.0/JavaScript`

- Copy the following file into the new JavaScript directory:

`Modules/suitecommerce/Cart@X.Y.Z/JavaScript/Cart.AddToCart.Button.View.js`

In this case, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

- Open `Cart.AddToCart.Button.View.js` and make the following change:

Locate the following line in the new file:

```
'click [data-type="add-to-cart"]': 'addToCart'
```

Replace the line with the following lines:

```
'mouseup [data-type="add-to-cart"]': 'addToCart'  
, 'click [data-type="add-to-cart"]': 'addToCart'
```

- Save the file.

Step 2: Prepare the Developer Tools For Your Customization

This step shows how to set up the `ns.package.json` file for your custom module and modify `distro.json` to make sure that the Gulp tasks include your modules when you deploy.

- Open the `Modules/extensions/Cart.Extension@1.0.0` module directory.

- Create a file in this directory and name it `ns.package.json`:

`Modules/extensions/Cart.Extension@1.0.0/ns.package.json`

- Build the `ns.package.json` file using the following code:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  },
  "overrides": {
    "suitecommerce/Cart@X.Y.Z/JavaScript/Cart.AddToCart.Button.View.js" : "JavaScript/Cart.AddToCart.Button.View.js"
  }
}
```

- In `distro.json`, add your custom modules to the `modules` object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
```

```

    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
},
"modules": {
    "extensions/Cart.Extension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
...

```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

5. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, the **Add to Cart** button works with one click.

URLs with Redundant Facets Generated

i Applies to: SuiteCommerce Advanced | Elbrus

This topic applies to the Elbrus release of SuiteCommerce Advanced (SCA).

When searching for an item using facets, the application can create a URL containing redundant facets. This error occurs because new facets are appended to the existing URL with a forward slash (/). As a result, the appended URL is relative rather than absolute. For example, the following filter can appear for a web site:

```
http://<baseUrl>/Girls/Mens/Girls/Mens
```

This patch extends the `getUrl()` method in `Facets.Translator.js` for the **Facets** module. The code changes ensure that each URL is prepended with a forward slash (if it does not already have one), making it an absolute URL.

To implement this patch, you must extend the `prototype` object for the `getURL()` method. You can download the code samples described in this procedure here: [RandomFacetsGenerated.zip](#).



Note: Before proceeding, familiarize yourself with [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend the `Facets.Translator.js` File

This step explains how to extend the `Facets.Translator.js` file, located in the **Facets** module.

1. Open the `Modules/extensions` directory and create a subdirectory to maintain your customizations. Give this directory a name similar to the module being customized. For example, create `Modules/extensions/FacetsExtension@1.0.0`

2. In your new FacetsExtension@1.0.0 directory, create a subdirectory called **JavaScript**.

For example: **Modules/extensions/FacetsExtension@1.0.0/JavaScript**

3. In your new JavaScript subdirectory, create a JavaScript file to extend **FacetsTranslator.js**.

Name this file according to best practices. For example:

Facets.Translator.Extension.js

4. Open this file and extend the **getUrl()** method as shown in the following code snippet.

```
define('Facets.Translator.Extension'
  , [ 'Facets.Translator'
    , 'underscore'
    , 'jQuery'
    , 'SC.Configuration'
    , 'UrlHelper'
  ]
  , function (
    FacetsTranslator
  '-
  , jQuery
  , Configuration
  )
{
  'use strict';

  _.extend(FacetsTranslator.prototype,
  {
    // @method getUrl
    // Gets the url for current state of the object
    getUrl: function getUrl ()
    {
      var url = this.categoryUrl || ''
      , self = this;

      // Prepares seo limits
      var facets_seo_limits = {};

      if (SC.ENVIRONMENT.jsEnvironment === 'server')
      {
        facets_seo_limits = {
          number_of_facets_groups: this.configuration.facetsSeoLimits && this.configuration.facetsSeoLimits.number_of_facets_groups || false
          , number_of_facets_values: this.configuration.facetsSeoLimits && this.configuration.facetsSeoLimits.number_of_facets_values || false
          , options: this.configuration.facetsSeoLimits && this.configuration.facetsSeoLimits.options || false
        };
      }

      // If there are too many facets selected
      if (facets_seo_limits.number_of_facets_groups && this.facets.length > facets_seo_limits.number_of_facets_groups)
      {
        return '#';
      }
    }
  }
)
```

```

// Encodes the other Facets
var sorted_facets = _.sortBy(this.facets, 'url')
, facets_as_options = [];

for (var i = 0; i < sorted_facets.length; i++)
{
    var facet = sorted_facets[i];

    // Category should be already added
    if ((facet.id === 'commercecategoryname') || (facet.id === 'category'))
    {
        break;
    }

    var name = facet.url || facet.id
    , value = '';

    switch (facet.config.behavior)
    {
        case 'range':
            facet.value = (typeof facet.value === 'object') ? facet.value : {from: 0, to: facet.value};
            value = facet.value.from +
self.configuration.facetDelimiters.betweenRangeFacetsValues + facet.value.to;
            break;
        case 'multi':
            value = facet.value;
    }
    sort().join(self.configuration.facetDelimiters.betweenDifferentFacetsValues);

        if (facets_seo_limits.numberOfFacetsValues && facet.value.length >
facets_seo_limits.numberOfFacetsValues)
        {
            return '#';
        }

        break;
    default:
        value = facet.value;
    }

    if (self.facetIsParameter(name))
    {
        facets_as_options.push({ facetName: name, facetValue: value });
    }
    else
    {
        // Do not add a facet separator at the begining of an url
        if (url !== '')
        {
            url += self.configuration.facetDelimiters.betweenDifferentFacets;
        }

        url += name + self.configuration.facetDelimiters.betweenFacetNameAndValue + value;
    }
}

```

```

}

url = (url !== '') ? url : '/' + this.configuration.fallbackUrl;

// Encodes the Options
var tmp_options = {}
, separator = this.configuration.facetDelimiters.betweenOptionNameAndValue;

if (this.options.order && this.options.order !== this.configuration.defaultOrder)
{
    tmp_options.order = 'order' + separator + this.options.order;
}

if (this.options.page && parseInt(this.options.page, 10) !== 1)
{
    tmp_options.page = 'page' + separator + encodeURIComponent(this.options.page);
}

if (this.options.show && parseInt(this.options.show, 10) !== this.configuration.defaultShow)
{
    tmp_options.show = 'show' + separator + encodeURIComponent(this.options.show);
}

if (this.options.display && this.options.display !== this.configuration.defaultDisplay)
{
    tmp_options.display = 'display' + separator + encodeURIComponent(this.options.display);
}

if (this.options.keywords && this.options.keywords !== this.configuration.defaultKeywords)
{
    tmp_options.keywords = 'keywords' + separator + encodeURIComponent(this.options.keywords);
}

for (i = 0; i < facets_as_options.length; i++)
{
    var facet_option_obj = facets_as_options[i];

    tmp_options[facet_option_obj.facetName] = facet_option_obj.facetName + separator
+ encodeURIComponent(facet_option_obj.facetValue);
}

var tmp_options_keys = _.keys(tmp_options)
, tmp_options_vals = _.values(tmp_options);

// If there are options that should not be indexed also return #
if (facets_seo_limits.options && _.difference(tmp_options_keys,
facets_seo_limits.options).length)
{
    return '#';
}

url += (tmp_options_vals.length) ? this.configuration.facetDelimiters.betweenFacetsAndOptions
+ tmp_options_vals.join(this.configuration.facetDelimiters.betweenDifferentOptions) : '';

if (url && url[0] !== this.configuration.facetDelimiters.betweenDifferentFacets)

```

```

        {
            url = this.configuration.facetDelimiters.betweenDifferentFacets + url;
        }
        return _(url).fixUrl();
    }
);
});

```

- Save the file.

Step 2: Prepare the Developer Tools For Your Customization

This step shows how to set up the ns.package.json file for your custom module and modify distro.json to make sure that the Gulp tasks include your modules when you deploy.

- Open your new FacetsExtension@1.0.0 module.
- Create a file in this directory and name it ns.package.json:
Modules/extensions/FacetsExtension@1.0.0/ns.package.json
- Build the ns.package.json file using the following code:

```

{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}

```

- In distro.json, add your custom modules to the **modules** object.

Your code should look similar to the following example:

```

{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/FacetsExtension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    // ...
}

```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

- Add **Facets.Translator.Extension** as a dependency to SCA entry point within the **SC.Shopping.Starter** entrypoint of the JavaScript array.

Your Distro.js file should look similar to the following:

```

"tasksConfig": {
//...
"javascript": [
//...
{
    "entryPoint": "SC.Shopping.Starter",
    "exportFile": "shopping.js",
    "dependencies": [
//...
        "Newsletter",
        "ProductDetailToQuote",
        "StoreLocator"
        "Facets.Translator.Extension"
    ],
    //...
}
]
}

```

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.
Upon successful deployment, filters to the products do not contain redundancies.

Content Flickers or Disappears When Browsing the Product Listing Page

 **Applies to:** SuiteCommerce Advanced | Kilimanjaro

In some implementations of Kilimanjaro, content added to the site through SMT may render intermittently or disappear and appear briefly on the product listing page. This error does not occur in SMT but does occur in the published content on the site.

This patch overrides CMSadapter.Component.js in the **CMSadapter** module with the correct code. You can download the code samples described in this procedure here: [Kilimanjaro-ContentFlickerPatch.zip](#).

 **Note:** In general, NetSuite best practice is to extend JavaScript using the JavaScript **prototype** object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify a file in a way that you cannot extend, and therefore requires you to use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1: Create the CMSadapter.Component.js Override File

1. Open the Modules/extensions directory and create a subdirectory to maintain your customizations. Give this directory a name similar to the module being customized. For example, create **Modules/extensions/CMSadapter.Extension@1.0.0**
2. In your new CMSadapter.Extension@1.0.0 directory, create a subdirectory called **JavaScript**. For example: **Modules/extensions/CMSadapter.Extension@1.0.0/JavaScript**

3. Copy the following file:

Modules/suitecommerce/CMSadapter@X.Y.Z/CMSadapter.Component.js

In this example, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

4. Paste a copy in your new module's JavaScript directory.

For example: **Modules/extensions/CMSadapter.Extension@1.0.0/JavaScript/CMSadapter.Component.js**

5. Open your new copy of CMSadapter.Component.js and locate the following lines:

```
var parent_view_instance = cct_container.parent;

parent_view_instance.addChildViewInstances(generator, true);

self._addViewCctsToRerender(parent_view_instance, generator);
```

6. Replace these lines with the following:

```
var placeholders = self.getPlaceholderViews('[data-cms-area="' + container_name + '"]');

if (placeholders && placeholders.views)
{
    _.each(placeholders.views, function(parent_view_instance)
    {
        cct_container.parent = parent_view_instance;
        cct_container.parentView = parent_view_instance;

        parent_view_instance.addChildViewInstances(generator, true);

        self._addViewCctsToRerender(parent_view_instance, generator);
    });
}
```

7. Save the file.

Step 2: Prepare the Developer Tools For Your Customization

1. Open your Modules/extensions/CMSadapter.Extension@1.0.0 module directory.

2. Create a file in this directory and name it ns.package.json.

Modules/extensions/CMSadapter.Extension@1.0.0/ns.package.json

3. Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ],
    "overrides": {
      "suitecommerce/CMSadapter@X.Y.Z/JavaScript/CMSadapter.Component.js" : "JavaScript/CMSadapter.Component.js"
    }
  }
}
```

In this example, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

4. Open the distro.json file. This is located in your root directory.
5. Add your custom modules to the `modules` object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Kilimanjaro",
  "version": "2.0",
  "isSCA": true,
  "buildToolsVersion": "1.3.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/CMSadapter.Extension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

6. Save the distro.json file.

Step 3: Test and Deploy Your Customization

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, the content on the product listing page does not flicker or disappear.

Enabling Google AdWords Causes Error on Login

 **Applies to:** SuiteCommerce Advanced | Vinson

In some implementations of Vinson, Google Ads (previously called Google Adwords) can cause the following error:

`"Uncaught TypeError: e.doCallback is not a function"`

This error can occur when the following conditions are met:

- Google Ads is enabled for a domain.
- A customer adds items to their cart and logs out without completing the transaction.
- The customer then attempts to log in again.

This error occurs in the **Tracker** module. To fix the error, you must extend the **prototype** object of the **Tracker.prototype.trackEvent()** function. For an example of the changes needed for this patch, see [EnablingGoogleAdWordsCausesErroronLogin.zip](#).



Note: Before proceeding, familiarize yourself with [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend the Tracker.js File

1. If you have not done so already, create a directory to store your custom module. Give this directory a name similar to the module being customized. For example, create **Modules/extensions/TrackerExtension@1.0.0**
2. In your new TrackerExtension @1.0.0 directory, create a subdirectory called **JavaScript**. For example: **Modules/extensions/TrackerExtension@1.0.0/JavaScript**
3. In your new JavaScript subdirectory, create a JavaScript file to extend Tracker.js. Name this file according to best practices. For example:
TrackerExtension.js
4. Open this file and extend the **Tracker.prototype.trackEvent()** method as shown in the following code snippet.

```
//@module Tracker
define('Tracker'
  [
    'Singleton'
    , 'underscore'
    , 'Backbone'
  ]
  ,
  function (
    Singleton
    ,
    -
    Backbone
  )
{
  'use strict';

  _.(extend Tracker.prototype,
  Tracker.prototype.trackEvent = function Tracker.prototype.trackEvent (event)
  {
    this.track('trackEvent', event);

    if (event.callback)
    {
      var doCallback = _.find(this.trackers, function (tracker)
      {
        return tracker.doCallback && tracker.doCallback();
      });

      !doCallback && event.callback();
    }

    return this;
  );
}
)
```

```
});
```

- Save the file.

Step 2: Prepare the Developer Tools For Your Customization

- Open your new TrackerExtension@1.0.0 module directory.
- Create a file in this directory and name it ns.package.json.

Modules/extensions/TrackerExtension@1.0.0/ns.package.json

- Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

- Open the distro.json file. This is located in your root directory.
- Add your custom modules to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Vinson Release",
  "version": "2.0",
  "buildToolsVersion": "1.2.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/TrackerExtension": "1.0.0",
    "suitecommerce/Account": "2.2.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

- Add **TrackerExtension** as a dependency to SCA entry point within the **SC.Shopping.Starter** entrypoint of the JavaScript array.

Your distro.js file should look similar to the following:

```
"tasksConfig": {
// ...
"javascript": [
  // ...
  {
    "entryPoint": "SC.Shopping.Starter",
    "exportFile": "shopping.js",
    ...
  }
]
```

```

"dependencies": [
    // ...
    "TrackerExtension",
    "Backbone.View.Plugins",
    "jQuery.html",
    // ...
]

```

- Save the distro.json file.

Step 3: Test and Deploy Your Customization

- Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
- Confirm your results.
Upon successful deployment, enabling Google Ads does not cause errors when a customer logs in or out with items in the cart.

URL for Commerce Categories Contains Incorrect Delimiters

 **Applies to:** SuiteCommerce Advanced | Elbrus

In some implementations of SCA, a URL that uses the Item Search API to access a commerce category contains incorrect delimiters. As a result, accessing pages for a specific category in a SuiteCommerce store returns a 404 error code, even if items for the category exist. For example, a URL might contain %3F instead of the question mark (?) character or %2F instead of forward slash (/).

This patch contains updated code for the **FacetsTranslator** object constructor. To implement this patch, you must override the Facets.Translator.js JavaScript file in the **Facets** module. You can download the code samples described in this procedure here: [IncorrectDelimetersforCommerceCategories.zip](#)

 **Note:** In general, best practice is to extend JavaScript using the JavaScript **prototype** object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify a file that you cannot extend, and you must use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1: Create the `Facets.Translation.js` Override File

This step shows how to override the `Facets.Translator.js` file, located in the **Facets** module.

- If you have not done so already, create a directory to store your custom module.
- Open this directory and create a custom module to maintain your customizations.
Give this module a name similar to the module being customized.
For example:
Modules/extensions/FacetsExtension@1.0.0
- In your new module, create a subdirectory called `JavaScript`.
For example:
Modules/extensions/FacetsExtension@1.0.0/JavaScript
- Copy the following file into the new directory:

Modules/suitecommerce/Facets@X.Y.Z/JavaScript/FacetsTranslator.js

In this case, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

5. Open your new FacetsTranslator.js and make the following change:

Locate the JavaScript code for the **FacetsTranslator** constructor in your new file:

```
//@constructor @param {Array} facets @param {Object} options @param {Object} configuration
function FacetsTranslator (facets, options, configuration, category)
{
    // Enforces new
    if (!(this instanceof FacetsTranslator))
    {
        return new FacetsTranslator(facets, options, configuration, category);
    }

    ...

    else if (facets)
    {
        // It's an API option object
        this.parseOptions(facets);
    }
}
```

Replace these lines with the following code:

```
//@constructor @param {Array} facets @param {Object} options @param {Object} configuration
function FacetsTranslator (facets, options, configuration, category)
{
    // Enforces new
    if (!(this instanceof FacetsTranslator))
    {
        return new FacetsTranslator(facets, options, configuration, category);
    }

    // Facets go Here
    this.facets = [];

    // Other options like page, view, etc. goes here
    this.options = {};

    // This is an object that must contain a fallbackUrl and a lists of facet configurations
    this.configuration = configuration || default_config;

    // Get the facets that are in the sitesettings but not in the config.
    // These facets will get a default config (max, behavior, etc.) - FacetsTranslator
    // Include facet aliases to be considered as a possible route
    var facets_data = Configuration.get('siteSettings.facetfield')
    , facets_to_include = [];

    _.each(facets_data, function(facet)
    {
        if (facet.facetfieldid !== 'commercecATEGORY')
        {
```

```

facets_to_include.push(facet.facetfieldid);
// If the facet has an urlcomponent defined, then add it to the possible values list.
facet.urlcomponent && facets_to_include.push(facet.urlcomponent);

// Finally, include URL Component Aliases...
_.each(facet.urlcomponentaliases, function(facet_alias)
{
  facets_to_include.push(facet_alias.urlcomponent);
});
}

facets_to_include = _.union(facets_to_include, _.pluck(Configuration.get('facets'), 'id'));
facets_to_include = _.uniq(facets_to_include);

this.facetsToInclude = facets_to_include;

this.isCategoryPage = !!category;

if (_.isBoolean(category) && category)
{
  var index = facets.length
  , facetsToInclude = this.facetsToInclude.slice(0);

  facetsToInclude.push(this.configuration.facetDelimiters.betweenFacetsAndOptions);

  _.each(facetsToInclude, function(facetname)
  {
    var i = facets.indexOf(facetname);

    if (i !== -1 && i < index)
    {
      index = i;
    }
  });
}

var categoryUrl = facets.substring(0, index);

facets = facets.substring(index);

if (categoryUrl[0] !== '/')
{
  categoryUrl = '/' + categoryUrl;
}

if (categoryUrl[categoryUrl.length - 1] === '/')
{
  categoryUrl = categoryUrl.substring(0, categoryUrl.length - 1);
}

this.categoryUrl = categoryUrl;
}

else if (_.isString(category))
{

```

```

        this.categoryUrl = category;
    }

    // We cast on top of the passed in parameters.
    if (facets && options)
    {
        this.facets = facets;
        this.options = options;
    }

    else if (_.isString(facets))
    {
        // It's a url
        this.parseUrl(facets);
    }

    else if (facets)
    {
        // It's an optional API object
        this.parseOptions(facets);
    }
}

```

6. Save the file.

Step 2. Prepare the Developer Tools For Your Override

The next step shows how to create the `ns.package.json` file for the custom module and shows how to modify the `distro.json` file to make sure that the Gulp tasks include your modules when you deploy.

1. Open the Modules/extensions/FacetsExtension@1.0.0 module directory.
 2. Create a file in this directory and name it `ns.package.json`.
- `Modules/extensions/FacetsExtension@1.0.0/ns.package.json`
3. Build the `ns.package.json` file using the following code (where X.Y.Z indicates the current version of your module in SuiteCommerce):

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ],
    "overrides": {
      "suitecommerce/Facets@X.Y.Z/JavaScript/Facets.Translations.js" :
      "JavaScript/Facets.Translations.js"
    }
  }
}
```

4. Save the `ns.package.json` file.
5. Open the `distro.json` file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

6. In the `distro.json` file, add your custom modules to the `modules` object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/FacetsExtension": "1.0.0",
    ...
  }
}
```

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

7. Save the distro.json file.

Step 3: Test and Deploy Your Extension

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, the URL used to access products by category includes the proper delimiters and shows the products instead of a 404 error.

Order Summary for Item-Based Promotions

i Applies to: SuiteCommerce Advanced | Elbrus

If an order includes an item with an item-based promotion, the Order Summary does not properly display the price for the items. The Shopping Cart correctly displays the price but the price in the Order Summary is incorrect .

This patch updates the `detail3` variable in the JavaScript files used to display the item price after discount in the Order Summary. You can download the code samples described in this procedure here: [FixTotalInOrderSummary.zip](#) in the File Cabinet.

i Note: In general, best practice is to extend JavaScript using the JavaScript `prototype` object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify a file that you cannot extend, and you must use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

To apply this patch, you override the modules used to display the total in the Order Summary.

Step 1: Create and Update the Required Override Files

1. If you have not done so already, create a directory to store your custom module, for example, `Modules\extensions`.

2. Open this directory and create the following subdirectories to maintain your customizations.

Give these directories a name similar to the modules being customized.

For example, create the following directories in the Modules/extensions directory:

- CreditMemoExtension@1.0.0\JavaScript
- InvoiceExtension@1.0.0 \JavaScript
- OrderWizard.Module.CartItemsExtension@1.0.0 \JavaScript
- OrderWizard.Module.ShipmethodExtension@1.0.0 \JavaScript
- QuoteExtension@1.0.0 \JavaScript
- ReturnAuthorizationExtension@1.0.0 \JavaScript

3. Copy the following source files into their respective target directories, where the target directory looks like the following:

`Modules/extensions/[module]Extension@1.0.0/JavaScript`

4. Copy the files in the following list:

- `Modules/suitecommerce/CreditMemo@X.Y.Z/JavaScript/CreditMemo.Details.View.js`
- `Modules/suitecommerce/Invoice@X.Y.Z/JavaScript/Invoice.Details.View.js`
- `Modules/suitecommerce/OrderHistory@X.Y.Z/JavaScript/OrderHistory.ReturnAuthorization.View.js`
- `Modules/suitecommerce/OrderWizard.Module.CartItems@X.Y.Z/JavaScript/OrderWizard.Module.CartItems.PickupInStore.Package.View.js`
- `Modules/suitecommerce/OrderWizard.Module.CartItems@X.Y.Z/JavaScript/OrderWizard.Module.CartItems.Ship.js`
- `Modules/extensions/OrderWizard.Module.Shipmethod.Extension@X.Y.Z/JavaScript/OrderWizard.Module.MultiShipTo.Shipmethod.Package.View.js`
- `Modules/suitecommerce/Quote@X.Y.Z/JavaScript/Quote.Details.View.js`
- `Modules/suitecommerce/ReturnAuthorization@X.Y.Z/JavaScript/ReturnAuthorization.Detail.View.js`
- `Modules/suitecommerce/ReturnAuthorization@X.Y.Z/JavaScript/ReturnAuthorization.Form.View.js`



Note: Remember, X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

5. For each of the target files in the table listed above, make the following code change.

Locate the following line in the new file:

```
, detail3: 'amount_formatted'
```

Replace this line with the following code:

```
, detail3: 'total_formatted'
```

Step 2. Prepare the Developer Tools for Your Overrides

1. Create an `ns.package.json` file in each of the following directories:
- `CreditMemoExtension@1.0.0`

- InvoiceExtension@1.0.0
 - OrderWizard.Module.CartItemsExtension@1.0.0
 - OrderWizard.Module.ShipmethodExtension@1.0.0
 - QuoteExtension@1.0.0
 - ReturnAuthorizationExtension@1.0.0
2. In each ns.package.json file, enter the following code, where <module name> is the name of the module you are overriding and <file name> is the name of the JavaScript file that contains the overridden JavaScript code:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*"
    ],
  },
  "overrides": {
    "suitecommerce/<module name>@X.Y.Z/JavaScript/<file name>" :
      "extensions/JavaScript/<file name>",
  }
}
```

3. Update the distro.json file in the root directory with the following code for each of the modules you are overriding:

Your code should look similar to the following example:

```
...
},
"extensions/CreditMemoExtension" : "X.Y.Z",
"extensions/InvoiceExtension" : "X.Y.Z"
"extensions/OrderHistoryExtension" : "X.Y.Z"
"extensions/OrderWizard.Module.CartItems.Extension" : "X.Y.Z"
"extensions/OrderWizard.Module.CartItems.Extension" : "X.Y.Z"
"extensions/QuoteExtension" : "X.Y.Z"
"extensions/ReturnAuthorizationExtension" : "X.Y.Z"
"extensions/MyExampleCartExtension1": "1.0.0",
...
```

X.Y.Z represents the version of the module in your implementation of SuiteCommerce Advanced.

Step 3. Test and Deploy Your Override

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)).
2. Confirm your results. The prices in the Shopping Cart now correctly match the prices in the Order Summary.

CSS Error Hides First div Element on Product Details Page

i Applies to: SuiteCommerce Advanced | Mont Blanc | Vinson

In some implementations of SCA, the CSS hides the first **div** element, which typically includes the product description on the product details page. As a result, the content in the product description is excluded from the output of the SEO Page Generator, which can affect the SEO ranking for the product.

This patch adds an additional check to validate that the Page Generator generates the content and then unhides the **div**. To implement this patch, you must override a view file in the **ItemDetails** module. You can download the code samples described in this procedure here: [CSSHidesFirstDiv.zip](#).



Note: In general, best practice is to extend JavaScript using the JavaScript **prototype** object. This ensures the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify a file that you cannot extend, and you must use a custom module to override the existing module file. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1. Override the Item.Details.View.js File

In this step, you override the Item.Details.View.js file, located in the **ItemDetails** module.

1. If you have not done so already, create a directory in which to store your custom modules.
For example, Modules/extensions.
2. Open this directory and create a subdirectory to maintain your customizations.
Give this directory a name similar to the module being customized.
For example, create **Modules/extensions/ItemDetailsExtension@1.0.0**
3. In your new **ItemDetailsExtension@1.0.0** directory, create a subdirectory called **JavaScript**.
4. Copy the following file into this directory:
Modules/suitecommerce/ItemDetails@X.Y.Z/JavaScript/Item.Details.View.js
5. Open your new Item.Details.View.js file and make the following change:

Locate the following JavaScript code:

```
if(!overflow)
{
    this$('.item-details-more').hide();
}
```

Replace it with the following JavaScript code:

```
if(!overflow && !SC.isPageGenerator())
{
    this$('.item-details-more').hide();
}
```

6. Save Item.Details.View.js .

Step 2. Prepare the Developer Tools for Your Override

The next step shows how to set up the ns.package.json file for your custom module and modify distro.json to include the custom module.

1. Open the Modules/extensions/ItemDetailsExtension@1.0.0 module directory.
 2. Create a file in this directory and name it **ns.package.json**.
- Modules/extensions/ItemDetailsExtension@1.0.0/ns.package.json**
3. Build the ns.package.json file using the following code (where X.Y.Z indicates the current version of your module in SuiteCommerce) :

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*"
    ],
    "overrides": {
      "suitecommerce/ItemDetails@X.Y.Z/JavaScript/Item.Details.View.js" : JavaScript/
      Item.Details.View.js",
    }
  }
}
```

4. Save the ns.package.json file.
5. Open the distro.json file.
This file is located in the top-level directory of your SuiteCommerce Advanced source code.
6. In distro.json , add your custom module to the **modules** object.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Vinson Release",
  "version": "2.0",
  "buildToolsVersion": "1.2.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/ItemDetailsExtension": "X.Y.Z",
    ...
  }
}
```

7. This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the **modules** object. The order of precedence in this list does not matter.

Step 3. Test and Deploy Your Override

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, all the required **div** elements appear in the Product Details Page.

Invoice Terms Not Included In Order Details

ⓘ Applies to: SuiteCommerce Advanced | pre-Denali

In Reference My Account v1.05, if a shopper checks out using Invoice (Terms), the order in My Account does not include the invoice terms under **Payment Method** in the order details.

This patch updates the **setPaymentMethod** function in Models.js . To implement this patch, make a copy of Models.js and update it to use the **terms** object to set the payment terms in Reference My Account.

To customize the Models.js file:

1. If you have not done so already, copy the Models.js file from the reference folder to your custom folder in the NetSuite File Cabinet.
Models.js is located in the File Cabinet at My Account 1.05 > Reference My Account > ssp_libraries.
2. Update the **setPaymentMethod** function in the Models.js file from your custom folder:
Locate the following code in Models.js:

```
if (record.getFieldValue('terms'))
{
    paymentmethod.type = 'invoice';

    paymentmethod.purchasenumber = record.getFieldValue('otherrefnum');

    paymentmethod.paymentterms = {
        internalid: record.getFieldValue('terms')
        , name: record.getText('terms')
    };
}
```

Replace these lines with the following code:

```
if (terms)
{
    paymentmethod.type = 'invoice';

    paymentmethod.purchasenumber = record.getFieldValue('otherrefnum');

    paymentmethod.paymentterms = {
        internalid: record.getFieldValue('terms') || terms
        , name: record.getText('terms') || nlapiLookupField(record.recordType
        , record.getFieldValue('id'), 'terms', true)
    };
}
```

3. Configure your SSP Application to use the custom Models.js file.
 1. Open the My Account SSP application record at Setup > SuiteCommerce Advanced > SSP Applications.
 2. On the **Scripts** subtab, update the **Libraries** list to include the Models.js file from your custom folder.

This file must maintain the correct position at the bottom of the list of library files.

3. Click Save.
4. Verify that the customization was successful.
If a shopper checks out with Invoice (Terms) for an order, the payment method appears correctly under **Payment Method** for the order details in My Account.

Users Required to Re-enter Credit Card Payment Method Details on Payment Page

ⓘ Applies to: SuiteCommerce Advanced | Elbrus | Kilimanjaro

This section applies to the **Elbrus and Kilimanjaro** releases of SuiteCommerce Advanced.

In some cases, users making a purchase must re-enter credit card payment method details on the payment page to successfully place an order.

ⓘ Note: Before proceeding, familiarize yourself with the [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend the OrderWizard.Module.PaymentMethod.Creditcard.js File

This step explains how to extend the OrderWizard.Module.PaymentMethod.Creditcard.js file, which is located in the **OrderWizard.Module.PaymentMethod@x.y.z** module. You can download the code samples described in this procedure here: [OrderWizardPaymentMethodExtension-Elbrus and Kilimanjaro.zip](#)

1. If you have not done so already, create a directory to store your custom module.
Following best practices, name this directory **extensions** and place it in your Modules directory. Depending on your implementation and customizations, this directory might already exist.
2. Open your extensions directory and create a custom module to maintain your customizations.
Give this directory a unique name that is similar to the module being customized. For example:
Modules/extensions/OrderWizardPaymentMethodExtension@1.0.0
3. In your new module, create a subdirectory called JavaScript.
4. In your new JavaScript subdirectory, create a JavaScript file.
Give this file a unique name that is similar to the file being modified. For example:
OrderWizard.Module.PaymentMethod.Creditcard.Extension.js
5. Open this file and overwrite the following method as described below:
 - **_submit()**

Your file should match the following code snippet:

```
define(
  'OrderWizard.Module.PaymentMethod.Creditcard.Extension'
, [
  'Utils'
, 'OrderWizard.Module.PaymentMethod'
, 'jQuery'
, 'Profile.Model'
]
, function (
  Utils
, OrderWizardModulePaymentMethod
```

```

    ,  jQuery
    ,  ProfileModel  )
{
  'use strict';
  _.extend(OrderWizardModulePaymentMethodCreditcard.prototype,
  // @method submit
  submit: function ()
  {
    // This order is being payed with some other method (Gift Cert probably)
    if (this.wizard.hidePayment())
    {
      return jQuery.Deferred().resolve();
    }

    var self = this;

    if (this.requireccsecuritycode)
    {
      this.isSecurityNumberInvalid = false;
      // we need to store this temporally (frontend) in case a module in the same step
      // fails validation, making the credit card section re-rendered.
      // We don't want the user to have to type the security number multiple times
      this.ccsecuritycode = this.$('input[name="ccsecuritycode"]').val();
    }

    // if we are adding a new credit card
    if (this.creditcardView)
    {
      var fake_event = jQuery.Event('click', {
        target: this.creditcardView.$('form').get(0)
      })
      , result = this.creditcardView.saveForm(fake_event);

      if (!result || result.frontEndValidationErrors)
      {
        // There were errors so we return a rejected promise
        return jQuery.Deferred().reject({
          errorCode: 'ERR_CHK_INCOMPLETE_CREDITCARD'
          , errorMessage: _('The Credit Card is incomplete').translate()
        });
      }

      var save_result = jQuery.Deferred();

      result.done(function (model)
      {
        self.creditcardView = null;

        ProfileModel.getInstance().get('creditcards').add(model, {
          silent: true
        });

        self.setCreditCard({
          model: model
        });
      });
    }
  }
}

```

```

    });

        save_result.resolve();
    }).fail(function (error)
    {
        save_result.reject(error.responseJSON);
    });

    return save_result;
}
// if there are already credit cards
else
{
    this.setSecurityNumber();

    OrderWizardModulePaymentMethod.prototype.submit.apply(this, arguments);

    this.setCreditCard({
        model: this.paymentMethod.get('creditcard')
    });

    return this.isValid().fail(function (error)
    {
        if (error === self.securityNumberErrorMessage)
        {
            self.isSecurityNumberInvalid = true;
        }
    })
    .done(function ()
    {
        self.isSecurityNumberInvalid = false;

    })
    .always(function ()
    {
        // Call self.render() instead of self._render() because the last one didn't assign the
        events to the DOM
        self.render();
    });
}
});
});

```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Customizations

1. Open the OrderWizardPaymentMethodExtension@1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.
Modules/extensions/OrderWizardPaymentMethodExtension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "javascript": [

```

```

        "JavaScript/*.js"
    ]
}
}
```

4. Save the ns.package.json file.

5. Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

6. Add your custom module to the `modules` object to ensure that the Gulp tasks include it when you deploy.

In this example, the `extensions/OrderWizardPaymentMethodExtension` module is added at the beginning of the list of modules. However, you can add the module anywhere in the `modules` object. The order of precedence in this list does not matter.

```
{
  "name": "SuiteCommerce Advanced Elbrus Release",
  "version": "2.0",
  "buildToolsVersion": "1.2.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/OrderWizardPaymentMethodExtension": "1.0.0",
    "extensions/Configuration.Extension": "1.0.0",
    "extensions/SiteSettings.Extension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    "extensions/MyExamplePDPExtension1": "1.0.0",
    "extensions/MyExamplePDPExtension2": "1.0.0",
    "suitecommerce/Account": "2.3.0",
    ...
  }
}
```

7. Include the module definition

(`OrderWizard.Module.PaymentMethod.Creditcard.Extension`) in the dependencies array of the `javascript` object within the `SC.Checkout.Starter` entryPoint. You must include this file after the module you are extending. Your distro.json file should look similar to the following:

```
//...
"tasksConfig": {
//...
"javascript": [
  {
//...
    "entryPoint": "SC.Checkout.Starter",
    "exportFile": "checkout.js",
    "dependencies": [
//...
      "StoreLocator",
      "OrderWizard.Module.PaymentMethod.Creditcard.Extension"
    ],
  }
]
```

//...

8. Save the distro.json file.

Step 3: Deploy Your Extension

1. Deploy your source code customizations to your NetSuite account and test the functionality. See [Deploy to NetSuite](#) for details.



Note: Since this patch modifies a SuiteScript file, changes are not visible in your local environment. SuiteScript files run on the server and must be deployed to NetSuite to take effect.

2. Confirm your results.

After a successful deployment, your site properly completes the purchase process using the credit card payment method.

Selected Invoice Not Displayed When Making an Invoice Payment

Applies to: SuiteCommerce Advanced | Mont Blanc

In some cases, users making an invoice payment experience an issue where the **Make a Payment** page does not display the selected invoice. Instead, users receive the following message:

You don't have any Open Invoices at the moment, see Invoices Paid in Full

Perform the following steps to resolve this problem:

- Extend the `LivePayment.Model.js` file
- Prepare the Developer Tools for Your Customizations
- Deploy your extension

You can download the code samples described in this procedure here: [MakePaymentPageNotDisplaying-MontBlanc.zip](#).

Step 1: Extend the `LivePayment.Model.js` File



Note: The `LivePayment.Model.js` file is located within the `Modules/suitecommerce/LivePayment@x.x.x/SuiteScript` folder

1. If you have not done so already, create a directory to store your custom module.
Following best practices, name this directory **extensions** and place it in your `Modules` directory. Depending on your implementation and customizations, this directory might already exist.
2. Open your `extensions` directory and create a custom module to maintain your customizations.
Give this directory a unique name that is similar to the module being customized. For example:
`Modules/extensions/LivePayment.Extension@1.0.0/`
3. In your new module, create a subdirectory named **SuiteScript**.
4. In your `SuiteScript` subdirectory, create a new JavaScript file.

Give this file a unique name that is similar to the file being modified. For example:

`Modules/extensions/LivePayment.Extension@1.0.0/SuiteScript/
LivePayment.Model.Extension.js`

5. Open this file and set it up to overwrite the `setInvoices()` method of the `LivePayment.Model.js` file.

Your file should match the following code snippet:

```
define(
  'LivePayment.Model.Extension'
, ['LivePayment.Model', 'Utils', 'underscore']
, function (LivePaymentModel, Utils, _)
{
  'use strict';

  _.extend(LivePaymentModel.prototype,
  {
    setInvoices: function(customer_payment, result)
    {
      result.invoices = [];

      var invoice_ids_to_search = [];

      for (var i = 1; i <= customer_payment.getLineItemCount('apply'); i++)
      {
        if (customer_payment.getLineItemValue('apply', 'type', i) === 'Invoice')
        {
          var invoice = {
            internalid: customer_payment.getLineItemValue('apply', 'internalid', i)
          , total: Utils.toCurrency(customer_payment.getLineItemValue('apply', 'total', i))
          , total_formatted: Utils.formatCurrency(customer_payment.getLineItemValue('apply', 'total', i))
          , apply: customer_payment.getLineItemValue('apply', 'apply', i) === 'T'
          , applydate: customer_payment.getLineItemValue('apply', 'applydate', i)
          , currency: customer_payment.getLineItemValue('apply', 'currency', i)
          , discamt: Utils.toCurrency(customer_payment.getLineItemValue('apply', 'discamt', i))
          , discamt_formatted: Utils.formatCurrency(customer_payment.getLineItemValue('apply', 'discamt', i))
          , disc: Utils.toCurrency(customer_payment.getLineItemValue('apply', 'disc', i))
          , disc_formatted: Utils.formatCurrency(customer_payment.getLineItemValue('apply', 'disc', i))
          , discdate: customer_payment.getLineItemValue('apply', 'discdate', i)
          , amount: Utils.toCurrency(customer_payment.getLineItemValue('apply', 'amount', i))
          , amount_formatted: Utils.formatCurrency(customer_payment.getLineItemValue('apply', 'amount', i))
          , due: Utils.toCurrency(customer_payment.getLineItemValue('apply', 'due', i))
          , due_formatted: Utils.formatCurrency(customer_payment.getLineItemValue('apply', 'due', i))
          , tranid: customer_payment.getLineItemValue('apply', 'refnum', i)
        };
        result.invoices.push(invoice);
        invoice_ids_to_search.push(invoice.internalid);
      }
    }
  }
)
```

```

        return result;
    }

});
});

```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Customizations

1. Open the LivePayment.Extension @1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.

`Modules/extensions/LivePayment.Extension@1.0.0/ns.package.json`

3. Build the ns.package.json file using the following code:

```
{
  "gulp": {
    "ssp-libraries": [
      "SuiteScript/*.js"
    ]
  }
}
```

4. Save the ns.package.json file.

5. Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

6. Add your custom module to the **modules** object to ensure that the Gulp tasks include your extension when you deploy.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Mont Blanc",
  "version": "2.0",
  "buildToolsVersion": "1.1.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/LivePayment.Extension": "1.0.0",
    "suitecommerce/Account": "2.1.0",
    ...
  }
}
```

7. Add **LivePayment.Model.Extension** as a dependency to SCA entry point within the **ssp-libraries** object:

Your code should look similar to the following example:

```
//...
  "ssp-libraries": {
    "entryPoint": "SCA",
```

```

"dependencies": [
    "Application",
    /**
     *LivePayment.Model.Extension"
    /**
],
/**

```

- Save the distro.json file.

Step 3: Deploy Your Extension

- Deploy your customizations to your NetSuite account. See [Deploy to NetSuite](#).



Note: Since this patch modifies a SuiteScript file, changes are not visible in your local environment. SuiteScript files run on the server and must be deployed to NetSuite.

- Confirm your results.

Upon successful deployment, invoices selected for payment should be visible on the Make a Payment page.

Log In to See Prices Message Appears When Users are Logged In

Applies to: SuiteCommerce Advanced | Elbrus

If you use the Log In to See Prices feature on your Elbrus implementation of SuiteCommerce Advanced, you might experience an issue where the application prompts users to log in to see prices even though they have successfully logged into your site.

To correct this issue, apply the patch described here. Due to the nature of this change, the best practice is to override the SC.Configuration.js file. You can download the code samples described in this procedure here: [LogInToSeePricesPatch--Elbrus.zip](#).



Important: These downloadable code samples assume that you have not made any previous customizations to the SC.Configuration.js file. The Override method refers to making changes by replacing the functionality of an entire file with your own custom version. This can potentially introduce errors when you deploy your code. Be aware of any existing customizations to this file and familiarize yourself with the [Best Practices for Customizing SuiteCommerce Advanced](#) before overriding any files.

Step 1: Override the SC.Configuration.js File

- Create a directory to store your custom module.

Following best practices, name this directory **extensions** and place it in your Modules directory. Depending on your implementation and customizations, this directory might already exist.

- Open your extensions directory and create a custom module to maintain your customizations.

Give this directory a unique name that is similar to the module being customized. For example:

Modules/extensions/SCA.Extension@1.0.0/

3. In your new module, create a subdirectory named JavaScript.
4. Copy the source SC.Configuration.js file located in Modules > suitecommerce > SCA@x.y.z/JavaScript/ and paste into your new JavaScript directory (where x.y.z represents the version of the module in your implementation of SuiteCommerce Advanced).

For example, paste your copy of this into the following location:

Modules/extensions/SCA.Extension@1.0.0/JavaScript/SC.Configuration.js

5. Open locate the following lines:

```
if(entry.placeholder)
{
    entry.text = '';
}
entry['class'] = 'header-menu-level' + entry.level + '-anchor';
```

6. Replace these lines with the following code:

```
if(entry.dataTouchpoint) entry['data-touchpoint']=entry.dataTouchpoint;
if(entry.dataHashtag) entry['data-hashtag']=entry.dataHashtag;
if(entry.placeholder)
{
    entry.text = '';
}
entry['class'] = 'header-menu-level' + entry.level + '-anchor';
```

When you are finished, your custom file should include the following code:

```
// ...

// navigation hierarchy bindings.
_.each(baseConfiguration.navigationData, function (entry)
{
    if (!entry)
    {
        return;
    }
    else
    {
        if(entry.dataTouchpoint) entry['data-touchpoint']=entry.dataTouchpoint;
        if(entry.dataHashtag) entry['data-hashtag']=entry.dataHashtag;
        if(entry.placeholder)
        {
            entry.text = '';
        }
        entry['class'] = 'header-menu-level' + entry.level + '-anchor';
    }
    if (entry.parentId)
    {
        var parent = _.find(baseConfiguration.navigationData, function (e)
        {
            return e.id==entry.parentId;
        });
        parent = parent || {};
        parent.categories = parent.categories || [];
        parent.categories.push(entry);
    }
});
```

```

    }
    if (entryclassnames)
    {
        entry['class'] += ' ' + entryclassnames;
    }
});

//...

```

7. Save the file.

Step 2: Prepare the Developer Tools for Your Override

1. Open the SCA.Extension@1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.
Modules/extensions/SCA.Extension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code, where x.y.z equals the version of the module you are overriding.

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ],
    "overrides": {
      "suitecommerce/SCA@x.y.z/JavaScript/SC.Configuration.js" : "JavaScript/SC.Configuration.js"
    }
  }
}
```

Note: Replace the string x.y.z in the above example with the version of the module in your version of SuiteCommerce Advanced.

4. Save the ns.package.json file.
5. Open the distro.json file.
This file is located in the top-level directory of your SuiteCommerce Advanced source code.
6. Add your module to the **modules** object to ensure that the Gulp tasks include it when you deploy.
Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/SCA.Extension": "1.0.0",
  }
}
```

```
"extensions/MyExampleCartExtension1": "1.0.0",
// ...
```

7. Save the file.

Step 3:

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.
Upon successful deployment, the application no longer requests that logged-in users need to log in to see prices.

Item Record HTML Meta Data Not Appearing on Page Meta Data

 **Applies to:** SuiteCommerce Advanced | Elbrus

Elbrus implementations of SuiteCommerce Advanced experience an issue where the value of an Item record's **Meta Tag HTML** field does not appear in the page's meta data. To correct this issue, create a custom module to extend ProductDetails.Base.Views.js as described in this section.

In addition to making these changes, you must create an ns.package.json file and update your distro.json file for any custom modules you include. You can download the code samples described in this procedure here: [HTMLMetaTagsNotAppearingPatch--Elbrus.zip](#).

Step 1: Extend the ProductDetails.Base.Views.js File

1. If you have not done so already, create a directory to store your custom module.
Following best practices, name this directory **extensions** and place it in your Modules directory. Depending on your implementation and customizations, this directory might already exist.
2. Open your extensions directory and create a custom module to maintain your customizations.
Give this directory a unique name that is similar to the module being customized. For example:
Modules/extensions/ProductDetails.Extension@1.0.0/
3. In your new module, create a subdirectory named JavaScript.
Give this file a unique name that is similar to the file being modified. For example:
**Modules/extensions/ProductDetails.Extension@1.0.0/
ProductDetails.Base.View.Extension.js**
4. In your JavaScript subdirectory, create a new JavaScript file.
Your file should match the following code snippet:

```
define(
    'ProductDetails.Base.View.Extension'
, [
    'ProductDetails.Base.View'
    , 'jQuery'
```

```

        , 'underscore'
    ]
    function (
        ProductDetailsBaseView

        , jQuery
        ,
    )
{
    'use strict';

    return {
        mountToApp: function ()
        {
            _.extend(ProductDetailsBaseView.prototype,
            {
                getMetaTags: function getMetaTags ()
                {
                    return jQuery('<head/>').html(
                        jQuery.trim(
                            this.model.get('item').get('_metaTags')
                        )
                    ).children('meta');
                }
            });
        }
    );
}
);

```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Customizations

1. Open the ProductDetails.Extension@1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.
Modules/extensions/ProductDetails.Extension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code

```
{
    "gulp": {
        "javascript": [
            "JavaScript/*.js"
        ]
    }
}
```

4. Save the ns.package.json file.
5. Open the distro.json file.
This file is located in the top-level directory of your SuiteCommerce Advanced source code.
6. Add your custom module to the **modules** object to ensure that the Gulp tasks include it when you deploy.

Your code should look similar to the following example:

```
{
```

```

    "name": "SuiteCommerce Advanced Elbrus",
    "version": "2.0",
    "buildToolsVersion": "1.3.0",
    "folders": {
        "modules": "Modules",
        "suitecommerceModules": "Modules/suitecommerce",
        "extensionsModules": "Modules/extensions",
        "thirdPartyModules": "Modules/third_parties",
        "distribution": "LocalDistribution",
        "deploy": "DeployDistribution"
    },
    "modules": {
        "extensions/ProductDetails.Extension": "1.0.0",
        "extensions/MyExampleCartExtension1": "1.0.0",
        /**
        */
    }
}

```

7. Include the module definition (`ProductDetails.Base.View.Extension`) as a dependency within the JavaScript `SC.Shopping.Starter` entry point.

Your distro.json file should look similar to the following:

```

"tasksConfig": {
//...
    "javascript": [
        {
            "entryPoint": "SC.Shopping.Starter",
            "exportFile": "shopping.js",
            "dependencies": [
                /**
                */
                "StoreLocator",
                "ProductDetails.Base.View.Extension"
            ]
            /**
            */
        }
    ]
}

```

8. Save the distro.json file.

Step 3: Test and Deploy Your Extension

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.
Upon successful deployment, your Item record Meta Tag HTML value should appear in the meta data within the DOM of the associated product details page.

Delivery Options Not Appearing After Editing the Cart and Re-entering a Shipping Address

ⓘ Applies to: SuiteCommerce Advanced | Elbrus

In some cases, delivery method options do not display after editing the cart. If a user performs the following series of events, the delivery method options do not display properly:



Note: This issue applies to Site Builder Extensions implementations only.

1. The user adds an item to the cart and proceeds to checkout.
2. The user enters a shipping address, including a zip code.
3. The user clicks out of the zip code field and clicks **Edit Cart**.
4. Without making any changes, the user again proceeds to Checkout.
5. The user enters the same shipping information, including the same zip code.

If you experience this issue on your Site Builder Extensions implementation, create a custom module to extend OrderWizard.Module.Shipmethod.js as described in this section.

In addition to making these changes, you must create an ns.package.json file and update your distro.json file for any custom modules you include. You can download the code samples described in this procedure here: [DeliverMethodsNotAppearingPatch---SiteBuilderExtensions.zip](#).

Step 1: Extend the OrderWizard.Module.Shipmethod.js File

1. If you have not done so already, create a directory to store your custom module. Following best practices, name this directory **extensions** and place it in your Modules directory. Depending on your implementation and customizations, this directory might already exist.
2. Open your extensions directory and create a custom module to maintain your customizations. Give this directory a unique name that is similar to the module being customized. For example:
Modules/extensions/OrderWizard.Module.Shipmethod.Extension@1.0.0/
3. In your new module, create a subdirectory named JavaScript.
4. In your JavaScript subdirectory, create a new JavaScript file. Give this file a unique name that is similar to the file being modified. For example:
Modules/extensions/OrderWizard.Module.Shipmethod.Extension@1.0.0/OrderWizard.Module.Shipmethod.Extension.js
5. Open this file and set it up to overwrite the **shipAddressChange()** method of the OrderWizard.Module.Shipmethod.js file.

Your file should match the following code snippet:

```
define(
  'OrderWizard.Module.Shipmethod.Extension'
, [
  'OrderWizard.Module.Shipmethod'

  , 'jQuery'
  , 'underscore'
]

, function (
  OrderWizardModuleShipmethod

  , jQuery
  ,
)
{

  'use strict';
}
```

```

_.extend(OrderWizardModuleShipmethod.prototype,
{
    shipAddressChange: function (model, value)
    {
        this.currentAddress = value;

        var order_address = this.model.get('addresses')
            , previous_address = this.previousAddress && (order_address.get(this.previousAddress)
|| this.addresses.get(this.previousAddress))
            , current_address = this.currentAddress && order_address.get(this.currentAddress)
|| this.addresses.get(this.currentAddress)
            , changed_zip = previous_address && current_address && previous_address.get('zip') !
== current_address.get('zip')
            , changed_state = previous_address && current_address && previous_address.get('state') !
== current_address.get('state')
            , changed_country = previous_address && current_address && previous_address.get('country') !
== current_address.get('country');

        // if previous address is equal to current address we compare the previous values on the
model.
        if (this.previousAddress && this.currentAddress && this.previousAddress
== this.currentAddress)
        {
            changed_zip = current_address.previous('zip') !== current_address.get('zip');
            changed_country = current_address.previous('country') !== current_address.get('country');
            changed_state = current_address.previous('state') !== current_address.get('state');
        }

        // reload ship methods only if there is no previous address or when change the country or
zipcode
        if ((!previous_address && current_address) || changed_zip || changed_country || changed_state)
        {
            // if its selected a valid address, reload Methods
            if (this.model.get('isEstimating') || this.addresses.get(this.model.get('shipaddress'))))
            {
                this.reloadMethods();
            }
        }
        else
        {
            this.render();
        }

        if (value)
        {
            this.previousAddress = value;
        }

        // if we select a new address, bind the sync method for possible address edits
        if (this.currentAddress)
        {
            var selected_address = this.addresses.get(this.currentAddress);
            if(selected_address)
            {

```

```

        selected_address.on('sync', jquery.proxy(this, 'reloadMethods'), this);
    }

    // if there was a different previous address, remove the sync handler
    if(this.previousAddress && this.previousAddress !== this.currentAddress)
    {
        var previous_selected_address = this.addresses.get(this.previousAddress);
        if(previous_selected_address)
        {
            previous_selected_address.off('sync');
        }
    }
});

});

```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Customizations

1. Open the OrderWizard.Module.Shipmethod.Extension@1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.
Modules/extensions/OrderWizard.Module.Shipmethod.Extension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Save the ns.package.json file.
5. Open the distro.json file.
This file is located in the top-level directory of your SuiteCommerce Advanced source code.
6. Add your custom module to the **modules** object to ensure that the Gulp tasks include it when you deploy.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Site Builder - Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "OrderWizard.Module.Shipmethod.Extension": {
      "version": "1.0.0"
    }
  }
}
```

```

    "extensions/OrderWizard.Module.Shipmethod.Extension": "1.0.0",
    "suitecommerce/Account": "2.3.0",
    //...

```

- Include the module definition (`OrderWizard.Module.Shipmethod.Extension`) as a dependency within the JavaScript `SC.Checkout.Starter` entry point.

Your distro.json file should look similar to the following:

```

"tasksConfig": {
//...
    "javascript": [
        {
            "entryPoint": "SC.Checkout.Starter",
            "exportFile": "checkout.js",
            "dependencies": [
                //...
                "BrontoIntegration",
                "OrderWizard.Module.Shipmethod.Extension"
            ]
            //...
        }
    ]
//...
}

```

- Save the distro.json file.

Step 3: Test and Deploy Your Extension

- Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
- Confirm your results.

Upon successful deployment, all delivery options will appear after editing the cart and re-entering a shipping address.

Order Confirmation and Thank You Page is Blank

 **Applies to:** SuiteCommerce Advanced | Mont Blanc | Vinson

In some cases, an Order Confirmation and Thank You page might not appear after a user places an order in your web store. If you experience this issue on your SuiteCommerce Advanced site, perform the following steps to correct the problem.

The suggested action involves extending the `LiveOrderModel.js` file (within the `SuiteScript` folder) to overwrite the `confirmationCreateResult()` method.

In addition to making these changes, you must create an `ns.package.json` file and update your `distro.json` file for any custom modules you include. You can download the code samples described in this procedure here: [LiveOrderModelThankYouPagePatch.zip](#).

Step 1: Extend the `LiveOrderModel.js` File

- If you have not done so already, create a directory to store your custom module.

Following best practices, name this directory **extensions** and place it in your `Modules` directory. Depending on your implementation and customizations, this directory might already exist.

2. Open your extensions directory and create a custom module to maintain your customizations.

Give this directory a unique name that is similar to the module being customized. For example:

Modules/extensions/LiveOrder.Extension@1.0.0/

3. In your new module, create a subdirectory named SuiteScript.

4. In your SuiteScript subdirectory, create a new JavaScript file.

Give this file a unique name that is similar to the file being modified. For example:

**Modules/extensions/LiveOrder.Extension@1.0.0/SuiteScript/
LiveOrder.Model.Extension.js**

5. Open this file and set it up to overwrite the **confirmationCreateResult()** method of the **LiveOrder.Model.js** file.

Your file should match the following code snippet:

```
define(
  'LiveOrder.Model.Extension'
, [
  'LiveOrder.Model'
, 'Utils'
, 'underscore'
]

, function (
  LiveOrderModel
, Utils

)
{
  'use strict';

  _.extend(LiveOrderModel.prototype,
  {
    confirmationCreateResult: function (placed_order)
    {
      var result = {
        internalid: placed_order.getId()
      , tranid: placed_order.getFieldValue('tranid')
      , summary: {
          subtotal: Utils.toCurrency(placed_order.getFieldValue('subtotal'))
        , subtotal_formatted: Utils.formatCurrency(placed_order.getFieldValue('subtotal'))

          , taxtotal: Utils.toCurrency(placed_order.getFieldValue('taxtotal'))
        , taxtotal_formatted: Utils.formatCurrency(placed_order.getFieldValue('taxtotal'))

          , shippingcost: Utils.toCurrency(placed_order.getFieldValue('shippingcost'))
        , shippingcost_formatted: Utils.formatCurrency(placed_order.getFieldValue('shippingcost'))

          , handlingcost: Utils.toCurrency(placed_order.getFieldValue('althandlingcost'))
        , handlingcost_formatted: Utils.formatCurrency(placed_order.getFieldValue('althandlingcost'))
      }
    }
  }
}
```

```

        , discounttotal: Utils.toCurrency(placed_order.getFieldValue('discounttotal'))
        , discounttotal_formatted: Utils.formatCurrency(placed_order.getFieldValue('discounttotal'))


        , giftcertapplied: Utils.toCurrency(placed_order.getFieldValue('giftcertapplied'))
        , giftcertapplied_formatted: Utils.formatCurrency(placed_order.getFieldValue('giftcertapp
lied'))


        , total: Utils.toCurrency(placed_order.getFieldValue('total'))
        , total_formatted: Utils.formatCurrency(placed_order.getFieldValue('total'))
    }
};

result.promocode = (placed_order.getFieldValue('promocode')) ? {
    internalid: placed_order.getFieldValue('promocode')
    , name: placed_order.getFieldText('promocode')
    , code: placed_order.getFieldText('couponcode')
    , isvalid: true
} : null;

result.paymentmethods = [];
for (var i = 1; i <= placed_order.getLineItemCount('giftcertredemption'); i++)
{
    result.paymentmethods.push({
        type: 'giftcertificate'
        , giftcertificate: {
            code: placed_order.getLineItemValue('giftcertredemption', 'authcode_display', i)
            , amountapplied: placed_order.getLineItemValue('giftcertredemption', 'authcodeapplied',
i)
            , amountapplied_formatted: Utils.formatCurrency(placed_order.getLineItemVal
ue('giftcertredemption', 'authcodeapplied', i))
            , amountremaining: placed_order.getLineItemValue('giftcertredemption', 'authcodeamt
remaining', i)
            , amountremaining_formatted: Utils.formatCurrency(placed_order.getLineItemVal
ue('giftcertredemption', 'authcodeamtremaining', i))
            , originalamount: placed_order.getLineItemValue('giftcertredemption', 'giftcertava
ilable', i)
            , originalamount_formatted: Utils.formatCurrency(placed_order.getLineItemVal
ue('giftcertredemption', 'giftcertavailable', i))
        }
    });
}

result.lines = [];

for (var i = 1; i <= placed_order.getLineItemCount('item'); i++)
{
    result.lines.push({
        item: {
            internalid: placed_order.getLineItemValue('item', 'item', i)
            // 'item_display' returns the 'sku' or if is a matrix returns 'sku_parent :
sku_child'
            // getLineItemText of 'item_display' returns the same as getLineItemText of 'item'
        }
    });
}

```

```

        , itemDisplay: placed_order.getLineItemText('item', 'item', i)
    }
    , quantity: parseInt(placed_order.getLineItemValue('item', 'quantity', i), 10)
    , rate: parseInt(placed_order.getLineItemValue('item', 'rate', i), 10)
    , options: placed_order.getLineItemValue('item', 'options', i)
);
}

return result;
}
));
});
);

```

- Save the file.

Step 2: Prepare the Developer Tools for Your Customizations

- Open the LiveOrder.Extension @1.0.0 module.
- Create a file in this module and name it **ns.package.json**.

`Modules/extensions/LiveOrder.Extension@1.0.0/ns.package.json`

- Build the ns.package.json file using the following code

```
{
  "gulp": {
    "ssp-libraries": [
      "SuiteScript/*.js"
    ]
  }
}
```

- Save the ns.package.json file.

- Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

- Add your custom module to the **modules** object to ensure that the Gulp tasks include your extension when you deploy.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Mont Blanc",
  "version": "2.0",
  "buildToolsVersion": "1.1.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/LiveOrder.Extension": "1.0.0",
    "suitecommerce/Account": "2.1.0",
    //...
  }
}
```

- Add `LiveOrder.Model.Extension` as a dependency to `SCA` entry point within the `ssp-libraries` object:

Your code should look similar to the following example:

```
//...
  "ssp-libraries": {
    "entryPoint": "SCA",
    "dependencies": [
      "Application",
      ...
      "LiveOrder.Model.Extension"
    ],
  }
//...
```

- Save the distro.json file.

Step 3: Deploy Your Extension

- Deploy your customizations to your NetSuite account. See [Deploy to NetSuite](#).

Note: Since this patch modifies an SSP library file, changes are not visible in your local environment. SuiteScript files run on the server and must be deployed to NetSuite.

- Confirm your results.

Upon successful deployment, completed orders should result in displaying an Order Confirmation and Thank You page.

Matrix Item Options Not Displaying With Google Tag Manager Enabled

Applies to: SuiteCommerce Advanced | Elbrus

In some cases, not all selected matrix item options appear in the Product Details Page. This can occur on sites using Google Tag Manager. To correct this error, apply the patch described here to extend `GoogleTagManager.js` and overwrite the `trackProductView()` method.

In addition to making these changes, you must create an `ns.package.json` file and update your `distro.json` file for any custom modules you include. You can download the code samples described in this procedure here: [MatrixItemDisplayGTM.zip](#).

Step 1: Extend the `GoogleTagManager.js` File

- If you have not done so already, create a directory to store your custom module. Following best practices, name this directory **extensions** and place it in your Modules directory. Depending on your implementation and customizations, this directory might already exist.
- Open your extensions directory and create a custom module to maintain your customizations. Give this directory a unique name that is similar to the module being customized. For example:
`Modules/extensions/GoogleTagManager.Extension@1.0.0/`
- In your new module, create a subdirectory named JavaScript.
- In your JavaScript subdirectory, create a new JavaScript file.

Give this file a unique name that is similar to the file being modified. For example:

`Modules/extensions/GoogleTagManager.Extension@1.0.0/JavaScript/GoogleTagManager.Extension.js`

5. Open this file and set it up to overwrite the `trackProductView()` method of the `GoogleTagManager.js` file.

Your file should match the following code snippet:

```
define(
  'GoogleTagManager.Extension'
, [
  'GoogleTagManager'
, 'underscore'
]

, function (
  GoogleTagManager
)
{
  'use strict';

  return {
    mountToApp: function ()
    {
      _.extend(GoogleTagManager.prototype,
      {
        trackProductView: function (product)
        {
          var item = product.getItem();

          if (this.item && this.item.get('itemId') === item.get('_id'))
          {
            item.set('category', this.item.get('category'), { silent: true });
            item.set('list', this.item.get('list'), { silent: true });
          }

          var eventName = 'productView'
, selected_options = product.get('options').filter(function (option)
{
  return option.get('value') && option.get('value').label;
})
, price = product.getPrice()
, eventData = {
  'event': eventName
, 'data': {
    'sku': product.getSku()
, 'name': item.get('_name')
, 'variant': _.map(selected_options, function (option)
{

```

```
        return option.get('value').label;
    }).join(', ')
    , 'price': ((price.price) ? price.price : 0).toFixed(2)
    , 'category': item.get('category') || '' // as we do not support categories this is
just the url
    , 'list': item.get('list') || ''
    , 'page': this.getCategory()
}
};

this.item = null;

//Triggers a Backbone.Event so others can subscribe to this event and add/replace data
before is send it to Google Tag Manager
Tracker.trigger(eventName, eventData, item);
this.pushData(eventData);

return this;
}
});
}
}
});
```

- ## 6. Save the file.

Step 2: Prepare the Developer Tools for Your Customizations:

1. Open the GoogleTagManager.Extension @1.0.0 module.

Create a file in this module and name it **hs.package.json**.

- ## Modules/extensions/googleTagManager.ExternalModule

```
  {
    "gulp": {
        "javascript": [
            "JavaScript/* .js"
        ]
    }
}
```

- #### 4. Save the ns.package.json file.

- ## 5. Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

- Add your custom module to the `modules` object to ensure that the Gulp tasks include your extension when you deploy.

Your code should look similar to the following example:

```
{  
  "name": "SuiteCommerce Advanced Elbrus",  
  "version": "2.0",  
  "buildToolsVersion": "1.3.0",  
  "folders": {
```

```

    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/GoogleTagManager.Extension": "1.0.0",
    "extensions/MyExampleCartExtension1": "1.0.0",
    //...
  }
}

```

7. Add `GoogleTagManager.Extension` as a dependency to the following entry points of the `JavaScript` object:

- `SC.Shopping.Starter`
- `SC.MyAccount.Starter`
- `SC.Checkout.Starter`

Your code should look similar to the following example:

```

//...
"javascript": [
  {
    "entryPoint": "SC.Shopping.Starter",
    "exportFile": "shopping.js",
    "dependencies": [
      "Backbone.View.Plugins",
      //...
      "GoogleTagManager.Extension"
    ],
    //...
  },
  {
    "entryPoint": "SC.MyAccount.Starter",
    "exportFile": "myaccount.js",
    "dependencies": [
      "Backbone.View.Plugins",
      //...
      "GoogleTagManager.Extension"
    ],
    //...
  },
  {
    "entryPoint": "SC.Checkout.Starter",
    "exportFile": "checkout.js",
    "dependencies": [
      "Backbone.View.Plugins",
      //...
      "GoogleTagManager.Extension"
    ],
    //...
  },
  //...
]

```

8. Save the `distro.json` file.

Step 3: Test and Deploy Your Extension:

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, all selected matrix item options should appear in the Product Details Page.

Delivery Methods Not Appearing in One Page Checkout

 **Applies to:** SuiteCommerce Advanced | Mont Blanc | Vinson | Elbrus

In some cases, not all delivery methods appear when using the One Page Checkout flow. This issue sometimes occurs after a user enters a zip code when checking out as a guest.

To correct this error, extend the `mountToApp()` method located in the `CheckoutSkipLogin.js` file and hide the following line within comment tags:

```
data.cart && LiveOrderModel.getInstance().set(data.cart);
```

In addition to making these changes, you must create an `ns.package.json` file and update your `distro.json` file for any custom modules you include. You can download the code samples described in this procedure here: [OnePageCheckoutDeliveryMethodsPatch.zip](#).

Step 1: Extend the `CheckoutSkipLogin.js` File

1. If you have not done so already, create a directory to store your custom module. Following best practices, name this directory **extensions** and place it in your `Modules` directory. Depending on your implementation and customizations, this directory might already exist.
2. Open your `extensions` directory and create a custom module to maintain your customizations. Give this directory a unique name that is similar to the module being customized. For example:
`Modules/extensions/CheckoutSkipLogin.Extension@1.0.0/`
3. In your new module, create a subdirectory named `JavaScript`.
4. In your `JavaScript` subdirectory, create a new `JavaScript` file. Give this file a unique name that is similar to the file being modified. For example:
`Modules/extensions/CheckoutSkipLogin.Extension@1.0.0/JavaScript/CheckoutSkipLogin.Extension.js`
5. Open this file and set it up to extend the `mountToApp` method of the `CheckoutSkipLogin.js` file. Your file should match the following code snippet:

```
define(
  'CheckoutSkipLogin.Extension'
, [
    'CheckoutSkipLogin'
, 'Account.RegisterAsGuest.Model'

, 'jQuery'
, 'underscore'
]
```

```

    , function (
      CheckoutSkipLogin
    , AccountRegisterAsGuestModel

    , jQuery
    ,
  )
}

{
  'use strict';

  _.extend(CheckoutSkipLogin.prototype,
  {

    mountToApp: function(application)
    {
      // do nothing if the mode is disabled
      if (!application.getConfig('checkout.skipLogin'))
      {
        return;
      }

      //this function is called only if skip login mode is enabled
      var registerUserAsGuest = function ()
      {
        var promise = jQuery.Deferred()
        , profile_model = ProfileModel.getInstance();
        if (profile_model.get('isGuest') === 'F' && profile_model.get('isLoggedIn') === 'F')
        {
          var checkoutStep = application.getLayout().currentView.currentStep;

          checkoutStep && checkoutStep.disableNavButtons();

          new AccountRegisterAsGuestModel().save().done(function(data)
          {
            var skipLoginDontUpdateProfile = profile_model.get('skipLoginDontUpdateProfile');

            if(skipLoginDontUpdateProfile && data.user)
            {
              _.each(skipLoginDontUpdateProfile, function(attr)
              {
                delete data.user[attr];
              });
            }

            data.user && profile_model.set(data.user);
            application.getLayout().currentView.wizard.options.profile = profile_model;
            //data.cart && LiveOrderModel.getInstance().set(data.cart);
            data.touchpoints && (application.Configuration.siteSettings.touchpoints =
data.touchpoints);
            promise.resolve();
            checkoutStep && checkoutStep.enableNavButtons();
            jQuery('[data-action="skip-login-message"]').remove();
          });
        }
      };
    }
  });
}

```

```

        });
    }
    else
    {
        promise.resolve();
    }
    return promise;
};

// add 'this.application' to models that doesn't have it.
AddressModel.prototype.application = application;
CreditCardModel.prototype.application = application;
ProfileModel.prototype.application = application;

// wrap save() method to LiveOrderModel, AddressModel and CreditCardModel
var wrapper = function(superFn)
{
    var self = this
    ,   super_arguments = Array.prototype.slice.apply(arguments, [1, arguments.length])
    ,   promise = jQuery.Deferred();

    if (!promiseGuest)
    {
        promiseGuest = registerUserAsGuest();
    }

    promiseGuest.done(function ()
    {
        var result = superFn.apply(self, super_arguments);

        if (result)
        {
            result.done(function ()
            {
                promise.resolve.apply(result, arguments);
            }).fail(function()
            {
                promise.reject.apply(result, arguments);
            });
        }
        else
        {
            // Notify future promises that a front end validation took place and no promise is
            returned
            promise.frontEndValidationError = true;
            promise.reject.apply(result, super_arguments);
        }
    });
};

_(promise).extend({error: function(){return this;},success: function(){return this;}});

return promise;
};

// don't wrap on non-secure domains (Site Builder cart is in Checkout :/ )

```

```

        if (window.location.protocol !== 'http:')
    {
        LiveOrderModel.prototype.save = _wrap(LiveOrderModel.prototype.save, wrapper);
    }

    AddressModel.prototype.save = _wrap(AddressModel.prototype.save, wrapper);

    CreditCardModel.prototype.save = _wrap(CreditCardModel.prototype.save, wrapper);

    ProfileModel.prototype.save = _wrap(ProfileModel.prototype.save, wrapper);
}

});

});

```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Customizations

1. Open the CheckoutSkipLogin.Extension@1.0.0 module.
 2. Create a file in this module and name it **ns.package.json**.
- Modules/extensions/CheckoutSkipLogin.Extension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Save the ns.package.json file.
 5. Open the distro.json file.
- This file is located in the top-level directory of your SuiteCommerce Advanced source code.
6. Add your custom module to the **modules** object to ensure that the Gulp tasks include your extension when you deploy.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Mont Blanc",
  "version": "2.0",
  "buildToolsVersion": "1.1.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/CheckoutSkipLogin.Extension": "1.0.0",
    "suitecommerce/Account": "2.1.0",
  }
}
```

```
//...
```

- Add `CheckoutSkipLogin.Extension` as a dependency to the `SC.Checkout.Starter` entry point of the `JavaScript` object.

Your code should look similar to the following example:

```
//...
"javascript": [
    //...
    {
        "entryPoint": "SC.Checkout.Starter",
        "exportFile": "checkout.js",
        "dependencies": [
            "Backbone.View.Plugins",
            //...
            "CheckoutSkipLogin.Extension"
        ],
    },
//...
```

- Save the `distro.json` file.

Step 3: Test and Deploy Your Extension

- Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
- Confirm your results.

Upon successful deployment, entering a zip code as a guest displays all delivery method options.

Mastercard 2-Series BIN Regex Patch

ⓘ Applies to: SuiteCommerce Advanced | Denali | Mont Blanc | Vinson

The Elbrus release of SuiteCommerce Advanced and later support the latest Mastercard 2-Series BIN range by default. To provide this capability for implementations prior to the Elbrus release, you need to perform the updates described in the section corresponding to your implementation:

- Denali – Mastercard Regex Patch
- Mont Blanc – Mastercard Regex Patch
- Vinson – Mastercard Regex Patch

Denali – Mastercard Regex Patch

ⓘ Applies to: SuiteCommerce Advanced | Denali

In your Denali implementation, the credit card regex values reside in the `Utils.js` file, which is located in the `Utilities` module. To update this module, redefine the `paymentMethodIdCreditCart()` method as described in this section. You can download the code samples described in this procedure here: [MastercardBinRegex--DenaliCodeSamples.zip](#).

Step 1: Extend the Utils.js File

1. If you have not done so already, create a directory to store your custom module.

Following best practices, name this directory **extensions** and place it in your Modules directory. Depending on your implementation and customizations, this directory might already exist.

2. Open your extensions directory and create a custom module to maintain your customizations.

Give this directory a unique name that is similar to the module being customized. For example:

Modules/extensions/Utilities.Extension@1.0.0/

3. In your new Utilities.Extension@1.0.0 module, create a subdirectory called JavaScript.

4. In your JavaScript subdirectory, create a new JavaScript file.

Give this file a unique name that is similar to the file being modified. For example:

Modules/extensions/Utilities.Extension@1.0.0/JavaScript/Utils.Extension.js

5. Open this file and set it up to redefine the **paymentIdCreditCart** method of the Utils.js file.



Important: This must be wrapped inside the **mountToApp()** method.

Your file should match the following code snippet:

```
define(
  'Utils.Extension'
, [
  'Utils'
, 'underscore'
]
, function (
  Utils

  '
)
{

  'use strict';

  return {
    mountToApp: function ()
    {
      Utils.prototype.paymentIdCreditCart=

        function paymentIdCreditCart (cc_number)
        {
          // regex for credit card issuer validation
          var cards_reg_ex = {
            'VISA': /^4[0-9]{12}(?:[0-9]{3})?$/,
            'Master Card': /^(5[1-5][0-9]{14}|2(2|2[1-9]|3-9)[0-9])|[3-6][0-9][0-9]|7([0-1][0-9]|20)[0-9]{12})$/ // previous value: /^5[1-5][0-9]{14}$/
            , 'American Express': /^3[47][0-9]{13}$/,
            , 'Discover': /^6(?:011|5[0-9]{2})[0-9]{12}$/,
            , 'Maestro': /^(?:5[0678]\d\d|6304|6390|67\d\d)\d{8,15}$/,
            }
          // get the credit card name
        }
    }
}
)
```

```

        , paymentmethod_name;

        // validate that the number and issuer
        _.each(cards_reg_ex, function (reg_ex, name)
        {
            if (reg_ex.test(cc_number))
            {
                paymentmethod_name = name;
            }
        });

        var paymentmethod = paymentmethod_name && _.findWhere(SC.ENVIRONMENT.siteSettings.paymentmethods,
{name: paymentmethod_name.toString()});

        return paymentmethod && paymentmethod.internalid;
    }
);
});
);

```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Customizations

1. Open the Utilities.Extension@1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.
Modules/extensions/Utilities.Extension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Save the ns.package.json file.
5. Open the distro.json file.
This file is located in the top-level directory of your SuiteCommerce Advanced source code.
6. Add your custom module to the **modules** object to ensure that the Gulp tasks include your extension when you deploy.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Mont Blanc",
  "version": "2.0",
  "buildToolsVersion": "1.1.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
  }
}
```

```

        "deploy": "DeployDistribution"
    },
    "modules": {
        "extensions/Utilities.Extension": "1.0.0",
        "suitecommerce/Account": "2.1.0",
        //...
    }
}

```

7. Add `Utils.Extension` as a dependency to the following entry points of the `JavaScript` object:
 - `SC.Shopping.Starter`
 - `SC.MyAccount.Starter`
 - `SC.Checkout.Starter`

Your code should look similar to the following example:

```

//...
"javascript": [
    {
        "entryPoint": "SC.Shopping.Starter",
        "exportFile": "shopping.js",
        "dependencies": [
            "Backbone.View.Plugins",
            //...
            "Utils.Extension"
        ],
        //...
    },
    {
        "entryPoint": "SC.MyAccount.Starter",
        "exportFile": "myaccount.js",
        "dependencies": [
            "Backbone.View.Plugins",
            //...
            "Utils.Extension"
        ],
        //...
    },
    {
        "entryPoint": "SC.Checkout.Starter",
        "exportFile": "checkout.js",
        "dependencies": [
            "Backbone.View.Plugins",
            //...
            "Utils.Extension"
        ],
        //...
    },
    //...
]
//...

```

8. Save the distro.json file.

Step 3: Test and Deploy Your Extension

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.

2. Confirm your results.

Upon successful deployment, the new Mastercard regex should be included in your compiled shopping.js, myaccount.js, and checkout.js files.

Mont Blanc – Mastercard Regex Patch

 **Applies to:** SuiteCommerce Advanced | Mont Blanc

In your Mont Blanc implementation, the credit card regex values reside in the SC.Configuration.js file, which is located in the SCA module. To update this module, extend the **paymentmethods** array as described in this section. You can download the code samples described in this procedure here: [MastercardBinRegex---MontBlancCodeSample.zip](#).

Step 1: Extend the SC.Configuration.js File

1. If you have not done so already, create a directory to store your custom module. Following best practices, name this directory **extensions** and place it in your Modules directory. Depending on your implementation and customizations, this directory might already exist.
2. Open your extensions directory and create a custom module to maintain your configuration modifications. Give this directory a unique name that is similar to the module being customized. For example:
Modules/extensions/SCA.Extension@1.0.0/
3. In your new SCA.Extension@1.0.0 module, create a subdirectory called JavaScript.
4. In your JavaScript subdirectory, create a new JavaScript file. Give this file a unique name that is similar to the file being modified. For example:
Modules/extensions/SCA.Extension@1.0.0/JavaScript/SC.Configuration.Extension.js
5. Open this file and set it up to extend the **paymentmethods** array of the SC.Configuration.js file.



Important: This must be wrapped inside the `mountToApp()` method.

Your file should match the following code snippet:

```
define(
  'SC.Configuration.Extension'
, [
  'SC.Configuration'
, 'underscore'
]

, function (
  Configuration
{
  -
})
```

```
'use strict';

return {
    mountToApp: function ()
    {
        _.extend(Configuration,
        {
            paymentmethods: [
                {
                    key: '5,5,1555641112' // 'VISA'
                    , regex: /^4[0-9]{12}(?:[0-9]{3})?$/}
                ,
                {
                    key: '4,5,1555641112' // 'Master Card'
                    , regex: /^(5[1-5][0-9]{14}|2(2[2|9]|3-9)[0-9])|[3-6][0-9][0-9]|7([0-1][0-9]|20))[0-9]{12})$/}
                ,
                {
                    key: '6,5,1555641112' // 'American Express'
                    , regex: /^3[47][0-9]{13}$/}
                ,
                {
                    key: '3,5,1555641112' // 'Discover'
                    , regex: /^6(?:011|5[0-9]{2})[0-9]{12}$/}
                ,
                {
                    key: '16,5,1555641112' // 'Maestro'
                    , regex: /^(?:5[0678]\d\d|6304|6390|67\d\d)\d{8,15}$/}
                ,
                {
                    key: '17,3,1555641112' // External
                    , description: 'This company allows both private individuals and businesses to accept payments over the Internet'}
                ]
            });
        }
    });
};
```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Customizations

1. Open the SCA.Extension@1.0.0 module module.
2. Create a file in this module and name it **ns.package.json**.

Modules/extensions/SCA.Extension@1.0.0/ns.package.json

3. Build the ns.package.json file using the following code

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

```
}
```

4. Save the ns.package.json file.

5. Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

6. Add your custom module to the **modules** object to ensure that the Gulp tasks include your extension when you deploy.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Mont Blanc",
  "version": "2.0",
  "buildToolsVersion": "1.1.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/SCA.Extension": "1.0.0",
    "suitecommerce/Account": "2.1.0",
    //...
}
```

7. Add **SC.Configuration.Extension** as a dependency to the following entry points of the **JavaScript** object:

- SC.Shopping.Starter
- SC.MyAccount.Starter
- SC.Checkout.Starter

Your code should look similar to the following example:

```
//...
"javascript": [
  {
    "entryPoint": "SC.Shopping.Starter",
    "exportFile": "shopping.js",
    "dependencies": [
      "Backbone.View.Plugins",
      //...
      "SC.Configuration.Extension"
    ],
    //...
  },
  {
    "entryPoint": "SC.MyAccount.Starter",
    "exportFile": "myaccount.js",
    "dependencies": [
      "Backbone.View.Plugins",
      //...
      "SC.Configuration.Extension"
    ],
    //...
  }
]
```

```

        },
{
  "entryPoint": "SC.Checkout.Starter",
  "exportFile": "checkout.js",
  "dependencies": [
    "Backbone.View.Plugins",
    //...
    "SC.Configuration.Extension"
  ],
  //...
},
//...

```

8. Save the distro.json file.

Step 3: Test and Deploy Your Extension

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.
Upon successful deployment, the new Mastercard regex should be included in your compiled shopping.js, myaccount.js, and checkout.js files.

Vinson – Mastercard Regex Patch

 **Applies to:** SuiteCommerce Advanced | Vinson

In your Vinson implementation, the credit card regex values are configured using the SuiteCommerce Configuration record. The default values for this record reside in the PaymentMethods.json configuration file, which is located in the Configuration module.

This section explains how to update the default Mastercard regex value in the JSON source code. You can download the code samples described in this procedure here: [MastercardBinRegex---VinsonCodeSample.zip](#).

 **Note:** For instructions on configuring this property for a domain, see [Site Configuration](#). For more information on the regex property, see [Payment Methods Subtab](#).

Step 1: Create a JSON modification file

1. If you have not done so already, create a directory to store your custom module. Following best practices, name this directory **extensions** and place it in your Modules directory. Depending on your implementation and customizations, this directory might already exist.
2. Open your extensions directory and create a custom module to maintain your configuration modifications.
Give this directory a unique name that is similar to the module being customized. For example:
Modules/extensions/Configuration.Modification@1.0.0/
3. In your new Configuration.Modification@1.0.0 module, create a subdirectory called Configuration.
4. In your Configuration subdirectory, create a JSON file.
Give this file a unique name that is similar to the file being modified. For example:

Modules/extensions/Configuration.Modification@1.0.0/Configuration/PaymentMethodsModification.json

- Open this file and create a `modifications` array, declaring the target property to be replaced, the action to occur, and new value.

In this case, you declare the Master Card by key and provide the new regex value. Your file should match the following code snippet:

```
{
  "type": "object",

  "modifications" : [
    {
      "target": "$.properties.paymentmethods.default[?(@.key == '4,5,1555641112')].regex",
      "action": "replace",
      "value": "^(5[1-5][0-9]{14}|2(2(2[1-9]|[3-9][0-9])|[3-6][0-9][0-9]|7([0-1][0-9]|20))[0-9]{12})$"
    }
  ]
}
```



Note: For more information on creating a modification, see [Modify JSON Configuration Files](#).

- Save the file.

Step 2: Prepare the Developer Tools for Your Customizations

- Open the Configuration.Modification@1.0.0 module.
- Create a file in this module and name it `ns.package.json`.

Modules/extensions/Configuration.Modification@1.0.0/ns.package.json

- Build the ns.package.json file using the following code

```
{
  "gulp": {
    "configuration": [
      "Configuration/*.json"
    ]
  }
}
```

- Save the ns.package.json file.

- Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

- Add your custom module to the `modules` object to ensure that the Gulp tasks include your modification when you deploy.

Your code should look similar to the following example:

```
{
  "name": "SuiteCommerce Advanced Vinson Release",
  "version": "2.0",
  "buildToolsVersion": "1.2.1",
```

```

"folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
},
"modules": {
    "extensions/Configuration.Modification": "1.0.0",
    "suitecommerce/Account": "2.2.0",
    //...
}

```

- Save the distro.json file.

Step 3: Test and Deploy Your Extension

- In your top-level directory for your Vinson source code, run the following command:

`gulp configuration`

2. Navigate to your LocalDistribution directory.
3. Open the configurationManifest.json file.

Your file should reflect the following changes:

```

[
{
    "type": "object",
    "modifications": [
        {
            "target": "$.properties.paymentmethods.default[?(@.key == '4,5,1555641112')].regex",
            "action": "replace",
            "value": "^(5[1-5][0-9]{14}|2(2(2[1-9]| [3-9][0-9])|[3-6][0-9][0-9]|7([0-1][0-9]|20))[0-9]{12})$"
        }
    ]
},
"default": [
    {
        "key": "5,5,1555641112",
        "regex": "^4[0-9]{12}(?:[0-9]{3})?$$",
        "description": "VISA"
    },
    {
        "key": "4,5,1555641112",
        "regex": "^(5[1-5][0-9]{14}|2(2(2[1-9]| [3-9][0-9])|[3-6][0-9][0-9]|7([0-1][0-9]|20))[0-9]{12})$",
        "description": "Master Card"
    }
]

```

After confirming the results locally, you can deploy your changes to your NetSuite account. See [Deploy to NetSuite](#) for details.



Important: Saved SuiteCommerce Configuration records take precedence over the configurationManifest.json file. Therefore, if you have saved any changes to the SuiteCommerce Configuration record for a domain prior to deploying JSON modifications, your deployed modifications do not appear in the UI or on the website for that domain. The values in your saved record take precedence. To make this change take effect in a previously saved record, configure the record for the applicable domain directly.

Auto-Apply Promotions for Elbrus

i Applies to: SuiteCommerce Advanced | Elbrus

With the Kilimanjaro release of SuiteCommerce Advanced, auto-apply promotions are supported by default. In order to provide this capability to the Elbrus release, you need to update your implementation with the changes described in this section. Changes include modifications to existing JavaScript, Template, and Sass files and the addition of a new custom **PromocodeNotifications** module.

- [Modifications to Existing Promotions Code](#)
- [Custom PromocodeNotifications Module](#)

Once these changes are made in your code, you must also enable auto-apply and set up promotions records to use auto-apply. See the help topic [Promotions](#).

Modifications to Existing Promotions Code



Important: Each section below describes the required changes needed. Since many files are affected, and for simplicity, we have not described the detailed implementation steps here. However, you should implement these changes using the best practices of using extensions and overrides while ensuring that you do not impact previous customizations to your implementation. For more information, see [Best Practices for Customizing SuiteCommerce Advanced](#).

In addition to making the changes described, you must create ns.package.json files and update your distro.json file for any custom modules you include the code updates in.



Note: After creating modifications to existing modules, you also need to create a new PromocodeNotifications custom module. Updates to the distro.json file must include this module as well. See [Custom PromocodeNotifications Module](#).

Modify LiveOrder\SuiteScript\LiveOrder.Model.js

Replace

In the **update** method:

```
var current_order = this.get();
```

With

```
var current_order = this.get();
this.setOldPromocodes();
```

Replace

In the `confirmationCreateResult` method:

```
result.promocodes.push({
  internalid: placed_order.getLineItemValue('promotions', 'couponcode', i)
  , code: placed_order.getLineItemValue('promotions', 'couponcode_display', i)
  , isvalid: placed_order.getLineItemValue('promotions', 'promotionisValid', i) === 'T'
  , discountrate_formatted: '' //placed_order.getLineItemValue('promotions', 'discountrate', i)
});
```

With

```
if(placed_order.getLineItemValue('promotions', 'applicabilitystatus', i) !== 'NOT_APPLIED')
{
  result.promocodes.push({
    internalid: placed_order.getLineItemValue('promotions', 'couponcode', i)
    , code: placed_order.getLineItemValue('promotions', 'couponcode_display', i)
    , isvalid: placed_order.getLineItemValue('promotions', 'promotionisValid', i) === 'T'
    , discountrate_formatted: '' //placed_order.getLineItemValue('promotions', 'discountrate', i)
  });
}
```

Replace

in the `addLines` method:

```
var items = []
, self = this;
```

With

```
var items = []
, self = this;

this.setOldPromocodes();
```

Replace

In the `removeLine` method:

```
// Removes the line
ModelsInit.order.removeItem(line_id);
```

With

```
this.setOldPromocodes();
// Removes the line
ModelsInit.order.removeItem(line_id);
```

Replace

In the `getPromoCodes` method:

```
, getPromoCodes: function getPromoCodes (order_fields)
```

```

{
  var result = [];

  if (order_fields.promocodes && order_fields.promocodes.length)
  {
    _.each(order_fields.promocodes, function (promo_code)
    {
      // @class LiveOrder.Model.PromoCode
      result.push({
        // @property {String} internalid
        internalid: promo_code.internalid
        // @property {String} code
        , code: promo_code.promocode
        // @property {Boolean} isvalid
        , isvalid: promo_code.isvalid === 'T'
        // @property {String} discountrate_formatted
        , discountrate_formatted: ''
        , errmsg : promo_code.errormsg
        , name: promo_code.discount_name
        , rate: promo_code.discount_rate
        , type: promo_code.discount_type
      });
    });
  }

  return result;
}

```

With

```

, getPromoCodes: function getPromoCodes (order_fields)
{
  var result = []
  , self = this;

  if (order_fields.promocodes && order_fields.promocodes.length)
  {
    _.each(order_fields.promocodes, function (promo_code)
    {
      // @class LiveOrder.Model.PromoCode
      var promocode = {
        // @property {String} internalid
        internalid: promo_code.internalid
        // @property {String} code
        , code: promo_code.promocode
        // @property {Boolean} isvalid
        , isvalid: promo_code.isvalid === 'T'
        // @property {String} discountrate_formatted
        , discountrate_formatted: ''
        , errmsg : promo_code.errormsg
        , name: promo_code.discount_name
        , rate: promo_code.discount_rate
        , type: promo_code.discount_type
      };
    });
  }

  return result;
}

```

```

    if (!_.isUndefined(promo_code.is_auto_applied))
    {
        // @property {Boolean} isautoapplied
        promocode.isautoapplied = promo_code.is_auto_applied;
        // @property {String} applicabilitystatus
        promocode.applicabilitystatus = (promo_code.applicability_status) ? promo_code.applicability_status : '';
        // @property {String} applicabilityreason
        promocode.applicabilityreason = (promo_code.applicability_reason) ? promo_code.applicability_reason : '';
    }

    if (!_.isUndefined(promo_code.is_auto_applied) && !_.isUndefined(promo_code.applicability_status) && !_.isUndefined(promo_code.applicability_reason) && !_.isUndefined(self.old_promocodes))
    {
        var old_promocode = (self.old_promocodes) ? _.find(self.old_promocodes, function (old_promo_code){ return old_promo_code.internalid === promo_code.internalid; }) : '';

        if (!old_promocode || old_promocode.applicability_status !== promo_code.applicability_status || (!promo_code.is_auto_applied && promo_code.applicability_reason !== old_promocode.applicability_reason))
        {
            promocode.notification = true;
        }
    }

    result.push(promocode);
});

delete this.old_promocodes;
}

return result;
}

```

Replace

In the `getOptionByCartOptionId` method:

```

, getOptionByCartOptionId: function getOptionByCartOptionId (options, cart_option_id)
{
    return _.findWhere(options, {cartOptionId: cart_option_id});
}

```

With

```

, getOptionByCartOptionId: function getOptionByCartOptionId (options, cart_option_id)
{
    return _.findWhere(options, {cartOptionId: cart_option_id});
}

// @method setOldPromocodes sets a local instance of the order's promocodes, used to be able to detect changes
// in a promocode.
, setOldPromocodes: function setOldPromocodes ()
{
    var order_fields = this.getFieldValues();
    this.old_promocodes = order_fields.promocodes;
}

```

Modify Transaction\SuiteScript\Transaction.Model.js

Replace

In the `getRecordPromocodes` method:

```
this.result.promocodes.push({
  // @class Transaction.Model.Get.Promocode
  // @property {String} internalid
  internalid: this.record.getLineItemValue('promotions', 'couponcode', i)
  // @property {String} code
  , code: this.record.getLineItemValue('promotions', 'couponcode_display', i)
  // @property {Boolean} isvalid
  , isvalid: this.record.getLineItemValue('promotions', 'promotionisvalid', i) === 'T'
  // @property {String} discountrate_formatted
  , discountrate_formatted: '/this.record.getLineItemValue("promotions", "discountrate", i)
});
```

With

```
if(this.record.getLineItemValue('promotions', 'applicabilitystatus', i) !== 'NOT_APPLIED'){
  this.result.promocodes.push({
    // @class Transaction.Model.Get.Promocode
    // @property {String} internalid
    internalid: this.record.getLineItemValue('promotions', 'couponcode', i)
    // @property {String} code
    , code: this.record.getLineItemValue('promotions', 'couponcode_display', i)
    // @property {Boolean} isvalid
    , isvalid: this.record.getLineItemValue('promotions', 'promotionisvalid', i) === 'T'
    // @property {String} discountrate_formatted
    , discountrate_formatted: '/this.record.getLineItemValue("promotions", "discountrate", i)
  });
}
```

Modify Cart\Templates\cart_detailed.tpl

Replace

```
<div data-confirm-message class="cart-detailed-confirm-message">
</div>
```

With

```
<div data-confirm-message class="cart-detailed-confirm-message"></div>
<div data-view="Promocode.Notifications"></div>
```

Add the file Cart\Templates\cart_promocode_notifications.tpl

```
<div data-view="Promocode.Notification">
</div>
```

Add the file Cart/Sass/_cart-promocode-notifications.scss

```
//empty file
```

Modify Cart\JavaScript\Cart.Promocode.List.Item.View.js

Replace

```
//@module Cart
define('Cart.Promocode.List.Item.View'
, [
  'cart_promocode_list_item.tpl'

, 'Backbone'
]
, function (
  cart_promocode_list_item_tpl

, Backbone
)
```

With

```
//@module Cart
define('Cart.Promocode.List.Item.View'
, [
  'cart_promocode_list_item.tpl'

, 'Backbone'
, 'underscore'
]
, function (
  cart_promocode_list_item_tpl

, Backbone
'
)
```

Replace

In the `getContext` method:

```
var code = this.model.get('code')
, internalid = this.model.get('internalid');
```

With

```
var code = this.model.get('code')
, internalid = this.model.get('internalid')
```

```
, hide_autoapply_promo = (!_.isUndefined(this.model.get('isautoapplied'))) ? this.model.get('applicabilityreason') === 'DISCARDED_BEST_OFFER' || (this.model.get('isautoapplied') && this.model.get('applicabilitystatus') === 'NOT_APPLIED') : false;
```

Replace

In the `getContext` method:

```
//@property {Boolean} showPromo
showPromo: !!code
```

With

```
//@property {Boolean} showPromo
showPromo: !!code && !hide_autoapply_promo
```

Replace

In the `getContext` method:

```
//@property {Boolean} isEditable
, isEditable: !this.options.isReadOnly
```

With

```
//@property {Boolean} isEditable
, isEditable: !this.options.isReadOnly && !this.model.get('isautoapplied')
```

Add the file Cart\JavaScript\Cart.Promocode.Notifications.View.js

```
//@module Cart
define('Cart.Promocode.Notifications.View'
, [
  'GlobalViews.Message.View'
, 'cart_promocode_notifications.tpl'

, 'Backbone'
, 'Backbone.CompositeView'
, 'underscore'
]
, function (
  GlobalViewsMessageView
, cart_promocode_notifications

, Backbone
, BackboneCompositeView
'-
)
{
  'use strict';

//@class Cart.Promocode.Notification.View @extend Backbone.View
```

```

return Backbone.View.extend({

    // @property {Function} template
    template:cart_promocode_notifications

    // @method initialize
    // @return {Void}
    , initialize: function initialize ()
    {
        BackboneCompositeView.add(this);
        this.on('afterCompositeViewRender', this.afterViewRender, this);
    }

    // @property {ChildViews} childViews
    , childViews: {
        'Promocode.Notification': function ()
        {
            var notification = this.getNotification();

            return new GlobalviewsMessageView({
                message: notification.message
                , type: notification.type
                , closable: true
            });
        }
    }

    // @method afterViewRender lets parent model know the promotion already showed its current notification
    // @return {Void}
    , afterViewRender: function()
    {
        this.options.parentModel.trigger('promocodeNotificationShown', this.model.get('internalid'));
    }

    // @method getNotification
    // @return {Notification}
    , getNotification: function ()
    {
        var notification = {};

        if(this.model.get('applicabilitystatus') === 'APPLIED')
        {
            notification.type = 'success';
            notification.message = _('Promotion <strong>').translate() + this.model.get('code') + _('</strong> is now
affecting your order.').translate();
        }
        else if(this.model.get('applicabilityreason') === 'CRITERIA_NOT_MET')
        {
            notification.type = (!this.model.get('isautoapplied')) ? 'warning' : 'info';
            notification.message = _('Promotion <strong>').translate() + this.model.get('code') + _('</strong> is not
affecting your order. ').translate() + this.model.get('errormsg');
        }
        else if(this.model.get('applicabilityreason') === 'DISCARDED_BEST_OFFER')
        {
            notification.type = 'info';
        }
    }
})

```

```

    notification.message = _('We have chosen the best possible offer for you. Promotion <strong>').translate()
+ this.model.get('code') + _('</strong> is not affecting your order.').translate();
}

return notification;
}

//@method getContext
//@return {Cart.Promocode.Notifications.View.context}
, getContext: function getContext ()
{
  //@class Cart.Promocode.Notifications.View.context
  return {};
  //@class Cart.Promocode.Notifications.View
}
);
});
});

```

Modify Cart\JavaScript\Cart.Detailed.View.js

Replace

```
, 'Cart.Lines.View'
```

With

```
, 'Cart.Lines.View'
, 'Cart.Promocode.Notifications.View'
```

Replace

```
, CartLinesView
```

With

```
, CartLinesView
, CartPromocodeNotifications
```

Replace

In the `initialize` method:

```
this.model.on('LINE_ROLLBACK', this.render, this);
```

With

```
this.model.on('LINE_ROLLBACK', this.render, this);
this.model.on('promocodeNotificationShown', this.removePromocodeNotification, this);
```

Replace

In the `storeCollapsiblesState` method:

```
// @method storeColapsiblesState
// @return {Void}
, storeColapsiblesState: function ()
{
  this$('.collapse').each(function (index, element)
  {
    colapsibles_states[Utils.getFullPathForElement(element)] = jQuery(element).hasClass('in');
  });
}
```

With

```
// @method storeColapsiblesState
// @return {Void}
, storeColapsiblesState: function ()
{
  this$('.collapse').each(function (index, element)
  {
    colapsibles_states[Utils.getFullPathForElement(element)] = jQuery(element).hasClass('in');
  });
}

// @method removePromocodeNotification
// @param String promocode_id
// @return {Void}
, removePromocodeNotification: function(promocode_id)
{
  var promocode = _.findWhere(this.model.get('promocodes'), {internalid: promocode_id});

  delete promocode.notification;
}
```

Replace

In the `childViews` object:

```
, 'Item.ListNavigable': function ()
{
  return new BackboneCollectionView({
    collection: this.model.get('lines')
    , viewsPerRow: 1
    , childView: CartLinesView
    , childViewOptions: {
      navigable: true
      , application: this.application
      , SummaryView: CartItemSummaryView
      , ActionsView: CartItemActionsView
      , showAlert: false
    }
  });
}
```

With

```
, 'Item.ListNavigable': function ()
```

```

{
  return new BackboneCollectionView({
    collection: this.model.get('lines')
    , viewsPerRow: 1
    , childView: CartLinesView
    , childViewOptions: {
      navigable: true
      , application: this.application
      , SummaryView: CartItemSummaryView
      , ActionsView: CartItemActionsView
      , showAlert: false
    }
  });
}
, 'Promocode.Notifications': function ()
{
  var promotions = .filter(this.model.get('promocodes') || [], function (promocode) { return promocode.notification === true; });

  if(promotions.length){
    return new BackboneCollectionView({
      collection: promotions
      , viewsPerRow: 1
      , childView: CartPromocodeNotifications
      , childViewOptions: {
        parentModel: this.model
      }
    });
  }
}

```

Modify CheckoutApplication\JavaScript \SC.Checkout.Configuration.Steps.BillingFirst.js

Replace

```
, 'OrderWizard.Module.PromocodeForm'
```

With

```
, 'OrderWizard.Module.PromocodeForm'
, 'OrderWizard.Module.PromocodeNotifications'
```

Replace

```
, OrderWizardModulePromocodeForm
```

With

```
, OrderWizardModulePromocodeForm
```

```
, OrderWizardModulePromocodeNotification
```

Replace

In the Billing Address step:

```
[OrderWizardModuleMultiShipToEnableLink, {exclude_on_skip_step: true}]
```

With

```
[OrderWizardModulePromocodeNotification, {exclude_on_skip_step: true}]
, [OrderWizardModuleMultiShipToEnableLink, {exclude_on_skip_step: true}]
```

Replace

In the Shipping Address step:

```
, isActive: function ()
{
  return !this.wizard.isMultiShipTo();
}
, modules: [
  [OrderWizardModuleMultiShipToEnableLink, {exclude_on_skip_step: true}]
```

With

```
, isActive: function ()
{
  return !this.wizard.isMultiShipTo();
}
, modules: [
  [OrderWizardModulePromocodeNotification, {exclude_on_skip_step: true}]
, [OrderWizardModuleMultiShipToEnableLink, {exclude_on_skip_step: true}]]
```

Replace

In the Shipping method step:

```
[OrderWizardModuleAddressShipping, {edit_url: '/shipping/address'}]
```

With

```
[OrderWizardModulePromocodeNotification, {exclude_on_skip_step: true}]
, [OrderWizardModuleAddressShipping, {edit_url: '/shipping/address'}]]
```

Replace

In the Payment step:

```
OrderWizardModulePaymentMethodGiftCertificates
```

With

```
[OrderWizardModulePromocodeNotification, {exclude_on_skip_step: true}]
, OrderWizardModulePaymentMethodGiftCertificates
```

Replace

In the Review step:

```
, [OrderWizardModuleSubmitButton, {className: 'order-wizard-submitbutton-module-top'}]
```

With

```
, [OrderWizardModuleSubmitButton, {className: 'order-wizard-submitbutton-module-top'}]
, [OrderWizardModulePromocodeNotification, {exclude_on_skip_step: true}]
```

Modify CheckoutApplication\JavaScript \SC.Checkout.Configuration.Steps.OPC.js

Replace

```
, 'OrderWizard.Module.PromocodeForm'
```

With

```
, 'OrderWizard.Module.PromocodeForm'
, 'OrderWizard.Module.PromocodeNotifications'
```

Replace

```
, OrderWizardModulePromocodeForm
```

With

```
, OrderWizardModulePromocodeForm
, OrderWizardModulePromocodeNotification
```

Replace

In the Checkout Information step:

```
[OrderWizardModuleTitle, {title: _('Shipping Address').translate(), exclude_on_skip_step: true,
isActive: function() {return this.wizard.model.shippingAddressIsRequired();}}]
```

With

```
[OrderWizardModulePromocodeNotification, {exclude_on_skip_step: true}]
, [OrderWizardModuleTitle, {title: _('Shipping Address').translate(), exclude_on_skip_step: true,
isActive: function() {return this.wizard.model.shippingAddressIsRequired();}}]
```

Replace

In the Review step:

```
, [ //Mobile Top
OrderWizardModuleSubmitButton
,
{
  className: 'order-wizard-submitbutton-module-top'
}
]
```

With

```
, [ //Mobile Top
OrderWizardModuleSubmitButton
,
{
  className: 'order-wizard-submitbutton-module-top'
}
]
, [OrderWizardModulePromocodeNotification, {exclude_on_skip_step: true}]
```

Modify CheckoutApplication\JavaScript \SC.Checkout.Configuration.Steps.Standard.js

Replace

```
, 'OrderWizard.Module.PromocodeForm'
```

With

```
, 'OrderWizard.Module.PromocodeForm'
, 'OrderWizard.Module.PromocodeNotifications'
```

Replace

```
, OrderWizardModulePromocodeForm
```

With

```
, OrderWizardModulePromocodeForm
, OrderWizardModulePromocodeNotification
```

Replace

In the Shipping Address step:

```
OrderWizardModuleMultiShipToEnableLink
```

With

```
[OrderWizardModulePromocodeNotification, {exclude_on_skip_step: true}]
, OrderWizardModuleMultiShipToEnableLink
```

Replace

In the Payment step:

```
OrderWizardModulePaymentMethodGiftCertificates
```

With

```
[OrderWizardModulePromocodeNotification, {exclude_on_skip_step: true}]
, OrderWizardModulePaymentMethodGiftCertificates
```

Replace

In the Review step:

```
, [ //Mobile Top
  OrderWizardModuleSubmitButton
  ,
  {
    className: 'order-wizard-submitbutton-module-top'
  }
]
```

With

```
, [ //Mobile Top
  OrderWizardModuleSubmitButton
  ,
  {
    className: 'order-wizard-submitbutton-module-top'
  }
]
, [OrderWizardModulePromocodeNotification, {exclude_on_skip_step: true}]
```

Custom PromocodeNotifications Module

In addition to modifying existing SuiteCommerce Advanced modules, you need to add a custom module for Promotion Notifications. This custom module provides the template, Sass, and JavaScript files required to extend the Order Wizard module with promotions notifications.

Step 1: Create the custom PromocodeNotifications module:

1. Create a directory to store your custom module.

Give this directory a name similar to the module being customized. For example:

`Modules/extensions/OrderWizard.Module.PromocodeNotifications@1.0.0 module`

2. In your new OrderWizard.Module.PromocodeNotifications@1.0.0 module , create the following subdirectories and files.
 - `OrderWizard.Module.PromocodeNotifications@1.0.0\Templates\order_wizard_promocodenotifications.tpl`

```
<div data-type="Promocode.Notifications">
</div>
```

- `OrderWizard.Module.PromocodeNotifications@1.0.0/Sass/_order-wizard-promocodenotifications.scss`

```
//empty file
```

- `OrderWizard.Module.PromocodeNotifications@1.0.0/JavaScript/OrderWizard.Module.PromocodeNotifications.js`

```
// @module OrderWizard.Module.PromocodeNotifications
define(
  'OrderWizard.Module.PromocodeNotifications',
  [
    'Wizard.Module',
    'SC.Configuration',
    'Backbone.CollectionView',
    'Cart.Promocode.Notifications.View'

    , 'order_wizard_promocodenotifications.tpl'
    , 'underscore'
    , 'jQuery'
  ]
  , function (
    WizardModule
    , Configuration
    , BackboneCollectionView
    , CartPromocodeNotificationsView

    , order_wizard_promocodenotifications_tpl
    ,
    ,
    , jQuery
  )
{
  'use strict';

  // @class OrderWizard.Module.PromocodeNotifications @extends Wizard.Module
  return WizardModule.extend({
```

```

//@property {Function} template
template: order_wizard_promocodenotifications_tpl

//@method initialize
, initialize: function initialize ()
{
    WizardModule.prototype.initialize.apply(this, arguments);

    this.wizard.model.on('change:promocodes', this.render, this);

    this.wizard.model.on('promocodeNotificationShown', this.removePromocodeNotification, this);

    this._render();
}

, render: function()
{
    var promocodes = _.filter(this.wizard.model.get('promocodes') || [], function (promocode) { return
    promocode.notification === true; });

    if(promocodes.length){

        var message_collection_view = new BackboneCollectionView({
            collection: promocodes
            , viewsPerRow: 1
            , childView: CartPromocodeNotificationsView
            , childViewOptions: {
                parentModel: this.wizard.model
            }
        });

        message_collection_view.render();

        jQuery('[data-type="Promocode.Notifications"]').html(message_collection_view.$el.html());
    }
}

// @method removePromocodeNotification
// @param String promocode_id
// @return {Void}
, removePromocodeNotification: function (promocode_id)
{
    var promocode = _.findWhere(this.wizard.model.get('promocodes'), {internalid: promocode_id});

    delete promocode.notification;
}

//@method getContext
//@returns {OrderWizard.Module.PromocodeNotifications.Context}
, getContext: function getContext ()
{
    //@class OrderWizard.Module.PromocodeNotifications.Context
    return {};
    //@class OrderWizard.Module.PromocodeNotifications
}

```

```
  });
});
```

Step 2: Prepare the Developer Tools for Your Extension

1. Create a **ns.package.json** file in the OrderWizard.Module.PromocodeNotifications@1.0.0 module.

`Modules/extensions/OrderWizard.Module.PromocodeNotifications@1.0.0/ns.package.json`

2. Build the ns.package.json file using the following code.

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ],
    "sass": [
      "Sass/**/*.scss"
    ],
    "templates": [
      "Templates/*.tpl"
    ]
  }
}
```

3. Add your custom module to the **modules** object of the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code. Adding a reference to your custom module ensures that the Gulp tasks include your module when you deploy. In this example, the **OrderWizard.Module.PromocodeNotifications** module is added at the beginning of the list of modules. However, you can add the module anywhere in the **modules** object. The order of precedence in this list does not matter.

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/OrderWizard.Module.PromocodeNotifications": "1.0.0",
    "suitecommerce/Account": "2.3.0",
    "suitecommerce/Address": "2.4.0",
    ...
  }
}
```

4. Include the module definition ("OrderWizard.Module.PromocodeNotifications") in the dependencies array of the Checkout application JavaScript object.

Your distro.json file should look similar to the following:

```

"tasksConfig": {
//...
"javascript": [
//...
{
    "entryPoint": "SC.Checkout.Starter",
    "exportFile": "checkout.js",
    "dependencies": [
//...
        "OrderWizard.Module.PromocodeForm",
        "OrderWizard.Module.PromocodeNotifications"
    ],
//...
}
]
}

```



Note: Best practice is to place any new modules at the bottom of the list in the `dependencies` array.

- Save the distro.json file.



Note: Any changes to the distro file must also include references to any custom modules you may have created to extend or override files described in the section [Modifications to Existing Promotions Code](#).

Change Email Address Patch

Applies to: SuiteCommerce Advanced | Denali | Mont Blanc | Vinson | Elbrus

With the Kilimanjaro release of SuiteCommerce Advanced, the ability for users to change their email address is supported by default. In order to provide this capability to releases prior to the Kilimanjaro release, you need to update your implementation with the changes described in the section corresponding to your implementation:

- Denali — Change Email Address Patch
- Mont Blanc — Change Email Address Patch
- Vinson — Change Email Address Patch
- Elbrus — Change Email Address Patch



Important: Each section describes the required changes needed to achieve the change email address functionality. Since many files are affected, and for simplicity, we have not described the detailed implementation steps here. However, you should implement these changes using the best practices of using extensions and overrides while ensuring that you do not impact previous customizations to your implementation. For more information, see [Best Practices for Customizing SuiteCommerce Advanced](#).

In addition to making the changes described, you must create ns.package.json files and update your distro.json file for any custom modules you include the code updates in. Since this customization modifies SSP library files, changes are not immediately visible in your local environment. You must first deploy your custom module directly to NetSuite.

Denali — Change Email Address Patch

ⓘ Applies to: SuiteCommerce Advanced | Denali

To update a Denali implementation to give users the ability to change their email address, you'll need to modify and add the files as detailed below.



Important: This addition requires changes to templates, views, JavaScript, and SSP application files. Test changes thoroughly against your existing customizations before deploying to your published domain.

Modify CheckoutApplication/SuiteScript/checkout.ssp

Replace

```
var SiteSettings
, parameters
, siteType
, Environment
, Language
, Currency
, Error
, cart_bootstrap
, login
, Application;
```

With

```
var SiteSettings
, parameters
, siteType
, Environment
, Language
, Currency
, Error
, cart_bootstrap
, login
, Application
, password_reset_expired;
```

Replace

```
if (session.isChangePasswordRequest())
{
    parameters.fragment = 'reset-password';
    login = true;
}
```

With

```
if (parameters.passwdret)
{
    try
```

```

{
  if (session.isChangePasswordRequest())
  {
    parameters.fragment = 'reset-password';
    login = true;
  }
}
catch (e)
{
  password_reset_expired = true;
}

}

```

Replace

```
<% if (Error) { %>
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>
```

With

```

<% if (Error) { %>
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>
<% if (parameters.key) { %>
<script>SC.ENVIRONMENT.email_verification_error = true;</script>
<% } else if (password_reset_expired) { %>
<script>SC.ENVIRONMENT.password_reset_expired_error = true;</script>
<% } else if (parameters.passwdret && parameters.fragment !== 'reset-password') { %>
<script>SC.ENVIRONMENT.password_reset_invalid_error = true;</script>
<% } %>

```

Modify LoginRegister/Javascript/LoginRegister.View.js

Replace

```

define('LoginRegister.View'
, [ 'login_register.tpl'

, 'Profile.Model'
, 'LoginRegister.Login.View'
, 'LoginRegister.Register.View'
, 'LoginRegister.CheckoutAsGuest.View'
, 'Backbone.CompositeView'
, 'SC.Configuration'

, 'Backbone'
, 'underscore'
, 'Utils'
]
, function (
  login_register_tpl

```

```

, ProfileModel
, LoginView
, RegisterView
, CheckoutAsGuestView
, BackboneCompositeView
, Configuration

, Backbone
, -
)

```

With

```

define('LoginRegister.View'
, [ 'login_register.tpl'

, 'GlobalViews.Message.View'
, 'Profile.Model'
, 'LoginRegister.Login.View'
, 'LoginRegister.Register.View'
, 'LoginRegister.CheckoutAsGuest.View'
, 'Backbone.CompositeView'
, 'SC.Configuration'

, 'Backbone'
, 'underscore'
, 'Utils'
]
, function (
  login_register_tpl

, GlobalViewsMessageView
, ProfileModel
, LoginView
, RegisterView
, CheckoutAsGuestView
, BackboneCompositeView
, Configuration

, Backbone
, -
)

```

Replace

```

this.enableCheckoutAsGuest = is_checking_out && profile_model.get('isLoggedIn') === 'F' &&
(Configuration.getRegistrationType() === 'optional' || Configuration.getRegistrationType() === 'disabled');

BackboneCompositeView.add(this);

```

With

```

this.enableCheckoutAsGuest = is_checking_out && profile_model.get('isLoggedIn') === 'F' &&
(Configuration.getRegistrationType() === 'optional' || Configuration.getRegistrationType() === 'disabled');

```

```

if (SC.ENVIRONMENT.email_verification_error)
{
    this.message = _('The validation process has failed. Please login into your account and click on the validation link again.').translate();
    delete SC.ENVIRONMENT.email_verification_error;
}
else if (SC.ENVIRONMENT.password_reset_invalid_error)
{
    this.message =_('Your reset password link is invalid. Request a new one using the Forgot Password link.').translate();
    delete SC.ENVIRONMENT.password_reset_invalid_error;
}
else if (SC.ENVIRONMENT.password_reset_expired_error)
{
    this.message =_('Your reset password link has expired. Request a new one using the Forgot Password link.').translate();
    delete SC.ENVIRONMENT.password_reset_expired_error;
}

BackboneCompositeView.add(this);

```

Replace

```

, 'Register': function ()
{
    return new RegisterView(this.child_view_options);
}

```

With

```

, 'Register': function ()
{
    return new RegisterView(this.child_view_options);
}
, 'Messages': function ()
{
    if (this.message)
    {
        return new GlobalViewsMessageView({
            message: this.message
            , type: 'error'
            , closable: true
        });
    }
}

```

Modify LoginRegister/Templates/login_register.tpl

Replace

```
<header class="login-register-header">
```

```

{{#if showRegister}}
<h1 class="login-register-title"> {{translate 'Log in | Register'}}</h1>
{{else}}
<h1 class="login-register-title"> {{translate 'Log in'}}</h1>
{{/if}}
</header>

```

With

```

<header class="login-register-header">
{{#if showRegister}}
<h1 class="login-register-title"> {{translate 'Log in | Register'}}</h1>
{{else}}
<h1 class="login-register-title"> {{translate 'Log in'}}</h1>
{{/if}}
</header>

```

Modify MyAccountApplication/SuiteScript/my_account.ssp

Replace

```

<%
var SiteSettings
, siteType
, Environment
, Language
, Currency
, Error
, Application;

try
{
    Application = require('Application');
    SiteSettings = require('SiteSettings.Model').get();
    siteType = SiteSettings.sitetype;

    Environment = Application.getEnvironment(session, request);

    Language = Environment.currentLanguage && Environment.currentLanguage.locale || '';
    Currency = Environment.currencyCodeSpecifiedOnUrl;

    // Access control, if you are not loged this will send you to the log in page
    if (!session.isLoggedIn() || session.getCustomer().isGuest())
    {
        var parameters = request.getAllParameters();

        delete parameters.sitepath;
        parameters.origin = 'customercenter';

        if (parameters.fragment)
        {
            parameters.origin_hash = parameters.fragment;
            delete parameters.fragment;
        }
    }
}

```

```

        }

        return nlapiSetRedirectURL('EXTERNAL', SiteSettings.touchpoints.login, null, false, parameters);
    }
} catch (e) {
    Error = Application.processError(e);
}

%>

```

With

```

<%
var SiteSettings
, siteType
, Environment
, Language
, Currency
, Error
, Application
, parameters
, email_change_verification
;

try
{
    Application = require('Application');
    SiteSettings = require('SiteSettings.Model').get();
    parameters = request.getAllParameters();
    siteType = SiteSettings.sitetype;

    Environment = Application.getEnvironment(session, request);

    Language = Environment.currentLanguage && Environment.currentLanguage.locale || '';
    Currency = Environment.currencyCodeSpecifiedOnUrl;

    // Access control, if you are not loged this will send you to the log in page
    if (!session.isLoggedIn2() || session.getCustomer().isGuest())
    {
        delete parameters.sitepath;
        parameters.origin = 'customercenter';

        if (parameters.fragment)
        {
            parameters.origin_hash = parameters.fragment;
            delete parameters.fragment;
        }

        return nlapiSetRedirectURL('EXTERNAL', SiteSettings.touchpoints.login, null, false, parameters);
    }
    else if (session.isLoggedIn2() && parameters.key)
    {
        try
        {

```

```

        session.verifyEmailChange(parameters.key)
        email_change_verification = true;
    }
    catch (e)
    {
        email_change_verification = e.details;
    }
}
} catch (e) {
    Error = Application.processError(e);
}
%>

```

Replace

```

<% if (Error) { %>
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>

```

With

```

<% if (Error) { %>
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>
<% if (email_change_verification) { %>
<script>SC.SESSION.email_change_verification = '<%= email_change_verification %>';</script>
<% } %>

```

Modify Overview/Javascript/Overview.Home.View.js

Replace

```

define('Overview.Home.View'
, [
    'SC.Configuration'
, 'Overview.Banner.View'
, 'Overview.Profile.View'
, 'Overview.Payment.View'
, 'Overview.Shipping.View'
, 'Backbone.CollectionView'
, 'OrderHistory.List.Tracking.Number.View'
, 'RecordViews.View'
, 'Handlebars'

, 'overview_home.tpl'

, 'Backbone'
, 'Backbone.CompositeView'
, 'underscore'
, 'Utils'
]
, function(
    Configuration

```

```

    , OverviewBannerView
    , OverviewProfileView
    , OverviewPaymentView
    , OverviewShippingView
    , BackboneCollectionView
    , OrderHistoryListTrackingNumberView
    , RecordViewsView
    , Handlebars

    , overview_home_tpl

    , Backbone
    , BackboneCompositeView
    '-
)

```

With

```

define('Overview.Home.View'
, [
    'SC.Configuration'
, 'GlobalViews.Message.View'
, 'Overview.Banner.View'
, 'Overview.Profile.View'
, 'Overview.Payment.View'
, 'Overview.Shipping.View'
, 'Backbone.CollectionView'
, 'OrderHistory.List.Tracking.Number.View'
, 'RecordViews.View'
, 'Handlebars'

, 'overview_home_tpl'

, 'Backbone'
, 'Backbone.CompositeView'
, 'underscore'
, 'Utils'
]
, function(
    Configuration
, GlobalViewsMessageView
, OverviewBannerView
, OverviewProfileView
, OverviewPaymentView
, OverviewShippingView
, BackboneCollectionView
, OrderHistoryListTrackingNumberView
, RecordViewsView
, Handlebars

, overview_home_tpl

, Backbone
, BackboneCompositeView
'-
)

```

```
)
```

Replace

```
this.creditcards.on('reset destroy change add', this.showContent, this);
}
```

With

```
this.creditcards.on('reset destroy change add', this.showContent, this);

if (SC.SESSION.email_change_verification)
{
    this.email_change_verification = SC.SESSION.email_change_verification;
    delete SC.SESSION.email_change_verification;
}
}
```

Replace

```
, 'Overview.Shipping': function()
{
    return new OverviewShippingView({ model: this.defaultShippingAddress });
}
```

With

```
, 'Overview.Shipping': function()
{
    return new OverviewShippingView({ model: this.defaultShippingAddress });
}

, 'Overview.Messages': function ()
{
    if (this.email_change_verification)
    {
        return new GlobalViewsMessageView({
            message: this.email_change_verification === 'true' ? _('Your email has been changed successfully to
<strong>').translate() + this.model.get('email') + '</strong>' : this.email_change_verification
            , type: this.email_change_verification === 'true' ? 'success' : 'error'
            , closable: true
        });
    }
}
```

Modify Overview/Templates/overview_home.tpl

Replace

```
<section class="overview-home">
    <div class="overview-home-orders" data-permissions="{{purchasesPermissions}}>
```

With

```
<section class="overview-home">
  <div data-view="Overview.Messages"></div>
  <div class="overview-home-orders" data-permissions="{{purchasesPermissions}}>
```

Add Profile/Javascript/Profile.ChangeEmailAddress.Model.js

```
// Profile.ChangeEmailAddress.Model.js
// -----
// View Model for changing user's email
// @module Profile
define(
  'Profile.ChangeEmailAddress.Model',
  [
    'Backbone',
    'underscore',
    'Utils'
  ],
  function (
    Backbone
  ) {
    'use strict';

    // @class Profile.ChangeEmailAddress.Model @extends Backbone.Model
    return Backbone.Model.extend(
      {
        urlRoot: 'services/Profile.Service.ss',
        validation: {
          current_password: { required: true, msg: _('Current password is required').translate() },
          confirm_email: [
            { required: true, msg: _('Confirm Email is required').translate() },
            { equalTo: 'new_email', msg: _('New Email and Confirm New Email do not match').translate() }
          ],
          new_email: { required: true, msg: _('New Email is required').translate() }
        }
      });
  });
});
```

Add Profile/Javascript/Profile.ChangeEmailAddress.View.js

```
// @module Profile
define(
  'Profile.ChangeEmailAddress.View',
  [
    'GlobalViews.Message.View',
    'Backbone.FormView',
    'SC.Configuration'

    , 'profile_change_email.tpl'

    , 'Backbone'
```

```

, 'underscore'
, 'Utils'
]
, function (
  GlobalViewsMessageView
, BackboneFormView
, Configuration

, profile_change_email_tpl

, Backbone
'-
)
{
  'use strict';

// @class Profile.ChangeEmailAddress.View @extends Backbone.View
return Backbone.View.extend({


  template: profile_change_email_tpl

, page_header: _('Change Email').translate()

, title: _('Change Email').translate()

, events: {
    'submit form': 'saveFormCustom'
  }

, bindings: {
    '[name="current_password)": "current_password"
    , '[name="new_email"]": "new_email"
    , '[name="confirm_email)": "confirm_email"
  }

, initialize: function()
{
  Backbone.View.prototype.initialize.apply(this, arguments);
  BackboneFormView.add(this);
}

, saveFormCustom: function ()
{
  this.new_email = this.$('[name="new_email"]').val();
  BackboneFormView.saveForm.apply(this, arguments);
}

, showSuccess: function (placeholder)
{
  var global_view_message = new GlobalViewsMessageView({
    message: _('A confirmation email has been sent to <strong>').translate() + this.new_email + '</strong>'
    , type: 'success'
    , closable: true
  });
}
}

```

```

        placeholder.html(global_view_message.render().$el.html());
    }
});
});
});
```

Modify Profile/Javascript/Profile.Information.View.js

Replace

```

define(
  'Profile.Information.View'
, [
  'SC.Configuration'
, 'GlobalViews.Message.View'
, 'Backbone.FormView'

, 'profile_information.tpl'

, 'Backbone'
, 'underscore'
, 'jQuery'
, 'Utils'
]
, function (
  Configuration
, GlobalViewsMessageView
, BackboneFormView

, profile_information_tpl

, Backbone
'-
, jQuery
)
{
  'use strict';

// @class Profile.Information.View @extends Backbone.View
return Backbone.View.extend({


  template: profile_information_tpl
, page_header: _('Profile Information').translate()
, title: _('Profile Information').translate()
, attributes: {'class': 'ProfileInformationView'}
, events: {
    'submit form': 'saveForm'
, 'change input[data-type="phone"]': 'formatPhone'
  }

, bindings: {
    '[name="firstname)": "firstname"
, '[name="lastname"]": "lastname"
, '[name="companyname"]": "companyname"
, '[name="phone"]": "phone"
  }
}
```

```

, initialize: function()
{
  BackboneFormView.add(this);
}

, formatPhone: function (e)
{
  var $target = jQuery(e.target);
  $target.val(_.($target.val()).formatPhone());
}

, showSuccess: function ()

```

With

```

define(
  'Profile.Information.View'
, [
  'SC.Configuration'
, 'GlobalViews.Message.View'
, 'Backbone.FormView'

, 'profile_information.tpl'

, 'Backbone'
, 'underscore'
, 'jQuery'
, 'Utils'
]
, function (
  Configuration
, GlobalViewsMessageView
, BackboneFormView

, profile_information_tpl

, Backbone
'-
, jQuery
)
{
  'use strict';

// @class Profile.Information.View @extends Backbone.View
return Backbone.View.extend({

  template: profile_information_tpl
, page_header: _('Profile Information').translate()
, title: _('Profile Information').translate()
, attributes: {'class': 'ProfileInformationView'}
, events: {
    'submit form': 'saveForm'
    , 'change input[data-type="phone"]'define(
'Profile.Information.View'

```

```

, [
  'SC.Configuration',
  'GlobalViews.Message.View',
  'Backbone.FormView'

  , 'Profile.ChangeEmailAddress.Model'
  , 'Profile.ChangeEmailAddress.View'

  , 'profile_information.tpl'

  , 'Backbone'
  , 'underscore'
  , 'jQuery'
  , 'Utils'
]
, function (
  Configuration
, GlobalViewsMessageView
, BackboneFormView

, ProfileChangeEmailModel
, ProfileChangeEmailView

, profile_information_tpl

, Backbone
, -
, jQuery
)
{
  'use strict';

// @class Profile.Information.View @extends Backbone.View
return Backbone.View.extend({


  template: profile_information_tpl
, page_header: _('Profile Information').translate()
, title: _('Profile Information').translate()
, attributes: {'class': 'ProfileInformationView'}
, events: {
    'submit form': 'saveForm'
    , 'change input[data-type="phone"]': 'formatPhone'
    , 'click [data-action="change-email"]': 'changeEmail'
  }

  , bindings: {
    '[name="firstname)": "firstname"
    , '[name="lastname"]": "lastname"
    , '[name="companyname"]": "companyname"
    , '[name="phone"]": "phone"
  }

  , initialize: function(options)
  {
    BackboneFormView.add(this);
  }
}

```

```

        this.application = options.application;
    }

    , formatPhone: function (e)
    {
        var $target = jQuery(e.target);
        $target.val(_.($target.val()).formatPhone());
    }

    , changeEmail: function ()
    {
        var model = new ProfileChangeEmailModel(this.model.attributes);

        var view = new ProfileChangeEmailView({
            application: this.application
        , model: model
        });

        var self = this;

        model.on('save', function () {
            view.showSuccess(self.$('[data-type="alert-placeholder"]'));
        });

        view.useLayoutError = true;

        this.application.getLayout().showInModal(view);
    }

    , showSuccess: function ()
': 'formatPhone'
    }

    , bindings: {
        '[name="firstname)": "firstname"
        , '[name="lastname)": "lastname"
        , '[name="companyname)": "companyname"
        , '[name="phone)": "phone"
    }

    , initialize: function()
    {
        BackboneFormView.add(this);
    }

    , formatPhone: function (e)
    {
        var $target = jQuery(e.target);
        $target.val(_.($target.val()).formatPhone());
    }

    , showSuccess: function ()

```

Add Profile/Sass/_profile-change-email.scss

```
.profile-change-email-button-back {
    @extend .button-back;
}

.profile-change-email-button-back-icon {
    @extend .button-back-icon;
}

.profile-change-email-form-label {
    display: inline-block;
}

.profile-change-email-form-group-label-required {
    @extend .input-required;
}

.profile-change-email {
    @extend .address-edit;
}

.profile-change-email-form-title {}
.profile-change-email-form-area {}

.profile-change-email-form {
    margin-top: $sc-base-margin * 3;
}

.profile-change-email-form-group,
.profile-change-email-form-actions {
    @extend .control-group;
}

.profile-change-email-form-label {
    @extend .input-label
}

.profile-change-email-form-group-label-required {}

.profile-change-email-form-group-input {
    @extend .input-large
}

.profile-change-email-form-actions-change {
    @extend .button-primary;
    @extend .button-medium;
}

.profile-change-email-group-form-controls{}

.profile-change-email-form-info-block{}
```

Modify Profile/SuiteScript/Profile.Model.js

Replace

```
define(
  'Profile.Model',
  ['SC.Model', 'Utils'],
  function (SCModel, Utils)
```

With

```
define(
  'Profile.Model',
  ['SC.Model', 'Models.Init', 'Utils'],
  function (SCModel, ModelsInit, Utils)
```

Replace

```
if (data.email && data.email !== this.currentSettings.email && data.email === data.confirm_email)
{
  if(data.isGuest === 'T')
  {
    customerUpdate.email = data.email;
  }
  else
  {
    login.changeEmail(data.current_password, data.email, true);
  }
}
```

With

```
if (data.email && data.email !== this.currentSettings.email && data.email === data.confirm_email && data.isGuest
    === 'T')
{
  customerUpdate.email = data.email;
}
else if (data.new_email && data.new_email === data.confirm_email && data.new_email !
    == this.currentSettings.email)
{
  session.login({
    email: data.email
    , password: data.current_password
  });
  login.changeEmail(data.current_password, data.new_email, true);
}
```

Add Profile/Templates/profile_change_email.tpl

```
<section class="profile-change-email">
```

```

<div data-type="alert-placeholder"></div>
<div class="profile-change-email-form-area">
  <form class="profile-change-email-form">
    <fieldset>
      <small class="profile-change-email-form-label">{{translate 'Required'}} <span class="profile-change-email-form-group-label-required">*</span></small>

      <div class="profile-change-email-form-group" data-input="new_email" data-validation="control-group">
        <label class="profile-change-email-form-group-label" for="new_email">{{translate 'New Email'}} <span class="profile-change-email-form-group-label-required">*</span></label>
        <div class="profile-change-email-group-form-controls" data-validation="control">
          <input type="email" class="profile-change-email-form-group-input" id="new_email" name="new_email" value="" placeholder="{{translate 'your@email.com'}}">
        </div>
      </div>

      <div class="profile-change-email-form-group" data-input="confirm_email" data-validation="control-group">
        <label class="profile-change-email-form-group-label" for="confirm_email">{{translate 'Confirm New Email'}} <span class="profile-change-email-form-group-label-required">*</span></label>
        <div class="profile-change-email-group-form-controls" data-validation="control">
          <input type="email" class="profile-change-email-form-group-input" id="confirm_email" name="confirm_email" value="" placeholder="{{translate 'your@email.com'}}">
        </div>
      </div>

      <div class="profile-change-email-form-group" data-input="current_email" data-validation="control-group">
        <label class="profile-change-email-form-group-label" for="current_password">{{translate 'Password'}} <span class="profile-change-email-form-group-label-required">*</span></label>
        <div class="profile-change-email-group-form-controls" data-validation="control">
          <input type="password" class="profile-change-email-form-group-input" id="current_password" name="current_password" value="">
        </div>
      </div>
    </fieldset>
    <p class="profile-change-email-form-info-block"><small> {{translate 'You will still be able to login with your current email address and password until your new email address is verified.'}} </small></p>
    <div class="profile-change-email-form-actions">
      <button type="submit" class="profile-change-email-form-actions-change">{{translate 'Send Verification Email'}}</button>
    </div>
  </form>
</div>
</section>

```

Modify Profile/Templates/profile_information.tpl

Replace

```

<p class="profile-information-input-email" id="email" name="email">
  {{email}}
</p>

```

With

```
<p class="profile-information-input-email" id="email" name="email">{{email}} | <a class="profile-information-change-email-address" data-action="change-email">{{translate 'Change Address'}}</a></p>
```

Mont Blanc — Change Email Address Patch

ⓘ Applies to: SuiteCommerce Advanced | Mont Blanc

To update a Mont Blanc implementation to give users the ability to change their email address, you'll need to modify and add the files as detailed below.



Important: This addition requires changes to templates, views, JavaScript, and SSP application files. Test changes thoroughly against your existing customizations before deploying to your published domain.

Modify CheckoutApplication/SuiteScript/checkout.ssp

Replace

```
var SiteSettings
, parameters
, siteType
, Environment
, Language
, Currency
, Error
, cart_bootstrap
, confirmation_order_id
, login
, Application;
```

With

```
var SiteSettings
, parameters
, siteType
, Environment
, Language
, Currency
, Error
, cart_bootstrap
, confirmation_order_id
, login
, Application
, password_reset_expired
;
```

Replace

```
if (session.isChangePasswordRequest())
{
    parameters.fragment = 'reset-password';
```

```

    login = true;
}

```

With

```

if (parameters.passwdret)
{
    try
    {
        if (session.isChangePasswordRequest())
        {
            parameters.fragment = 'reset-password';
            login = true;
        }
    }
    catch (e)
    {
        password_reset_expired = true;
    }
}

```

Replace

```

<% if (Error) { %>
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>

```

With

```

<% if (Error) { %>
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>

<% if (parameters.key) { %>
<script>SC.ENVIRONMENT.email_verification_error = true;</script>
<% } else if (password_reset_expired) { %>
<script>SC.ENVIRONMENT.password_reset_expired_error = true;</script>
<% } else if (parameters.passwdret && parameters.fragment !== 'reset-password') { %>
<script>SC.ENVIRONMENT.password_reset_invalid_error = true;</script>
<% } %>

```

Modify LoginRegister/Javascript/LoginRegister.View.js

Replace

```

define('LoginRegister.View'
, [ 'login_register.tpl'

, 'Profile.Model'
, 'LoginRegister.Login.View'
, 'LoginRegister.Register.View'
, 'LoginRegister.CheckoutAsGuest.View'

```

```

        , 'Backbone.CompositeView'
        , 'SC.Configuration'
        , 'Header.Simplified.View'
        , 'Footer.Simplified.View'

        , 'Backbone'
        , 'underscore'
        , 'Utils'
    ]
, function (
    login_register_tpl

    , ProfileModel
    , LoginView
    , RegisterView
    , CheckoutAsGuestView
    , BackboneCompositeView
    , Configuration
    , HeaderSimplifiedView
    , FooterSimplifiedView

    , Backbone
    ,
)

```

With

```

define('LoginRegister.View'
, [ 'login_register.tpl'

        , 'GlobalViews.Message.View'
        , 'Profile.Model'
        , 'LoginRegister.Login.View'
        , 'LoginRegister.Register.View'
        , 'LoginRegister.CheckoutAsGuest.View'
        , 'Backbone.CompositeView'
        , 'SC.Configuration'
        , 'Header.Simplified.View'
        , 'Footer.Simplified.View'

        , 'Backbone'
        , 'underscore'
        , 'Utils'
    ]
, function (
    login_register_tpl

    , GlobalViewsMessageView
    , ProfileModel
    , LoginView
    , RegisterView
    , CheckoutAsGuestView
    , BackboneCompositeView
    , Configuration
    , HeaderSimplifiedView

```

```
, FooterSimplifiedView
, Backbone
'
)
```

Replace

```
this.enableCheckoutAsGuest = is_checking_out && profile_model.get('isLoggedIn') === 'F' &&
(Configuration.getRegistrationType() === 'optional' || Configuration.getRegistrationType() === 'disabled');

BackboneCompositeView.add(this);
```

With

```
this.enableCheckoutAsGuest = is_checking_out && profile_model.get('isLoggedIn') === 'F' &&
(Configuration.getRegistrationType() === 'optional' || Configuration.getRegistrationType() === 'disabled');

if (SC.ENVIRONMENT.email_verification_error)
{
  this.message = _('The validation process has failed. Please login into your account and click on the validation
link again.').translate();
  delete SC.ENVIRONMENT.email_verification_error;
}
else if (SC.ENVIRONMENT.password_reset_invalid_error)
{
  this.message = _('Your reset password link is invalid. Request a new one using the Forgot Password
link.').translate();
  delete SC.ENVIRONMENT.password_reset_invalid_error;
}
else if (SC.ENVIRONMENT.password_reset_expired_error)
{
  this.message = _('Your reset password link has expired. Request a new one using the Forgot Password
link.').translate();
  delete SC.ENVIRONMENT.password_reset_expired_error;
}

BackboneCompositeView.add(this);
```

Replace

```
, 'Register': function ()
{
  return new RegisterView(this.child_view_options);
}
```

With

```
, 'Register': function ()
{
  return new RegisterView(this.child_view_options);
},
'Messages': function ()
```

```

{
  if (this.message)
  {
    return new GlobalViewsMessageView({
      message: this.message
      , type: 'error'
      , closable: true
    });
  }
}

```

Modify LoginRegister/Templates/login_register.tpl

Replace

```

<header class="login-register-header">
  {{#if showRegister}}
    <h1 class="login-register-title"> {{translate 'Log in | Register'}}</h1>
  {{else}}
    <h1 class="login-register-title"> {{translate 'Log in'}}</h1>
  {{/if}}
</header>

```

With

```

<header class="login-register-header">
  {{#if showRegister}}
    <h1 class="login-register-title"> {{translate 'Log in | Register'}}</h1>
  {{else}}
    <h1 class="login-register-title"> {{translate 'Log in'}}</h1>
  {{/if}}
</header>
<div data-view="Messages"></div>

```

Modify MyAccountApplication/SuiteScript/my_account.ssp

Replace

```

<%
  var SiteSettings
  , siteType
  , Environment
  , Language
  , Currency
  , Error
  , Application;

  try
  {
    Application = require('Application');
    SiteSettings = require('SiteSettings.Model').get();
    siteType = SiteSettings.sitetype;

```

```

Environment = Application.getEnvironment(session, request);

Language = Environment.currentLanguage && Environment.currentLanguage.locale || '';
Currency = Environment.currencyCodeSpecifiedOnUrl;

// Access control, if you are not loged this will send you to the log in page
if (!session.isLoggedIn2() || session.getCustomer().isGuest())
{
    var parameters = request.getAllParameters();

    delete parameters.sitepath;
    parameters.origin = 'customercenter';

    if (parameters.fragment)
    {
        parameters.origin_hash = parameters.fragment;
        delete parameters.fragment;
    }

    return nlapiSetRedirectURL('EXTERNAL', SiteSettings.touchpoints.login, null, false, parameters);
}
} catch (e) {
    Error = Application.processError(e);
}

%>

```

With

```

<%
var SiteSettings
, siteType
, Environment
, Language
, Currency
, Error
, Application
, parameters
, email_change_verification
;

try
{
    Application = require('Application');
    SiteSettings = require('SiteSettings.Model').get();
    parameters = request.getAllParameters();
    siteType = SiteSettings.sitetype;

    Environment = Application.getEnvironment(session, request);

    Language = Environment.currentLanguage && Environment.currentLanguage.locale || '';
    Currency = Environment.currencyCodeSpecifiedOnUrl;

```

```

// Access control, if you are not loged this will send you to the log in page
if (!session.isLoggedIn2() || session.getCustomer().isGuest())
{
    delete parameters.sitepath;
    parameters.origin = 'customercenter';

    if (parameters.fragment)
    {
        parameters.origin_hash = parameters.fragment;
        delete parameters.fragment;
    }

    return nlapiSetRedirectURL('EXTERNAL', SiteSettings.touchpoints.login, null, false, parameters);
}
else if (session.isLoggedIn2() && parameters.key)
{
    try
    {
        session.verifyEmailChange(parameters.key)
        email_change_verification = true;
    }
    catch (e)
    {
        email_change_verification = e.details;
    }
}
} catch (e) {
    Error = Application.processError(e);
}
%>

```

Replace

```

<% if (Error) { %>
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>

```

With

```

<% if (Error) { %>
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>
<% if (email_change_verification) { %>
<script>SC.SESSION.email_change_verification = '<%= email_change_verification %>';</script>
<% } %>

```

Modify Overview/Javascript/Overview.Home.View.js

Replace

```

define('Overview.Home.View'
, [
    'SC.Configuration'
, 'Overview.Banner.View'

```

```

        , 'Overview.Profile.View'
        , 'Overview.Payment.View'
        , 'Overview.Shipping.View'
        , 'Backbone.CollectionView'
        , 'OrderHistory.List.Tracking.Number.View'
        , 'RecordViews.View'
        , 'Handlebars'

        , 'overview_home.tpl'

        , 'Backbone'
        , 'Backbone.CompositeView'
        , 'underscore'
        , 'Utils'
    ]
, function(
    Configuration
, OverviewBannerView
, OverviewProfileView
, OverviewPaymentView
, OverviewShippingView
, BackboneCollectionView
, OrderHistoryListTrackingNumberView
, RecordViewsView
, Handlebars

, overview_home_tpl

, Backbone
, BackboneCompositeView
'-
)

```

With

```

define('Overview.Home.View'
, [
    'SC.Configuration'
, 'GlobalViews.Message.View'
, 'Overview.Banner.View'
, 'Overview.Profile.View'
, 'Overview.Payment.View'
, 'Overview.Shipping.View'
, 'Backbone.CollectionView'
, 'OrderHistory.List.Tracking.Number.View'
, 'RecordViews.View'
, 'Handlebars'

, 'overview_home.tpl'

, 'Backbone'
, 'Backbone.CompositeView'
, 'underscore'
, 'Utils'
]

```

```

, function(
  Configuration
, GlobalViewsMessageView
, OverviewBannerView
, OverviewProfileView
, OverviewPaymentView
, OverviewShippingView
, BackboneCollectionView
, OrderHistoryListTrackingNumberView
, RecordViewsView
, Handlebars

, overview_home_tpl

, Backbone
, BackboneCompositeView
'-
)

```

Replace

```

this.creditcards.on('reset destroy change add', this.showContent, this);
}

```

With

```

this.creditcards.on('reset destroy change add', this.showContent, this);

if (SC.SESSION.email_change_verification)
{
  this.email_change_verification = SC.SESSION.email_change_verification;
  delete SC.SESSION.email_change_verification;
}
}

```

Replace

```

, 'Overview.Shipping': function()
{
  return new OverviewShippingView({ model: this.defaultShippingAddress });
}

```

With

```

, 'Overview.Shipping': function()
{
  return new OverviewShippingView({ model: this.defaultShippingAddress });
}

, 'Overview.Messages': function ()
{
  if (this.email_change_verification)
  {
    return new GlobalViewsMessageView({

```

```

        message: this.email_change_verification === 'true' ? _('Your email has been changed successfully to
<strong>').translate() + this.model.get('email') + '</strong>' : this.email_change_verification
        , type: this.email_change_verification === 'true' ? 'success' : 'error'
        , closable: true
    });
}
}

```

Modify Overview/Templates/overview_home.tpl

Replace

```
<section class="overview-home">
  <div class="overview-home-orders" data-permissions="{{purchasesPermissions}}>
```

With

```
<section class="overview-home">
  <div data-view="Overview.Messages"></div>
  <div class="overview-home-orders" data-permissions="{{purchasesPermissions}}>
```

Add Profile/Javascript/Profile.ChangeEmailAddress.Model.js

```

// Profile.ChangeEmailAddress.Model.js
// -----
// View Model for changing user's email
// @module Profile
define(
  'Profile.ChangeEmailAddress.Model'
, [
  'Backbone'
, 'underscore'
, 'Utils'
]
, function (
  Backbone
)
{
  'use strict';

  // @class Profile.ChangeEmailAddress.Model @extends Backbone.Model
  return Backbone.Model.extend(
  {
    urlRoot: 'services/Profile.Service.ss'
  , validation: {
      current_password: { required: true, msg: _('Current password is required').translate() }
    , confirm_email: [
        { required: true, msg: _('Confirm Email is required').translate() }
      , { equalTo: 'new_email', msg: _('New Email and Confirm New Email do not match').translate() }
    ]
    , new_email: { required: true, msg: _('New Email is required').translate() }
    }
  });
}

```

});

Add Profile/Javascript/Profile.ChangeEmailAddress.View.js

```
// @module Profile
define(
  'Profile.ChangeEmailAddress.View',
  [
    'GlobalViews.Message.View',
    'Backbone.FormView',
    'SC.Configuration'

    , 'profile_change_email.tpl'

    , 'Backbone'
    , 'underscore'
    , 'Utils'
  ]
, function (
  GlobalViewsMessageView
, BackboneFormView
, Configuration

, profile_change_email_tpl

, Backbone
, _
)
{
  'use strict';

// @class Profile.ChangeEmailAddress.View @extends Backbone.View
return Backbone.View.extend({

  template: profile_change_email_tpl

  , page_header: _('Change Email').translate()

  , title: _('Change Email').translate()

  , events: {
    'submit form': 'saveFormCustom'
  }

  , bindings: {
    '[name="current_password"]': 'current_password'
    , '[name="new_email"]': 'new_email'
    , '[name="confirm_email"]': 'confirm_email'
  }

  , initialize: function()
  {
    Backbone.View.prototype.initialize.apply(this, arguments);
    BackboneFormView.add(this);
  }
})
```

```

    , saveFormCustom: function ()
    {
        this.new_email = this.$('[name="new_email"]').val();
        BackboneFormView.saveForm.apply(this, arguments);
    }

    , showSuccess: function (placeholder)
    {
        var global_view_message = new GlobalViewsMessageView({
            message: _('A confirmation email has been sent to <strong>').translate() + this.new_email + '</strong>',
            type: 'success',
            closable: true
        });

        placeholder.html(global_view_message.render().$el.html());
    }
});
});
});

```

Modify Profile/Javascript/Profile.Information.View.js

Replace

```

define(
    'Profile.Information.View'
    , [
        'SC.Configuration',
        'GlobalViews.Message.View',
        'Backbone.FormView'

        , 'profile_information.tpl'

        , 'Backbone'
        , 'underscore'
        , 'jQuery'
        , 'Utils'
    ]
    , function (
        Configuration
        , GlobalViewsMessageView
        , BackboneFormView

        , profile_information_tpl

        , Backbone
        ,
        ,
        , jQuery
    )
{
    'use strict';

    // @class Profile.Information.View @extends Backbone.View
    return Backbone.View.extend({

```

```

    template: profile_information_tpl
    , page_header: _('Profile Information').translate()
    , title: _('Profile Information').translate()
    , attributes: {'class': 'ProfileInformationView'}
    , events: {
        'submit form': 'saveForm'
        , 'change input[data-type="phone"]': 'formatPhone'
    }

    , bindings: {
        '[name="firstname)": "firstname"
        , '[name="lastname"]": "lastname"
        , '[name="companyname"]": "companyname"
        , '[name="phone"]": "phone"
    }

    , initialize: function()
    {
        BackboneFormView.add(this);
    }

    , formatPhone: function (e)
    {
        var $target = jQuery(e.target);
        $target.val(_.($target.val()).formatPhone());
    }

    , showSuccess: function ()

```

With

```

define(
    'Profile.Information.View'
    , [
        'SC.Configuration'
        , 'GlobalViews.Message.View'
        , 'Backbone.FormView'

        , 'Profile.ChangeEmailAddress.Model'
        , 'Profile.ChangeEmailAddress.View'

        , 'profile_information.tpl'

        , 'Backbone'
        , 'underscore'
        , 'jQuery'
        , 'Utils'
    ]
    , function (
        Configuration
        , GlobalViewsMessageView
        , BackboneFormView

        , ProfileChangeEmailModel
        , ProfileChangeEmailView

```

```

, profile_information_tpl

, Backbone
'
, -
, jQuery
)
{
'use strict';

// @class Profile.Information.View @extends Backbone.View
return Backbone.View.extend({

  template: profile_information_tpl
, page_header: _('Profile Information').translate()
, title: _('Profile Information').translate()
, attributes: {'class': 'ProfileInformationView'}
, events: {
    'submit form': 'saveForm'
, 'change input[data-type="phone)": "formatPhone"
, 'click [data-action="change-email"]": "changeEmail"
  }

, bindings: {
    '[name="firstname"]': 'firstname'
, '[name="lastname"]': 'lastname'
, '[name="companynname"]': 'companynname'
, '[name="phone"]': 'phone'
  }

, initialize: function(options)
{
  BackboneFormView.add(this);
  this.application = options.application;
}

, formatPhone: function (e)
{
  var $target = jQuery(e.target);
  $target.val(_.($target.val()).formatPhone());
}

, changeEmail: function ()
{
  var model = new ProfileChangeEmailModel(this.model.attributes);

  var view = new ProfileChangeEmailView({
    application: this.application
, model: model
});

  var self = this;

  model.on('save', function () {
    view.showSuccess(self.$('[data-type="alert-placeholder"]'));
  });
}
});

```

```

    });

    view.useLayoutError = true;

    this.application.getLayout().showInModal(view);
}

, showSuccess: function ()

```

Add Profile/Sass/_profile-change-email.scss

```

.profile-change-email-button-back {
    @extend .button-back;
}

.profile-change-email-button-back-icon {
    @extend .button-back-icon;
}

.profile-change-email-form-label {
    display: inline-block;
}

.profile-change-email-form-label-required {
    @extend .input-required;
}

.profile-change-email {
    @extend .address-edit;
}

.profile-change-email-form-title {}
.profile-change-email-form-area {}

.profile-change-email-form {
    margin-top: $sc-base-margin * 3;
}

.profile-change-email-form-group,
.profile-change-email-form-actions {
    @extend .control-group;
}

.profile-change-email-form-group-label {
    @extend .input-label
}

.profile-change-email-form-group-label-required {
}

.profile-change-email-form-group-input {
    @extend .input-large
}

.profile-change-email-form-actions-change {
}

```

```

    @extend .button-primary;
    @extend .button-medium;
}

.profile-change-email-group-form-controls{}

.profile-change-email-form-info-block{}

```

Modify Profile/SuiteScript/Profile.Model.js

Replace

```

define(
  'Profile.Model'
, ['SC.Model', 'Utils']
, function (SCModel, Utils)

```

With

```

define(
  'Profile.Model'
, ['SC.Model', 'Models.Init', 'Utils']
, function (SCModel, ModelsInit, Utils)

```

Replace

```

if (data.email && data.email !== this.currentSettings.email && data.email === data.confirm_email)
{
  if(data.isGuest === 'T')
  {
    customerUpdate.email = data.email;
  }
  else
  {
    login.changeEmail(data.current_password, data.email, true);
  }
}

```

With

```

if (data.email && data.email !== this.currentSettings.email && data.email === data.confirm_email && data.isGuest
  === 'T')
{
  customerUpdate.email = data.email;
}
else if (data.new_email && data.new_email === data.confirm_email && data.new_email !
  == this.currentSettings.email)
{
  ModelsInit.session.login({
    email: data.email
    , password: data.current_password
  });
  login.changeEmail(data.current_password, data.new_email, true);
}

```

{}

Add Profile/Templates/profile_change_email.tpl

```
<section class="profile-change-email">
    <div data-type="alert-placeholder"></div>
    <div class="profile-change-email-form-area">
        <form class="profile-change-email-form">
            <fieldset>
                <small class="profile-change-email-form-label">{{translate 'Required'}} <span class="profile-change-email-form-group-label-required">*</span></small>

                <div class="profile-change-email-form-group" data-input="new_email" data-validation="control-group">
                    <label class="profile-change-email-form-group-label" for="new_email">{{translate 'New Email'}}</label>
                    <span class="profile-change-email-form-group-label-required">*</span></label>
                    <div class="profile-change-email-group-form-controls" data-validation="control">
                        <input type="email" class="profile-change-email-form-group-input" id="new_email" name="new_email" value="" placeholder="{{translate 'your@email.com'}}">
                    </div>
                </div>

                <div class="profile-change-email-form-group" data-input="confirm_email" data-validation="control-group">
                    <label class="profile-change-email-form-group-label" for="confirm_email">{{translate 'Confirm New Email'}}</label>
                    <span class="profile-change-email-form-group-label-required">*</span></label>
                    <div class="profile-change-email-group-form-controls" data-validation="control">
                        <input type="email" class="profile-change-email-form-group-input" id="confirm_email" name="confirm_email" value="" placeholder="{{translate 'your@email.com'}}">
                    </div>
                </div>

                <div class="profile-change-email-form-group" data-input="current_email" data-validation="control-group">
                    <label class="profile-change-email-form-group-label" for="current_password">{{translate 'Password'}}</label>
                    <span class="profile-change-email-form-group-label-required">*</span></label>
                    <div class="profile-change-email-group-form-controls" data-validation="control">
                        <input type="password" class="profile-change-email-form-group-input" id="current_password" name="current_password" value="">
                    </div>
                </div>
            </fieldset>
            <p class="profile-change-email-form-info-block"><small> {{translate 'You will still be able to login with your current email address and password until your new email address is verified.'}} </small></p>
            <div class="profile-change-email-form-actions">
                <button type="submit" class="profile-change-email-form-actions-change">{{translate 'Send Verification Email'}}</button>
            </div>
        </form>
    </div>
</section>
```

Modify Profile/Templates/profile_information.tpl

Replace

```
<p class="profile-information-input-email" id="email">
{{email}}
</p>
```

With

```
<p class="profile-information-input-email" id="email">{{email}} | <a class="profile-information-change-email-address" data-action="change-email">{{translate 'Change Address'}}</a></p>
```

Vinson — Change Email Address Patch

ⓘ Applies to: SuiteCommerce Advanced | Vinson

To update a Vinson implementation to give users the ability to change their email address, you'll need to modify and add the files as detailed below.

⚠ Important: This addition requires changes to templates, views, JavaScript, and SSP application files. Test changes thoroughly against your existing customizations before deploying to your published domain.

Modify CheckoutApplication/SuiteScript/checkout.ssp

Replace

```
var SiteSettings
, parameters
, siteType
, Environment
, Language
, Currency
, Error
, login
, Application
, environmentParameters
```

With

```
var SiteSettings
, parameters
, siteType
, Environment
, Language
, Currency
, Error
, login
, Application
, environmentParameters
, password_reset_expired
;
```

Replace

```
if (session.isChangePasswordRequest())
{
    parameters.fragment = 'reset-password';
    login = true;
}
```

With

```
if (parameters.passwdret)
{
    try
    {
        if (session.isChangePasswordRequest())
        {
            parameters.fragment = 'reset-password';
            login = true;
        }
    }
    catch (e)
    {
        password_reset_expired = true;
    }
}
```

Replace

```
<% if (Error) { %>
    <script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>
```

With

```
<% if (Error) { %>
    <script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>

<% if (parameters.key) { %>
    <script>SC.ENVIRONMENT.email_verification_error = true;</script>
<% } else if (password_reset_expired) { %>
    <script>SC.ENVIRONMENT.password_reset_expired_error = true;</script>
<% } else if (parameters.passwdret && parameters.fragment !== 'reset-password') { %>
    <script>SC.ENVIRONMENT.password_reset_invalid_error = true;</script>
<% } %>
```

Modify LoginRegister/Javascript/LoginRegister.View.js

Replace

```
define('LoginRegister.View'
```

```

, [ 'login_register.tpl'

, 'Profile.Model'
, 'LoginRegister.Login.View'
, 'LoginRegister.Register.View'
, 'LoginRegister.CheckoutAsGuest.View'
, 'Backbone.CompositeView'
, 'SC.Configuration'
, 'Header.Simplified.View'
, 'Footer.Simplified.View'

, 'Backbone'
, 'underscore'
, 'Utils'
]
, function (
  login_register_tpl

, ProfileModel
, LoginView
, RegisterView
, CheckoutAsGuestView
, BackboneCompositeView
, Configuration
, HeaderSimplifiedView
, FooterSimplifiedView

, Backbone
' -
)

```

With

```

define('LoginRegister.View'
, [ 'login_register.tpl'

, 'GlobalViews.Message.View'
, 'Profile.Model'
, 'LoginRegister.Login.View'
, 'LoginRegister.Register.View'
, 'LoginRegister.CheckoutAsGuest.View'
, 'Backbone.CompositeView'
, 'SC.Configuration'
, 'Header.Simplified.View'
, 'Footer.Simplified.View'

, 'Backbone'
, 'underscore'
, 'Utils'
]
, function (
  login_register_tpl

, GlobalViewsMessageView
, ProfileModel

```

```

    , LoginView
    , RegisterView
    , CheckoutAsGuestView
    , BackboneCompositeView
    , Configuration
    , HeaderSimplifiedView
    , FooterSimplifiedView

    , Backbone
    ' -
)

```

Replace

```

this.enableCheckoutAsGuest = is_checking_out && profile_model.get('isLoggedIn') === 'F' && (Configuration.getRegistrationType() === 'optional' || Configuration.getRegistrationType() === 'disabled');

BackboneCompositeView.add(this);

```

With

```

this.enableCheckoutAsGuest = is_checking_out && profile_model.get('isLoggedIn') === 'F' &&
(Configuration.getRegistrationType() === 'optional' || Configuration.getRegistrationType() === 'disabled');

if (SC.ENVIRONMENT.email_verification_error)
{
    this.message = _('The validation process has failed. Please login into your account and click on the validation link again.').translate();
    delete SC.ENVIRONMENT.email_verification_error;
}
else if (SC.ENVIRONMENT.password_reset_invalid_error)
{
    this.message =_('Your reset password link is invalid. Request a new one using the Forgot Password link.').translate();
    delete SC.ENVIRONMENT.password_reset_invalid_error;
}
else if (SC.ENVIRONMENT.password_reset_expired_error)
{
    this.message =_('Your reset password link has expired. Request a new one using the Forgot Password link.').translate();
    delete SC.ENVIRONMENT.password_reset_expired_error;
}

BackboneCompositeView.add(this);

```

Replace

```

, 'Register': function ()
{
    return new RegisterView(this.child_view_options);
}

```

With

```
, 'Register': function () {
  {
    return new RegisterView(this.child_view_options);
  }
, 'Messages': function () {
  {
    if (this.message)
    {
      return new GlobalViewsMessageView({
        message: this.message
        , type: 'error'
        , closable: true
      });
    }
  }
}
```

Modify LoginRegister/Templates/login_register.tpl

Replace

```
<header class="login-register-header">
{{#if showRegister}}
<h1 class="login-register-title"> {{translate 'Log in | Register'}}</h1>
{{else}}
<h1 class="login-register-title"> {{translate 'Log in'}}</h1>
{{/if}}
</header>
```

With

```
<header class="login-register-header">
{{#if showRegister}}
<h1 class="login-register-title"> {{translate 'Log in | Register'}}</h1>
{{else}}
<h1 class="login-register-title"> {{translate 'Log in'}}</h1>
{{/if}}
</header>

<div data-view="Messages"></div>
```

Modify MyAccountApplication/SuiteScript/my_account.ssp

Replace

```
<%
var SiteSettings
, siteType
, Environment
, Language
, Currency
, Error
```

```

, Application
, environmentParameters
, parameters
, external_payment;

try
{
    SiteSettings = require('SiteSettings.Model').get();
    parameters = request.getAllParameters();

    // Access control, if you are not loged this will send you to the log in page
    if (!session.isLoggedIn2() || session.getCustomer().isGuest())
    {
        delete parameters.sitepath;
        parameters.origin = 'customercenter';

        if (parameters.fragment)
        {
            parameters.origin_hash = parameters.fragment;
            delete parameters.fragment;
        }
    }

    return nlapiSetRedirectURL('EXTERNAL', SiteSettings.touchpoints.login, null, false, parameters);
}

Application = require('Application');
Environment = Application.getEnvironment(request);
environmentParameters = [];
siteType = SiteSettings.sitetype;

Language = Environment.currentLanguage && Environment.currentLanguage.locale || '';
Currency = Environment.currencyCodeSpecifiedOnUrl;

environmentParameters.push('lang=' + Language);
environmentParameters.push('cur=' + Currency);

_.each(require('ExternalPayment.Model').getParametersFromRequest(request), function(value, key) {
    environmentParameters.push(key.concat('=', value));
});

}
catch (e)
{
    Error = Application.processError(e);
}

%>

```

With

```

<%
var SiteSettings
, siteType
, Environment
, Language
, Currency

```

```

, Error
, Application
, environmentParameters
, parameters
, external_payment
, email_change_verification
;

try
{
    Application = require('Application');
    Environment = Application.getEnvironment(request);
    environmentParameters = [];
    SiteSettings = require('SiteSettings.Model').get();
    parameters = request.getAllParameters();

    // Access control, if you are not loged this will send you to the log in page
    if (!session.isLoggedIn2() || session.getCustomer().isGuest())
    {
        delete parameters.sitepath;
        parameters.origin = 'customercenter';

        if (parameters.fragment)
        {
            parameters.origin_hash = parameters.fragment;
            delete parameters.fragment;
        }

        return nlapiSetRedirectURL('EXTERNAL', SiteSettings.touchpoints.login, null, false, parameters);
    }
    else if (session.isLoggedIn2() && parameters.key)
    {
        try
        {
            session.verifyEmailChange(parameters.key)
            email_change_verification = true;
        }
        catch (e)
        {
            email_change_verification = e.details;
        }
    }
}

siteType = SiteSettings.sitetype;

Language = Environment.currentLanguage && Environment.currentLanguage.locale || '';
Currency = Environment.currencyCodeSpecifiedOnUrl;

environmentParameters.push('lang=' + Language);
environmentParameters.push('cur=' + Currency);

_.each(require('ExternalPayment.Model').getParametersFromRequest(request), function(value, key) {
    environmentParameters.push(key.concat('=', value));
});
}

```

```

    catch (e)
    {
        Error = Application.processError(e);
    }

%>

```

Replace

```

<% if (Error) { %>
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>

```

With

```

<% if (Error) { %>
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>
<% if (email_change_verification) { %>
<script>SC.SESSION.email_change_verification = '<%= email_change_verification %>';</script>
<% } %>

```

Modify Overview/Javascript/Overview.Home.View.js

Replace

```

define('Overview.Home.View'
, [
    'SC.Configuration'
, 'Overview.Banner.View'
, 'Overview.Profile.View'
, 'Overview.Payment.View'
, 'Overview.Shipping.View'
, 'Backbone.CollectionView'
, 'OrderHistory.List.Tracking.Number.View'
, 'RecordViews.View'
, 'Handlebars'

, 'overview_home.tpl'

, 'Backbone'
, 'Backbone.CompositeView'
, 'underscore'
, 'Utils'
]
, function(
    Configuration
, OverviewBannerView
, OverviewProfileView
, OverviewPaymentView
, OverviewShippingView
, BackboneCollectionView
, OrderHistoryListTrackingNumberView

```

```
, RecordViewsView
, Handlebars

, overview_home_tpl

, Backbone
, BackboneCompositeView
' -
)
```

With

```
define('Overview.Home.View'
, [
  'SC.Configuration'
, 'GlobalViews.Message.View'
, 'Overview.Banner.View'
, 'Overview.Profile.View'
, 'Overview.Payment.View'
, 'Overview.Shipping.View'
, 'Backbone.CollectionView'
, 'OrderHistory.List.Tracking.Number.View'
, 'RecordViews.View'
, 'Handlebars'

, 'overview_home.tpl'

, 'Backbone'
, 'Backbone.CompositeView'
, 'underscore'
, 'Utils'
]
, function(
  Configuration
, GlobalViewsMessageView
, OverviewBannerView
, OverviewProfileView
, OverviewPaymentView
, OverviewShippingView
, BackboneCollectionView
, OrderHistoryListTrackingNumberView
, RecordViewsView
, Handlebars

, overview_home_tpl

, Backbone
, BackboneCompositeView
' -
)
```

Replace

```
this.creditcards.on('reset destroy change add', this.showContent, this);
```

```
}
```

With

```
this.creditcards.on('reset destroy change add', this.showContent, this);

if (SC.SESSION.email_change_verification)
{
    this.email_change_verification = SC.SESSION.email_change_verification;
    delete SC.SESSION.email_change_verification;
}
```

Replace

```
, 'Overview.Shipping': function()
{
    return new OverviewShippingView({ model: this.defaultShippingAddress });
}
```

With

```
, 'Overview.Shipping': function()
{
    return new OverviewShippingView({ model: this.defaultShippingAddress });
}

, 'Overview.Messages': function ()
{
    if (this.email_change_verification)
    {
        return new GlobalViewsMessageView({
            message: this.email_change_verification === 'true' ? _('Your email has been changed successfully to
<strong>').translate() + this.model.get('email') + '</strong>' : this.email_change_verification
            , type: this.email_change_verification === 'true' ? 'success' : 'error'
            , closable: true
        });
    }
}
```

Modify Overview/Templates/overview_home.tpl

Replace

```
<section class="overview-home">
    <div class="overview-home-orders" data-permissions="{{purchasesPermissions}}>
```

With

```
<section class="overview-home">
    <div data-view="Overview.Messages"></div>
    <div class="overview-home-orders" data-permissions="{{purchasesPermissions}}>
```

Add Profile/Javascript/Profile.ChangeEmailAddress.Model.js

```
// Profile.ChangeEmailAddress.Model.js
// -----
// View Model for changing user's email
// @module Profile
define(
  'Profile.ChangeEmailAddress.Model',
  [
    'Backbone',
    'underscore',
    'Utils'
  ],
  function (
    Backbone
  ) {
    'use strict';

    // @class Profile.ChangeEmailAddress.Model @extends Backbone.Model
    return Backbone.Model.extend(
      {
        urlRoot: 'services/Profile.Service.ss',
        validation: {
          current_password: { required: true, msg: _('Current password is required').translate() },
          confirm_email: [
            { required: true, msg: _('Confirm Email is required').translate() },
            { equalTo: 'new_email', msg: _('New Email and Confirm New Email do not match').translate() }
          ],
          new_email: { required: true, msg: _('New Email is required').translate() }
        }
      });
  });
});
```

Add Profile/Javascript/Profile.ChangeEmailAddress.View.js

```
// @module Profile
define(
  'Profile.ChangeEmailAddress.View',
  [
    'GlobalViews.Message.View',
    'Backbone.FormView',
    'SC.Configuration'

    , 'profile_change_email.tpl'

    , 'Backbone'
    , 'underscore'
    , 'Utils'
  ],
  function (
    GlobalViewsMessageView
    , BackboneFormView
```

```

, Configuration

, profile_change_email_tpl

, Backbone
'
)
{
'use strict';

// @class Profile.ChangeEmailAddress.View @extends Backbone.View
return Backbone.View.extend({

  template: profile_change_email_tpl

, page_header: _('Change Email').translate()

, title: _('Change Email').translate()

, events: {
  'submit form': 'saveFormCustom'
}

, bindings: {
  '[name="current_password"]': 'current_password'
, '[name="new_email"]': 'new_email'
, '[name="confirm_email"]': 'confirm_email'
}

, initialize: function()
{
  Backbone.View.prototype.initialize.apply(this, arguments);
  BackboneFormView.add(this);
}

, saveFormCustom: function ()
{
  this.new_email = this.$('[name="new_email"]').val();
  BackboneFormView.saveForm.apply(this, arguments);
}

, showSuccess: function (placeholder)
{
  var global_view_message = new GlobalViewsMessageView({
    message: _('A confirmation email has been sent to <strong>').translate() + this.new_email + '</strong>'
    , type: 'success'
    , closable: true
  });

  placeholder.html(global_view_message.render().$el.html());
}

});
});

```

Modify Profile/Javascript/Profile.Information.View.js

Replace

```

define(
  'Profile.Information.View'
, [
  'SC.Configuration'
, 'GlobalViews.Message.View'
, 'Backbone.FormView'

, 'profile_information.tpl'

, 'Backbone'
, 'underscore'
, 'jQuery'
, 'Utils'
]
, function (
  Configuration
, GlobalViewsMessageView
, BackboneFormView

, profile_information_tpl

, Backbone
, _
, jQuery
)
{
  'use strict';

// @class Profile.Information.View @extends Backbone.View
return Backbone.View.extend({


  template: profile_information_tpl
, page_header: _('Profile Information').translate()
, title: _('Profile Information').translate()
, attributes: {'class': 'ProfileInformationView'}
, events: {
    'submit form': 'saveForm'
, 'change input[data-type="phone"]': 'formatPhone'
  }

, bindings: {
    '[name="firstname)": "firstname"
, '[name="lastname"]": "lastname"
, '[name="companynname"]": "companynname"
, '[name="phone"]": "phone"
  }

, initialize: function()
{
  BackboneFormView.add(this);
}
}

```

```

, formatPhone: function (e)
{
  var $target = jQuery(e.target);
  $target.val(_($target.val()).formatPhone());
}

, showSuccess: function ()

```

With

```

define(
  'Profile.Information.View'
, [
  'SC.Configuration'
, 'GlobalViews.Message.View'
, 'Backbone.FormView'

, 'Profile.ChangeEmailAddress.Model'
, 'Profile.ChangeEmailAddress.View'

, 'profile_information_tpl'

, 'Backbone'
, 'underscore'
, 'jQuery'
, 'Utils'
]
, function (
  Configuration
, GlobalViewsMessageView
, BackboneFormView

, ProfileChangeEmailModel
, ProfileChangeEmailView

, profile_information_tpl

, Backbone
'-
, jQuery
)
{
  'use strict';

// @class Profile.Information.View @extends Backbone.View
return Backbone.View.extend({


  template: profile_information_tpl
, page_header: _('Profile Information').translate()
, title: _('Profile Information').translate()
, attributes: {'class': 'ProfileInformationView'}
, events: {
    'submit form': 'saveForm'
, 'change input[data-type="phone"]': 'formatPhone'

```

```

        , 'click [data-action="change-email"]': 'changeEmail'
    }

    , bindings: {
        '[name="firstname)": "firstname"
        , '[name="lastname)": "lastname"
        , '[name="companyname)": "companyname"
        , '[name="phone)": "phone"
    }

    , initialize: function(options)
    {
        BackboneFormView.add(this);
        this.application = options.application;
    }

    , formatPhone: function (e)
    {
        var $target = jQuery(e.target);
        $target.val(_.($target.val()).formatPhone());
    }

    , changeEmail: function ()
    {
        var model = new ProfileChangeEmailModel(this.model.attributes);

        var view = new ProfileChangeEmailView({
            application: this.application
            , model: model
        });

        var self = this;

        model.on('save', function () {
            view.showSuccess(self.$('[data-type="alert-placeholder"]'));
        });

        view.useLayoutError = true;

        this.application.getLayout().showInModal(view);
    }

    , showSuccess: function ()

```

Add Profile/Sass/_profile-change-email.scss

```

.profile-change-email-button-back {
    @extend .button-back;
}

.profile-change-email-button-back-icon {
    @extend .button-back-icon;
}

.profile-change-email-form-label {

```

```

        display: inline-block;
    }

.profile-change-email-form-group-label-required {
    @extend .input-required;
}

.profile-change-email {
    @extend .address-edit;
}

.profile-change-email-form-title {}
.profile-change-email-form-area {}

.profile-change-email-form {
    margin-top: $sc-base-margin * 3;
}

.profile-change-email-form-group,
.profile-change-email-form-actions {
    @extend .control-group;
}

.profile-change-email-form-group-label {
    @extend .input-label
}

.profile-change-email-form-group-label-required {}

.profile-change-email-form-group-input {
    @extend .input-large
}

.profile-change-email-form-actions-change {
    @extend .button-primary;
    @extend .button-medium;
}

.profile-change-email-group-form-controls{}

.profile-change-email-form-info-block{}

```

Modify Profile/SuiteScript/Profile.Model.js

Replace

```

if (data.email && data.email !== this.currentSettings.email && data.email === data.confirm_email)
{
    if(data.isGuest === 'T')
    {
        customerUpdate.email = data.email;
    }
    else

```

```
{
    login.changeEmail(data.current_password, data.email, true);
}
}
```

With

```
if (data.email && data.email !== this.currentSettings.email && data.email === data.confirm_email && data.isGuest
    === 'T')
{
    customerUpdate.email = data.email;
}
else if (data.new_email && data.new_email === data.confirm_email && data.new_email !
    == this.currentSettings.email)
{
    ModelsInit.session.login({
        email: data.email
        , password: data.current_password
    });
    login.changeEmail(data.current_password, data.new_email, true);
}
```

Add Profile/Templates/profile_change_email.tpl

```
<section class="profile-change-email">
<div data-type="alert-placeholder"></div>
<div class="profile-change-email-form-area">
    <form class="profile-change-email-form">
        <fieldset>
            <small class="profile-change-email-form-label">{{translate 'Required'}} <span class="profile-change-email-form-group-label-required">*</span></small>

            <div class="profile-change-email-form-group" data-input="new_email" data-validation="control-group">
                <label class="profile-change-email-form-group-label" for="new_email">{{translate 'New
Email'}}</label> <span class="profile-change-email-form-group-label-required">*</span></label>
                <div class="profile-change-email-group-form-controls" data-validation="control">
                    <input type="email" class="profile-change-email-form-group-
input" id="new_email" name="new_email" value="" placeholder="{{translate 'your@email.com'}}">
                </div>
            </div>

            <div class="profile-change-email-form-group" data-input="confirm_email" data-validation="control-group">
                <label class="profile-change-email-form-group-label" for="confirm_email">{{translate 'Confirm New
Email'}}</label> <span class="profile-change-email-form-group-label-required">*</span></label>
                <div class="profile-change-email-group-form-controls" data-validation="control">
                    <input type="email" class="profile-change-email-form-group-input" id="confirm_email" name="confirm_
email" value="" placeholder="{{translate 'your@email.com'}}">
                </div>
            </div>

            <div class="profile-change-email-form-group" data-input="current_email" data-validation="control-group">
                <label class="profile-change-email-form-group-label" for="current_password">{{translate 'Password'}}</label> <span class="profile-change-email-form-group-label-required">*</span></label>
                <div class="profile-change-email-group-form-controls" data-validation="control">
```

```

        <input type="password" class="profile-change-email-form-group-input" id="current_password"
      name="current_password" value="">
      </div>
      </div>
    </fieldset>
    <p class="profile-change-email-form-info-block"><small> {{translate 'You will still be able to login with
      your current email address and password until your new email address is verified.'}} </small></p>
    <div class="profile-change-email-form-actions">
      <button type="submit" class="profile-change-email-form-actions-change">{{translate 'Send Verification
      Email'}}</button>
    </div>
  </form>
</div>
</section>

```

Modify Profile/Templates/profile_information.tpl

Replace

```

<p class="profile-information-input-email" id="email">
  {{email}}
</p>

```

With

```

<p class="profile-information-input-email" id="email">{{email}} | <a class="profile-information-change-email-
  address" data-action="change-email">{{translate 'Change Address'}}</a></p>

```

Elbrus — Change Email Address Patch

ⓘ Applies to: SuiteCommerce Advanced | Elbrus

To update an Elbrus implementation to give users the ability to change their email address, you'll need to modify and add the files as detailed below.



Important: This addition requires changes to templates, views, JavaScript, and SSP application files. Test changes thoroughly against your existing customizations before deploying to your published domain.

Modify CheckoutApplication/SuiteScript/checkout.ssp

Replace

```

var SiteSettings
, parameters
, siteType
, Environment
, Language
, Currency
, Error

```

```
, login
, order
, session
, Application
, environmentParameters
;
```

With

```
var SiteSettings
, parameters
, siteType
, Environment
, Language
, Currency
, Error
, login
, order
, session
, Application
, environmentParameters
, password_reset_expired
;
```

Replace

```
if (session.isChangePasswordRequest())
{
    parameters.fragment = 'reset-password';
    login = true;
}
```

With

```
if (parameters.passwdret)
{
    try
    {
        if (session.isChangePasswordRequest())
        {
            parameters.fragment = 'reset-password';
            login = true;
        }
    }
    catch (e)
    {
        password_reset_expired = true;
    }
}
```

Replace

```
<% if (Error) { %>
```

```
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>
```

With

```
<% if (Error) { %>
  <script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>

<% if (parameters.key) { %>
  <script>SC.ENVIRONMENT.email_verification_error = true;</script>
<% } else if (password_reset_expired) { %>
  <script>SC.ENVIRONMENT.password_reset_expired_error = true;</script>
<% } else if (parameters.passwdret && parameters.fragment !== 'reset-password') { %>
  <script>SC.ENVIRONMENT.password_reset_invalid_error = true;</script>
<% } %>
```

Modify LoginRegister/Javascript/LoginRegister.View.js

Replace

```
define('LoginRegister.View',
  [ 'login_register.tpl'

    , 'Profile.Model'
    , 'LoginRegister.Login.View'
    , 'LoginRegister.Register.View'
    , 'LoginRegister.CheckoutAsGuest.View'
    , 'Backbone.CompositeView'
    , 'SC.Configuration'
    , 'Header.Simplified.View'
    , 'Footer.Simplified.View'

    , 'Backbone'
    , 'underscore'
    , 'Utils'
  ]
, function (
  login_register_tpl

  , ProfileModel
  , LoginView
  , RegisterView
  , CheckoutAsGuestView
  , BackboneCompositeView
  , Configuration
  , HeaderSimplifiedView
  , FooterSimplifiedView

  , Backbone
  '-
  , Utils
)
```

With

```
define('LoginRegister.View'
, [ 'login_register.tpl'

, 'GlobalViews.Message.View'
, 'Profile.Model'
, 'LoginRegister.Login.View'
, 'LoginRegister.Register.View'
, 'LoginRegister.CheckoutAsGuest.View'
, 'Backbone.CompositeView'
, 'SC.Configuration'
, 'Header.Simplified.View'
, 'Footer.Simplified.View'

, 'Backbone'
, 'underscore'
, 'Utils'
]
, function (
  login_register_tpl

, GlobalViewsMessageView
, ProfileModel
, LoginView
, RegisterView
, CheckoutAsGuestView
, BackboneCompositeView
, Configuration
, HeaderSimplifiedView
, FooterSimplifiedView

, Backbone
'-
, Utils
)
```

Replace

```
this.enableCheckoutAsGuest = is_checking_out && profile_model.get('isLoggedIn') === 'F' &&
(Configuration.getRegistrationType() === 'optional' || Configuration.getRegistrationType() === 'disabled');

BackboneCompositeView.add(this);
```

With

```
this.enableCheckoutAsGuest = is_checking_out && profile_model.get('isLoggedIn') === 'F' &&
(Configuration.getRegistrationType() === 'optional' || Configuration.getRegistrationType() === 'disabled');

if (SC.ENVIRONMENT.email_verification_error)
{
  this.message = _('The validation process has failed. Please login into your account and click on the validation
link again.').translate();
  delete SC.ENVIRONMENT.email_verification_error;
```

```

    }
    else if (SC.ENVIRONMENT.password_reset_invalid_error)
    {
        this.message = _('Your reset password link is invalid. Request a new one using the Forgot Password
link.').translate();
        delete SC.ENVIRONMENT.password_reset_invalid_error;
    }
    else if (SC.ENVIRONMENT.password_reset_expired_error)
    {
        this.message = _('Your reset password link has expired. Request a new one using the Forgot Password
link.').translate();
        delete SC.ENVIRONMENT.password_reset_expired_error;
    }

    BackboneCompositeView.add(this);
}

```

Replace

```

, 'Register': function ()
{
    return new RegisterView(this.child_view_options);
}

```

With

```

{
    return new RegisterView(this.child_view_options);
}
, 'Messages': function ()
{
    if (this.message)
    {
        return new GlobalViewsMessageView({
            message: this.message
            , type: 'error'
            , closable: true
        });
    }
}

```

Modify LoginRegister/Templates/login_register.tpl

Replace

```

<header class="login-register-header">
{{#if showRegister}}
<h1 class="login-register-title"> {{translate 'Log in | Register'}}</h1>
{{else}}
<h1 class="login-register-title"> {{translate 'Log in'}}</h1>
{{/if}}

```

```
</header>
```

With

```
<header class="login-register-header">
{{#if showRegister}}
<h1 class="login-register-title"> {{translate 'Log in | Register'}}</h1>
{{else}}
<h1 class="login-register-title"> {{translate 'Log in'}}</h1>
{{/if}}
</header>

<div data-view="Messages"></div>
```

Modify MyAccountApplication/SuiteScript/my_account.ssp

Replace

```
<%
var SiteSettings
, siteType
, Environment
, Language
, Currency
, Error
, Application
, environmentParameters
, session
, parameters
, external_payment
;

try
{
    SiteSettings = require('SiteSettings.Model').get();
    parameters = request.getAllParameters();
    session = require('SC.Models.Init').session;

    // Access control, if you are not loged this will send you to the log in page
    if (!session.isLoggedIn() || session.getCustomer().isGuest())
    {
        delete parameters.sitepath;
        parameters.origin = 'customercenter';

        if (parameters.fragment)
        {
            parameters.origin_hash = parameters.fragment;
            delete parameters.fragment;
        }

        return nlapiSetRedirectURL('EXTERNAL', SiteSettings.touchpoints.login, null, false, parameters);
    }

    Application = require('Application');
```

```

Environment = Application.getEnvironment(request);
environmentParameters = [];
siteType = SiteSettings.sitetype;

Language = Environment.currentLanguage && Environment.currentLanguage.locale || '';
Currency = Environment.currencyCodeSpecifiedOnUrl;

environmentParameters.push('lang=' + Language);
environmentParameters.push('cur=' + Currency);
environmentParameters.push('X-SC-Touchpoint=myaccount');

_.each(require('ExternalPayment.Model').getParametersFromRequest(request), function(value, key) {
    environmentParameters.push(key.concat('=', value));
});

}
catch (e)
{
    Error = Application.processError(e);
}

%>

```

With

```

<%
var SiteSettings
, siteType
, Environment
, Language
, Currency
, Error
, Application
, environmentParameters
, session
, parameters
, external_payment
, email_change_verification
;

try
{
    Application = require('Application');
    Environment = Application.getEnvironment(request);
    environmentParameters = [];
    SiteSettings = require('SiteSettings.Model').get();
    parameters = request.getAllParameters();
    session = require('SC.Models.Init').session;

    // Access control, if you are not loged this will send you to the log in page
    if (!session.isLoggedIn() || session.getCustomer().isGuest())
    {
        delete parameters.sitepath;
        parameters.origin = 'customercenter';

        if (parameters.fragment)

```

```

{
    parameters.origin_hash = parameters.fragment;
    delete parameters.fragment;
}

return nlapiSetRedirectURL('EXTERNAL', SiteSettings.touchpoints.login, null, false, parameters);
}
else if (session.isLoggedIn2() && parameters.key)
{
    try
    {
        session.verifyEmailChange(parameters.key)
        email_change_verification = true;
    }
    catch (e)
    {
        email_change_verification = e.details;
    }
}

siteType = SiteSettings.sitetype;

Language = Environment.currentLanguage && Environment.currentLanguage.locale || '';
Currency = Environment.currencyCodeSpecifiedOnUrl;

environmentParameters.push('lang=' + Language);
environmentParameters.push('cur=' + Currency);
environmentParameters.push('X-SC-Touchpoint=myaccount');

_.each(require('ExternalPayment.Model').getParametersFromRequest(request), function(value, key) {
    environmentParameters.push(key.concat('=', value));
});

}
catch (e)
{
    Error = Application.processError(e);
}

%>

```

Replace

```

<% if (Error) { %>
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>

```

With

```

<% if (Error) { %>
<script>SC.ENVIRONMENT.contextError = <%= JSON.stringify(Error) %>;</script>
<% } %>
<% if (email_change_verification) { %>
<script>SC.SESSION.email_change_verification = '<%= email_change_verification %>';</script>
<% } %>

```

Modify Overview/Javascript/Overview.Home.View.js

Replace

```
define('Overview.Home.View'
, [
  'SC.Configuration'
, 'Overview.Banner.View'
, 'Overview.Profile.View'
, 'Overview.Payment.View'
, 'Overview.Shipping.View'
, 'Backbone.CollectionView'
, 'OrderHistory.List.Tracking.Number.View'
, 'RecordViews.View'
, 'Handlebars'

, 'overview_home_tpl'

, 'Backbone'
, 'Backbone.CompositeView'
, 'underscore'
, 'Utils'
]
, function(
  Configuration
, OverviewBannerView
, OverviewProfileView
, OverviewPaymentView
, OverviewShippingView
, BackboneCollectionView
, OrderHistoryListTrackingNumberView
, RecordViewsView
, Handlebars

, overview_home_tpl

, Backbone
, BackboneCompositeView
'-
)
)
```

With

```
define('Overview.Home.View'
, [
  'SC.Configuration'
, 'GlobalViews.Message.View'
, 'Overview.Banner.View'
, 'Overview.Profile.View'
, 'Overview.Payment.View'
, 'Overview.Shipping.View'
, 'Backbone.CollectionView'
, 'OrderHistory.List.Tracking.Number.View'
, 'RecordViews.View'
, 'Handlebars'
```

```

        , 'overview_home.tpl'

        , 'Backbone'
        , 'Backbone.CompositeView'
        , 'underscore'
        , 'Utils'
    ]
, function(
    Configuration
, GlobalViewsMessageView
, OverviewBannerView
, OverviewProfileView
, OverviewPaymentView
, OverviewShippingView
, BackboneCollectionView
, OrderHistoryListTrackingNumberView
, RecordViewsView
, Handlebars

, overview_home_tpl

, Backbone
, BackboneCompositeView
' -
)

```

Replace

```

this.creditcards.on('reset destroy change add', this.showContent, this);
}

```

With

```

this.creditcards.on('reset destroy change add', this.showContent, this);

if (SC.SESSION.email_change_verification)
{
    this.email_change_verification = SC.SESSION.email_change_verification;
    delete SC.SESSION.email_change_verification;
}
}

```

Replace

```

, 'Overview.Shipping': function()
{
    return new OverviewShippingView({ model: this.defaultShippingAddress });
}

```

With

```

, 'Overview.Shipping': function()
{

```

```

        return new OverviewShippingView({ model: this.defaultShippingAddress });
    }

    , 'Overview.Messages': function ()
    {
        if (this.email_change_verification)
        {
            return new GlobalViewsMessageView({
                message: this.email_change_verification === 'true' ? _('Your email has been changed successfully to <strong>') + this.model.get('email') + '</strong>' : this.email_change_verification
                , type: this.email_change_verification === 'true' ? 'success' : 'error'
                , closable: true
            });
        }
    }
}

```

Modify Overview/Templates/overview_home.tpl

Replace

```
<section class="overview-home">
<div class="overview-home-orders" data-permissions="{{purchasesPermissions}}>
```

With

```
<section class="overview-home">
<div data-view="Overview.Messages"></div>
<div class="overview-home-orders" data-permissions="{{purchasesPermissions}}>
```

Add Profile/Javascript/Profile.ChangeEmailAddress.Model.js

```

// Profile.ChangeEmailAddress.Model.js
// =====
// View Model for changing user's email
// @module Profile
define(
    'Profile.ChangeEmailAddress.Model'
, [
    'Backbone'
, 'underscore'
, 'Utils'
]
, function (
    Backbone
)
{
    'use strict';

    // @class Profile.ChangeEmailAddress.Model @extends Backbone.Model
    return Backbone.Model.extend(
    {
        urlRoot: 'services/Profile.Service.ss'
        , validation: {

```

```

        current_password: { required: true, msg: _('Current password is required').translate() }
      , confirm_email: [
        { required: true, msg: _('Confirm Email is required').translate() }
        , { equalTo: 'new_email', msg: _('New Email and Confirm New Email do not match').translate() }
      ]
      , new_email: { required: true, msg: _('New Email is required').translate() }
    }
  );
});

```

Add Profile/Javascript/Profile.ChangeEmailAddress.View.js

```

// @module Profile
define(
  'Profile.ChangeEmailAddress.View'
, [
  'GlobalViews.Message.View'
, 'Backbone.FormView'
, 'SC.Configuration'

, 'profile_change_email.tpl'

, 'Backbone'
, 'underscore'
, 'Utils'
]
, function (
  GlobalViewsMessageView
, BackboneFormView
, Configuration

, profile_change_email_tpl

, Backbone
)
{
  'use strict';

// @class Profile.ChangeEmailAddress.View @extends Backbone.View
return Backbone.View.extend({

  template: profile_change_email_tpl

  , page_header: _('Change Email').translate()

  , title: _('Change Email').translate()

  , events: {
    'submit form': 'saveFormCustom'
  }

  , bindings: {
    '[name="current_password"]': 'current_password'
    , '[name="new_email"]': 'new_email'
  }
}

```

```

        , '[name="confirm_email"]': 'confirm_email'
    }

    , initialize: function()
    {
        Backbone.View.prototype.initialize.apply(this, arguments);
        BackboneFormView.add(this);
    }

    , saveFormCustom: function ()
    {
        this.new_email = this.$('[name="new_email"]').val();
        BackboneFormView.saveForm.apply(this, arguments);
    }

    , showSuccess: function (placeholder)
    {
        var global_view_message = new GlobalViewsMessageView({
            message: _('A confirmation email has been sent to <strong>').translate() + this.new_email + '</strong>',
            type: 'success',
            closable: true
        });

        placeholder.html(global_view_message.render().$el.html());
    }
);
});
});

```

Modify Profile/Javascript/Profile.Information.View.js

Replace

```

define(
    'Profile.Information.View'
, [
    'SC.Configuration',
    'GlobalViews.Message.View',
    'Backbone.FormView'

    , 'profile_information.tpl'

    , 'Backbone'
    , 'underscore'
    , 'jQuery'
    , 'Utils'
]
, function (
    Configuration
, GlobalViewsMessageView
, BackboneFormView

    , profile_information_tpl

    , Backbone
    ' -

```

```

    , jQuery
)
{
  'use strict';

  // @class Profile.Information.View @extends Backbone.View
  return Backbone.View.extend({

    template: profile_information_tpl
  , page_header: _('Profile Information').translate()
  , title: _('Profile Information').translate()
  , attributes: {'class': 'ProfileInformationView'}
  , events: {
      'submit form': 'saveForm'
    , 'change input[data-type="phone"]': 'formatPhone'
    }

  , bindings: {
      '[name="firstname)": "firstname"
    , '[name="lastname"]": "lastname"
    , '[name="companyname"]": "companyname"
    , '[name="phone"]": "phone"
    }

  , initialize: function()
  {
    BackboneFormView.add(this);
  }

  , formatPhone: function (e)
  {
    var $target = jQuery(e.target);
    $target.val(_.($target.val()).formatPhone());
  }

  , showSuccess: function ()
}

```

With

```

define(
  'Profile.Information.View'
, [
  'SC.Configuration'
, 'GlobalViews.Message.View'
, 'Backbone.FormView'

, 'Profile.ChangeEmailAddress.Model'
, 'Profile.ChangeEmailAddress.View'

, 'profile_information.tpl'

, 'Backbone'
, 'underscore'
, 'jQuery'
, 'Utils'

```

```

    ]
, function (
  Configuration
, GlobalViewsMessageView
, BackboneFormView

, ProfileChangeEmailModel
, ProfileChangeEmailView

, profile_information_tpl

, Backbone
'
-
, jQuery
)
{
'use strict';

// @class Profile.Information.View @extends Backbone.View
return Backbone.View.extend({

  template: profile_information_tpl
, page_header: _('Profile Information').translate()
, title: _('Profile Information').translate()
, attributes: {'class': 'ProfileInformationView'}
, events: {
    'submit form': 'saveForm'
, 'change input[data-type="phone"]': 'formatPhone'
, 'click [data-action="change-email"]': 'changeEmail'
  }

, bindings: {
    '[name="firstname)": "firstname"
, '[name="lastname"]": "lastname"
, '[name="companyname"]": "companyname"
, '[name="phone"]": "phone"
  }

, initialize: function(options)
{
  BackboneFormView.add(this);
  this.application = options.application;
}

, formatPhone: function (e)
{
  var $target = jQuery(e.target);
  $target.val(_.($target.val()).formatPhone());
}

, changeEmail: function ()
{
  var model = new ProfileChangeEmailModel(this.model.attributes);

  var view = new ProfileChangeEmailView({

```

```

        application: this.application
      , model: model
    });

    var self = this;

    model.on('save', function () {
      view.showSuccess(self.$('[data-type="alert-placeholder"]'));
    });

    view.useLayoutError = true;

    this.application.getLayout().showInModal(view);
  }

  , showSuccess: function ()

```

Add Profile/Sass/_profile-change-email.scss

```

.profile-change-email-button-back {
  @extend .button-back;
}

.profile-change-email-button-back-icon {
  @extend .button-back-icon;
}

.profile-change-email-form-label {
  display: inline-block;
}

.profile-change-email-form-group-label-required {
  @extend .input-required;
}

.profile-change-email {
  @extend .address-edit;
}

.profile-change-email-form-title {}
.profile-change-email-form-area {}

.profile-change-email-form {
  margin-top: $sc-base-margin * 3;
}

.profile-change-email-form-group,
.profile-change-email-form-actions {
  @extend .control-group;
}

.profile-change-email-form-group-label {
  @extend .input-label
}

```

```
.profile-change-email-form-group-label-required {
}

.profile-change-email-form-group-input {
    @extend .input-large
}

.profile-change-email-form-actions-change {
    @extend .button-primary;
    @extend .button-medium;
}

.profile-change-email-group-form-controls{}

.profile-change-email-form-info-block{}
```

Modify Profile/SuiteScript/Profile.Model.js

Replace

```
if (data.email && data.email !== this.currentSettings.email && data.email === data.confirm_email)
{
    if(data.isGuest === 'T')
    {
        customerUpdate.email = data.email;
    }
    else
    {
        login.changeEmail(data.current_password, data.email, true);
    }
}
```

With

```
if (data.email && data.email !== this.currentSettings.email && data.email === data.confirm_email && data.isGuest
    === 'T')
{
    customerUpdate.email = data.email;
}
else if (data.new_email && data.new_email === data.confirm_email && data.new_email !
== this.currentSettings.email)
{
    ModelsInit.session.login({
        email: data.email
        , password: data.current_password
    });
    login.changeEmail(data.current_password, data.new_email, true);
}
```

Add Profile/Templates/profile_change_email.tpl

```
<section class="profile-change-email">
```

```

<div data-type="alert-placeholder"></div>
<div class="profile-change-email-form-area">
  <form class="profile-change-email-form">
    <fieldset>
      <small class="profile-change-email-form-label">{{translate 'Required'}} <span class="profile-change-email-form-group-label-required">*</span></small>

      <div class="profile-change-email-form-group" data-input="new_email" data-validation="control-group">
        <label class="profile-change-email-form-group-label" for="new_email">{{translate 'New Email'}} <span class="profile-change-email-form-group-label-required">*</span></label>
        <div class="profile-change-email-group-form-controls" data-validation="control">
          <input type="email" class="profile-change-email-form-group-input" id="new_email" name="new_email" value="" placeholder="{{translate 'your@email.com'}}">
        </div>
      </div>

      <div class="profile-change-email-form-group" data-input="confirm_email" data-validation="control-group">
        <label class="profile-change-email-form-group-label" for="confirm_email">{{translate 'Confirm New Email'}} <span class="profile-change-email-form-group-label-required">*</span></label>
        <div class="profile-change-email-group-form-controls" data-validation="control">
          <input type="email" class="profile-change-email-form-group-input" id="confirm_email" name="confirm_email" value="" placeholder="{{translate 'your@email.com'}}">
        </div>
      </div>

      <div class="profile-change-email-form-group" data-input="current_email" data-validation="control-group">
        <label class="profile-change-email-form-group-label" for="current_password">{{translate 'Password'}} <span class="profile-change-email-form-group-label-required">*</span></label>
        <div class="profile-change-email-group-form-controls" data-validation="control">
          <input type="password" class="profile-change-email-form-group-input" id="current_password" name="current_password" value="">
        </div>
      </div>
    </fieldset>
    <p class="profile-change-email-form-info-block"><small> {{translate 'You will still be able to login with your current email address and password until your new email address is verified.'}} </small></p>
    <div class="profile-change-email-form-actions">
      <button type="submit" class="profile-change-email-form-actions-change">{{translate 'Send Verification Email'}}</button>
    </div>
  </form>
</div>
</section>

```

Modify Profile/Templates/profile_information.tpl

Replace

```

<p class="profile-information-input-email" id="email">
  {{email}}
</p>

```

With

```
<p class="profile-information-input-email" id="email">{{email}} | <a class="profile-information-change-email-address" data-action="change-email">{{translate 'Change Address'}}</a></p>
```

Duplicate Product Lists in Internet Explorer 11

ⓘ Applies to: SuiteCommerce Advanced | pre-Denali

In pre-Denali releases of SuiteCommerce Advanced, product lists in My Account are displayed in duplicate when using Internet Explorer 11. The duplication is a result of the caching behavior in IE 11 where product lists are cached by default. To correct this issue, modify the `ProductList.js` file as described here to force a new request to NetSuite and set the cache to `false` for product lists.

To modify the caching behavior defined in the `ProductList.js` file:

1. Copy the `ProductList.js` file at Reference Shopflow > js > src > app > modules > `ProductList` to the same location in the custom folder.
2. Modify the functions `getProductListsPromise` and `getProductList` with `{cache: false}` as shown in the following code snippet.

```
application.getProductListsPromise = function ()
{
  if (!application.productListsInstancePromise)
  {
    application.productListsInstancePromise = jQuery.Deferred();
    application.productListsInstance = new ProductListCollection();
    application.productListsInstance.application = application;

    // MODIFIED CODE FOR IE 11 CACHING
    application.productListsInstance.fetch({cache: false}).done(function(jsonCollection)
    {
      application.productListsInstance.set(jsonCollection);
      application.productListsInstancePromise.resolve(application.productListsInstance);
    });
  }

  return application.productListsInstancePromise;
};

// obtain a single ProductList with all its item's data
application.getProductList = function (id)
{
  var productList = new ProductListModel();
  productList.set('internalid', id);

  // MODIFIED CODE FOR IE 11 CACHING
  return productList.fetch({cache: false});
};
```

3. Save your changes.

Save for Later Item not Moved to Cart

ⓘ Applies to: SuiteCommerce Advanced | Mont Blanc

In Mont Blanc releases of SuiteCommerce Advanced, when users set an item as Save for Later and then return to move that item to the cart an error is returned. To correct this error, apply the patch described here to extend the `addItemToCartHandler()` method in the `ProductList.DetailsLater.View.js` file.

ⓘ Note: Before proceeding, familiarize yourself with the [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend the `ProductList.DetailsLater.View.js` File

This step explains how to extend the `ProductList.DetailsLater.View.js` file, which is located in the **ProductList** module. You can download the code samples described in this procedure here: [Product.List.Extension@1.0.0.zip](#)

1. If you have not done so already, create a directory to store your custom module.
2. Open this directory and create a subdirectory to maintain your customizations.

Give this directory a name similar to the module being customized. For example:

`Modules/extensions/Product.List.Extension@1.0.0`

3. In your new `Product.List.Extension@1.0.0` module, create a subdirectory called `JavaScript`.

`Modules/extensions/Product.List.Extension@1.0.0/JavaScript`

4. In your new `JavaScript` subdirectory, create a `JavaScript` file to extend `ProductList.DetailsLater.View.js`.

Name this file according to best practices. For example:

`ProductList.DetailsLater.View.Extension.js`

5. Open this file and extend the `addItemToCartHandler()` method as shown in the following code snippet.

```
define('ProductList.DetailsLater.View.Extension'
  [
    'ProductList.DetailsLater.View'
    , 'underscore'
  ]
  , function (
    ProductListDetailsView
  )
{
  'use strict';
  __.extend(ProductListDetailsView.prototype,
  {
    addItemToCartHandler : function (e)
    {
      e.stopPropagation();
      e.preventDefault();
      if (this.application.getConfig('addToCartBehavior') === 'showCartConfirmationModal')
      {
        this.cart.optimistic = null;
      }
    }
  });
}
```

```

var self = this
, selected_product_list_item_id = self.$(e.target).closest('article').data('id')
, selected_product_list_item = self.model.get('items').findWhere({
    internalid: selected_product_list_item_id.toString()
})
, selected_item = selected_product_list_item.get('item')
, selected_item_internalid = selected_item.internalid
, item_detail = selected_product_list_item.getItemForCart(selected_item_internalid, selected_product_list_item.get('quantity'), selected_item.itemoptions_detail, selected_product_list_item.getOptionsArra
y())
, add_to_cart.promise = this.addItemToCart(item_detail)
, whole.promise = jQuery.when(add_to_cart.promise, this.deleteListItem(selected_product_list_item))
.then(jQuery.proxy(this, 'executeAddToCartCallback'));

if (whole.promise)
{
    this.disableElementsOnPromise(whole.promise, 'article[data-item-id="' + selected_item_internalid
+ '"] a, article[data-item-id="' + selected_item_internalid + '"] button');
}
}

});

}
);

```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Extension

1. Open the Product.List.Extension@1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.
Modules/extensions/Product.List.Extension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Save the ns.package.json file.
5. Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

6. Add your custom module to the **modules** object.

This ensures that the Gulp tasks include your extension when you deploy. In this example, the **extensions/ProductList.DetailsLater.View.Extension** module is added at the beginning of the list of modules. However, you can add the module anywhere in the **modules** object. The order of precedence in this list does not matter.

```
{
```

```

    "name": "SuiteCommerce Advanced Elbrus",
    "version": "2.0",
    "buildToolsVersion": "1.3.0",
    "folders": {
        "modules": "Modules",
        "suitecommerceModules": "Modules/suitecommerce",
        "extensionsModules": "Modules/extensions",
        "thirdPartyModules": "Modules/third_parties",
        "distribution": "LocalDistribution",
        "deploy": "DeployDistribution"
    },
    "modules": {
        "extensions/ProductList.DetailsLater.View.Extension": "1.0.0",
        "suitecommerce/Account": "2.3.0",
        "suitecommerce/Address": "2.4.0",
        ...
    }
}

```

- Include the module definition ("ProductList.DetailsLater.View.Extension") in the dependencies array of the Checkout application JavaScript object.

Your distro.json file should look similar to the following:

```

"tasksConfig": {
//...
"javascript": [
//...
{
    "entryPoint": "SC.Checkout.Starter",
    "exportFile": "checkout.js",
    "dependencies": [
//...
        "CMSAdaptor",
        "Sensors",
        "ProductList.DetailsLater.View.Extension"
    ],
//...
}
]
}

```



Note: Best practice is to place any new modules at the bottom of the list in the `dependencies` array.

- Save the distro.json file.

Step 3: Test and Deploy Your Extension

- Deploy your customizations to your NetSuite account and test. See [Deploy to NetSuite](#).
- Verify your changes.

An error should not be returned when items set as Save for Later are later moved to the cart.

Running Gulp Commands Results in a Syntax Error

Applies to: SuiteCommerce Advanced | Denali | Mont Blanc | Vinson

This section applies to the **Vinson** release of SuiteCommerce Advanced and earlier.

Running any gulp commands from a command line or terminal can result in the following error:

```
SyntaxError: Invalid flags supplied to RegExp constructor 'u' at new RegExp  
(native)
```

To correct this error, edit the package.json file and reinstall Node.js dependencies for your top-level source files directory.



Note: This procedure requires editing the package.json code directly. You cannot override or extend this file.

Step 1: Edit the package.json File

This step explains how to add a new dependency to the package.json file, located in your top-level source code directory. You can download the code samples described in this procedure here: [package.json.zip](#).

1. Navigate to the top-level directory containing your SuiteCommerce Advanced source files.
2. Open the package.json file.
3. Add the following dependency to the **dependencies** object:

```
"xmlbuilder": "8.2.2"
```

Your edited code should look similar to the following:

```
{
  "name": "suitecommerce-builder",
  "version": "0.0.1",
  "description": "Sets of tasks to build a Reference Implementation",
  "main": "gulpfile.js",
  "private": true,
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    // ...
    "suitetalk": "file:./ns_npm_repository/suitetalk4node",
    "preconditions": "file:./ns_npm_repository/preconditions",
    "xmlbuilder": "8.2.2"
  },
  // ...
}
```

4. Save the file.

Step 2: Reinstall Dependencies and Test Your Changes

This step explains how to reinstall the Node.js dependencies required by the developer tools.

1. In your top-level source files directory, delete the **node_modules**/ subdirectory.
Deleting this subdirectory removes the dependencies and other files that were created when you initially set up your developer environment. See [Install Additional Files in the Source Directory](#) for more information on this process.
2. Open a command line or terminal.

3. Navigate to the top-level directory containing your SuiteCommerce Advanced source files.

4. Run the following command to reinstall the dependencies and include your edits:

```
npm install
```

5. Open a command line or terminal and navigate to your top-level source files directory.

6. Run Gulp.js using the following command:

```
gulp
```

This and any other gulp commands should run with no errors.

Missing Promo Code on Return Request

 **Applies to:** SuiteCommerce Advanced | Mont Blanc

This section applies to the **Mont Blanc** release of SuiteCommerce Advanced.

In Mont Blanc releases of SuiteCommerce Advanced, promo codes applied to the original sales order are not included in the calculations for a return request. To correct this error, apply the patch described here to extend the `setLines()` method in the `ReturnAuthorization.Model.js` file.



Note: Before proceeding, familiarize yourself with the [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend the `ReturnAuthorization.Model.js` File

This step explains how to extend the `ReturnAuthorization.Model.js` file, which is located in the **ReturnAuthorization** module. You can download the code samples described in this procedure here: [Return.Authorization.Extension@1.0.0.zip](#)

1. If you have not done so already, create a directory to store your custom module.

2. Open this directory and create a subdirectory to maintain your customizations.

Give this directory a name similar to the module being customized. For example:

`Modules/extensions/Return.Authorization.Extension@1.0.0`

3. In your new `Return.Authorization.Extension@1.0.0` module, create a subdirectory called `SuiteScript`.

`Modules/extensions/Return.Authorization.Extension@1.0.0/SuiteScript`

4. In your new `SuiteScript` subdirectory, create a `JavaScript` file to extend `ReturnAuthorization.Model.js`.

Name this file according to best practices. For example:

`ReturnAuthorization.Model.Extension.js`

5. Open this file and extend the `setLines()` method as shown in the following code snippet.

```
define('ReturnAuthorization.Model.Extension'
  [
    'ReturnAuthorization.Model'
    , 'underscore'
  ]
  , function (
    ReturnAuthorizationModel
```

```

' - 
)
{
'use strict';
_.extend(ReturnAuthorizationModel.prototype,
{

  setLines: function (return_authorization, lines, transaction_lines)
  {
    var line_count = return_authorization.getLineItemCount('item')
    ,   add_line = true
    ,   i = 1;

    while (i <= line_count)
    {
      var line_item_value = return_authorization.getLineItemValue('item', 'id', i);

      add_line = this.findLine(line_item_value, lines);

      if (add_line)
      {
        var transaction_line = _.findWhere(transaction_lines, { line: line_item_value });

        if (transaction_line)
        {
          return_authorization.setLineItemValue('item', 'rate', i, transaction_line.rate);
        }

        return_authorization.setLineItemValue('item', 'quantity', i, add_line.quantity);
        return_authorization.setLineItemValue('item', 'description', i, add_line.reason);
      }
      else
      {
        return_authorization.setLineItemValue('item', 'quantity', i, 0);
      }

      i++;
    }
  }
});

});
}
);

```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Extension

1. Open the Return.Authorization.Extension@1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.
Modules/extensions/Return.Authorization.Extension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code

```
{

```

```

    "gulp": {
      "ssp-libraries": [
        "SuiteScript/*.js"
      ]
    }
  }
}

```

4. Save the ns.package.json file.

5. Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

6. Add your custom module to the **modules** object.

This ensures that the Gulp tasks include your extension when you deploy. In this example, the **extensions/Return.Authorization.Extension** module is added at the beginning of the list of modules. However, you can add the module anywhere in the **modules** object. The order of precedence in this list does not matter.

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/Return.Authorization.Extension": "1.0.0",
    "suitecommerce/Account": "2.3.0",
    "suitecommerce/Address": "2.4.0",
    ...
  }
}
```

7. Include the module definition ("ReturnAuthorization.Model.Extension") in the dependencies array of the SCA application of the **ssp-libraries** object.

Your distro.json file should look similar to the following:

```
"tasksConfig": {
//...
  "ssp-libraries": {
    "entryPoint": "SCA",
    "dependencies": [
      "Application",
      "Account.Model",
      "Address.Model",
      //...
      "ReturnAuthorization.Model.Extension"
    ],
  }
}
```



Note: Best practice is to place any new modules at the bottom of the list in the **dependencies** array.

8. Save the distro.json file.

Step 3: Test and Deploy Your Extension

1. Deploy your customizations to your NetSuite account and test. See [Deploy to NetSuite](#).



Note: Since this patch modifies an SSP library file, changes are not visible in your local environment until you first deploy the customizations to NetSuite.

2. Verify your changes.

A returned order should include the original promo code that was used when the customer placed the order. The value on the sales order should match the value on the Return Authorization.

Enhanced Page Content Disappears when Resizing the Browser

Applies to: SuiteCommerce Advanced | Denali | Mont Blanc | Vinson | Elbrus

This section applies to the **Denali** release of SuiteCommerce Advanced and later.

Users might experience cases where enhanced pages do not maintain content when dynamically resizing the browser window. A typical scenario might include a user shrinking the browser window to such a point where the enhanced content disappears from view. Enlarging the browser window should cause any enhanced content to reinsert and appear. However, users might experience some cases where the enhanced content does not reappear in the browser.

To prevent this from happening, extend the ApplicationSkeleton.Layout.js and Content.js files as described in the steps below.

For more information on Enhanced Pages, see the help topic [Pages in SMT](#).



Note: Before proceeding, familiarize yourself with the [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend the ApplicationSkeleton.Layout.js File

This step explains how to extend the ApplicationSkeleton.Layout.js file, which is located in the **ApplicationSkeleton** module. You can download the code samples described in this procedure here: [ApplicationSkeletonExtension@1.0.0.zip](#)

1. If you have not done so already, create a directory to store your custom module. For example:
Modules/extensions
2. Open this directory and create a subdirectory to maintain your customizations.
Give this directory a name similar to the module being customized. For example:
Modules/extensions/ApplicationSkeletonExtension@1.0.0
3. In your new ApplicationSkeletonExtension@1.0.0 module, create a subdirectory called JavaScript.
Modules/extensions/ApplicationSkeletonExtension@1.0.0/JavaScript
4. In your new JavaScript subdirectory, create a JavaScript file to extend ApplicationSkeleton.Layout.js.
Name this file according to best practices. For example:

ApplicationSkeleton.Layout.Extension.js

5. Open this file and extend the `initialize()` method of the original module to include the following line:

```
self.trigger('resize', self.currentView);
```

Assuming that you have followed best practices outlined in this procedure, your extension should look like this:

```
define(
  'ApplicationSkeleton.Layout.Extension'
, [
  'ApplicationSkeleton.Layout'
, 'HeaderView'
, 'FooterView'
, 'Backbone.CompositeView'
, 'underscore'
, 'jQuery'
]
, function (
  ApplicationSkeletonLayout
, HeaderView
, FooterView
, BackboneCompositeView
, _
, jQuery
)
{
  'use strict';

  _.extend(ApplicationSkeletonLayout.prototype,
  {
    initialize: function (Application)
    {
      BackboneCompositeView.add(this);

      this.headerView = this.originalHeaderView = HeaderView;
      this.footerView = this.originalFooterView = FooterView;

      this.application = Application;
      this.windowWidth = jQuery(window).width();

      // @property {jQuery.Deferred} afterAppendViewPromise a promise that is resolve only if one
      // view was shown in this layout
      this.afterAppendViewPromise = jQuery.Deferred();

      var self = this;

      this.once('afterAppendView', function ()
      {
        self.afterAppendViewPromise.resolve();
      });

      jQuery(window).on('resize', _.throttle(function ()
```

```

    {

        if (_.getDeviceType(self.windowWidth) === _.getDeviceType(jQuery(window).width()))
        {
            return;
        }

        _.resetViewportWidth();

        self.updateHeader();
        self.updateFooter();

        self.trigger('resize', self.currentView);

        self.updateLayoutSB && self.updateLayoutSB();

        self.windowWidth = jQuery(window).width();

    }, 1000));
}
});
});

```

6. Save the file.

Step 2: Extend the Content.js File

This step explains how to extend the Content.js file, which is located in the **Content** module. The code required for your Content.js extension is the same for all versions of SCA. However, the original Content.js file differs slightly for each version of SCA. This procedure provides code examples for each version of SCA.

1. Download the appropriate code sample according to the implementation of SCA that you are customizing:

SCA Release	Download
Elbrus	ContentExtension@1.0.0---ElbrusSample.zip
Vinson	ContentExtension@1.0.0---VinsonSample.zip
Mont Blanc	ContentExtension@1.0.0---MontBlancSample.zip
Denali	ContentExtension@1.0.0---DenaliSample.zip

2. Open your custom extensions directory and create a subdirectory to maintain your customizations. Give this directory a name similar to the module being customized. For example:
Modules/extensions/ContentExtension@1.0.0
3. In your new ContentExtension@1.0.0 module, create a subdirectory called JavaScript.
Modules/extensions/ContentExtension@1.0.0/JavaScript
4. In your new JavaScript subdirectory, create a JavaScript file to extend Content.js. Name this file according to best practices. For example:
Content.Extension.js
5. Open this file and extend the **mountToApp()** method.
 - a. In your new **mountToApp()** method, search for the following line:

```
Layout.showInModal = _.wrap(Layout.showInModal, show_content_wrapper);
```

- b. Below this line, add the following:

```
Layout.on('resize', function(view)
{
    show_content_wrapper(function() { return jQuery.Deferred().resolve(); }, view);
});
```

6. Save the file.

Step 3: Prepare the Developer Tools for Your Customizations

1. Open the ApplicationSkeletonExtension@1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.
Modules/extensions/ApplicationSkeletonExtension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Save the ns.package.json file.
5. Repeat steps 2–4 for the ContentExtension@1.0.0 module. The two ns.package.json files should be identical.
6. Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

7. Add your custom module to the **modules** object to ensure that the Gulp tasks include your extension when you deploy.

In this example, the modules are added at the beginning of the list of modules. However, you can add the module anywhere in the **modules** object. The order of precedence in this list does not matter.



Note: The following example depicts the Distro.json file for Elbrus release. However, your customization of the **modules** object should look similar to what is depicted below.

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
```

```

    },
    "modules": {
        "extensions/ApplicationSkeletonExtension": "1.0.0",
        "extensions/ContentExtension": "1.0.0",
        "suitecommerce/Account": "2.2.0",
        "suitecommerce/Address": "2.3.0",
        ...
    }
}

```

- Include the file definition ("Content.Extension") in the dependencies array of the Shopping, Checkout, and MyAccount applications of the **JavaScript** object.

Your distro.json file should look similar to the following:

```

{
    "tasksConfig": {
        //...
        "javascript": [
            //...
            {
                "entryPoint": "SC.Shopping.Starter",
                "exportFile": "shopping.js",
                "dependencies": [
                    //...
                    "Newsletter",
                    "ProductDetailToQuote",
                    "Content.Extension"
                ],
                //...
                {
                    "entryPoint": "SC.MyAccount.Starter",
                    "exportFile": "myaccount.js",
                    "dependencies": [
                        //...
                        "Location.Model",
                        "StoreLocator.Model",
                        "Content.Extension"
                    ],
                    //...
                    {
                        "entryPoint": "SC.Checkout.Starter",
                        "exportFile": "checkout.js",
                        "dependencies": [
                            //...
                            "StoreLocatorAccessPoints",
                            "StoreLocator",
                            "Content.Extension"
                        ],
                        //...
                    }
                ]
            }
        ]
    }
}

```

- Save the distro.json file.

Step 4: Test and Deploy Your Extension

- Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
- Confirm your results.

After a successful deployment, dynamically resizing a browser window on a page with enhanced content results in no errors. The enhanced content appears on the page after enlarging the window to its original size.

Invoices Page Displays Incorrect Date Sort (pre-Denali)

ⓘ Applies to: SuiteCommerce Advanced | pre-Denali

This section applies to the **Reference My Account v1.05** bundle.

In Reference My Account v1.05, when shoppers sort invoices **By Due Date** or **By Invoice Date** on the Invoices page of the web store, the invoices display out of order. To correct the sort order, customize the `list()` method in the `Models.js` file as described in this section.

To customize the `Models.js` file:

1. If you have not done so already, copy the `Models.js` file from the reference folder to your custom folder in the NetSuite File Cabinet.
`Models.js` is located in the File Cabinet at `My Account 1.05 > Reference My Account > ssp_libraries`.
2. Update the `list()` method in the `Models.js` file from your custom folder to use the `nlapistringToDate()` method.

Find the following code:

```
return _.map(results || [], function (record)
{
    var due_date = record.getValue('duedate')
    , close_date = record.getValue('closedate')
    , tran_date = record.getValue('trandate')
    , due_in_milliseconds = new Date(due_date).getTime() - now
    , total = toCurrency(record.getValue(amount_field))
    , total_formatted = formatCurrency(record.getValue(amount_field));

    return {
        internalid: record.getId()
        , tranid: record.getValue('tranid')
        , order_number: record.getValue('tranid') // Legacy attribute
        , date: tran_date // Legacy attribute
        , summary: { // Legacy attribute
            total: total
            , total_formatted: total_formatted
        }
        , total: total
        , total_formatted: total_formatted
        , recordtype: record.getValue('type')
        , mainline: record.getValue('mainline')
        , amountremaining: toCurrency(record.getValue(amount_remaining))
        , amountremaining_formatted: formatCurrency(record.getValue(amount_remaining))
        , closedate: close_date
        , closedateInMilliseconds: new Date(close_date).getTime()
        , trandate: tran_date
        , tranDateInMilliseconds: new Date(tran_date).getTime()
    };
});
```

```

        , dueDate: due_date
        , dueInMilliseconds: due_in_milliseconds
        , isOverdue: due_in_milliseconds <= 0 && ((-1 * due_in_milliseconds) / 1000 / 60 / 60 / 24) >= 1
        , status: {
            internalId: record.getValue('status')
            , name: record.getText('status')
        }
        , currency: {
            internalId: record.getValue('currency')
            , name: record.getText('currency')
        }
    );
}
);

```

Replace these lines with the following code:

```

return _.map(results || [], function (record)
{
    var due_date = record.getValue('duedate')
    , close_date = record.getValue('closedate')
    , tran_date = record.getValue('trandate')
    , due_in_milliseconds = (!due_date ? nlapiStringToDate(due_date).getTime() :
(new Date()).getTime()) - now
    , total = toCurrency(record.getValue(amount_field))
    , total_formatted = formatCurrency(record.getValue(amount_field));
    return {
        internalid: record.getId()
        , tranid: record.getValue('tranid')
        , order_number: record.getValue('tranid') // Legacy attribute
        , date: tran_date // Legacy attribute
        , summary: { // Legacy attribute
            total: total
            , total_formatted: total_formatted
        }
        , total: total
        , total_formatted: total_formatted
        , recordtype: record.getValue('type')
        , mainline: record.getValue('mainline')
        , amountremaining: toCurrency(record.getValue(amount_remaining))
        , amountremaining_formatted: formatCurrency(record.getValue(amount_remaining))
        , closedate: close_date
        , closedateInMilliseconds: close_date ? nlapiStringToDate(close_date).getTime() : 0
        , trandate: tran_date
        , tranDateInMilliseconds: tran_date ? nlapiStringToDate(tran_date).getTime() : 0
        , dueDate: due_date
        , dueinmilliseconds: due_in_milliseconds
        , isOverdue: due_in_milliseconds <= 0 && ((-1 * due_in_milliseconds) / 1000 / 60 / 60 / 24) >= 1
        , status: {
            internalId: record.getValue('status')
            , name: record.getText('status')
        }
        , currency: {
            internalId: record.getValue('currency')
            , name: record.getText('currency')
        }
    }
});

```

```
    };
});
```

3. Configure your SSP Application to use the custom Models.js file.
 1. Open the My Account SSP application record at Setup > SuiteCommerce Advanced > SSP Applications.
 2. On the **Scripts** subtab, update the Libraries list to include the Models.js file from your custom folder.
This file must maintain the correct position at the bottom of the list of library files.
 3. Click **Save**.
4. Verify that the customization was successful.
The **By Due Date** and **By Invoice Date** sort order should be correct on the Invoices page in your web store. The invoices appear in ascending or descending order based on what the customer selects on the Invoice page in the web store.

PayPal Payments Cause Error at Checkout

 **Applies to:** SuiteCommerce Advanced | Mont Blanc

This section applies to the **Mont Blanc** release of SuiteCommerce Advanced.

In **Mont Blanc** releases of SuiteCommerce Advanced, customers may receive the following error message when placing orders using PayPal as the payment option:

Your order is below the minimum order amount of 0.

Although the error message is displayed, a sales order is created in NetSuite. To prevent this error message from displaying, extend the `past()` method as described in this procedure. Note that the error message is returned intermittently and is related to how PayPal returns sales amounts. You should apply this patch to avoid any cases of the error message being returned incorrectly.

 **Note:** Before proceeding, familiarize yourself with the [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend the OrderWizard.Module.PaymentMethod.PayPal.js File

This step explains how to extend the OrderWizard.Module.PaymentMethod.PayPal.js file, which is located in the **OrderWizard.Module.PaymentMethod** module. You can download the code samples described in this procedure here: [PaymentMethod.PayPal.Extension@1.0.0.zip](#)

1. If you have not done so already, create a directory to store your custom module.
2. Open this directory and create a subdirectory to maintain your customizations.
Give this directory a name similar to the module being customized. For example:
Modules/extensions/PaymentMethod.PayPal.Extension@1.0.0
3. In your new PaymentMethod.PayPal.Extension@1.0.0 module, create a subdirectory called JavaScript.
Modules/extensions/PaymentMethod.PayPal.Extension@1.0.0/JavaScript
4. In your new JavaScript subdirectory, create a JavaScript file to extend OrderWizard.Module.PaymentMethod.PayPal.js.

Name this file according to best practices. For example:

OrderWizard.Module.PaymentMethod.PayPal.Extension.js

5. Open this file and extend the **past()** method as shown in the following code snippet.

```
define('OrderWizard.Module.PaymentMethod.PayPal.Extension'
  [
    'OrderWizard.Module.PaymentMethod.PayPal'
    , 'underscore'
  ]
, function (
  OrderWizardModulePaymentMethodPayPal
)
{
  'use strict';
  _.extend(OrderWizardModulePaymentMethodPayPal.prototype,
  {
    past: function()
    {
      if (this.isActive() && !this.wizard.isPaypalComplete() && !this.wizard.hidePayment()
      && this.wizard.model.get('confirmation').isNew())
      {

        var checkout_url = Session.get('touchpoints.checkout')
        , joint = ~checkout_url.indexOf('?') ? '&' : '?'
        , previous_step_url = this.wizard.getPreviousStepUrl();

        checkout_url += joint + 'paypal=T&next_step=' + previous_step_url;

        Backbone.history.navigate(previous_step_url, {trigger: false, replace: true});

        document.location.href = checkout_url;

        throw new Error('This is not an error. This is just to abort javascript');
      }
    }
  });
});
```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Extension

1. Open the PaymentMethod.PayPal.Extension@1.0.0 module.

2. Create a file in this module and name it **ns.package.json**.

Modules/extensions/PaymentMethod.PayPal.Extension@1.0.0/ns.package.json

3. Build the ns.package.json file using the following code

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

```

        ]
    }
}

```

- Save the ns.package.json file.

- Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

- Add your custom module to the **modules** object.

This ensures that the Gulp tasks include your extension when you deploy. In this example, the **extensions/PaymentMethod.PayPal.Extension** module is added at the beginning of the list of modules. However, you can add the module anywhere in the **modules** object. The order of precedence in this list does not matter.

```
{
    "name": "SuiteCommerce Advanced Elbrus",
    "version": "2.0",
    "buildToolsVersion": "1.3.0",
    "folders": {
        "modules": "Modules",
        "suitecommerceModules": "Modules/suitecommerce",
        "extensionsModules": "Modules/extensions",
        "thirdPartyModules": "Modules/third_parties",
        "distribution": "LocalDistribution",
        "deploy": "DeployDistribution"
    },
    "modules": {
        "extensions/PaymentMethod.PayPal.Extension": "1.0.0",
        "suitecommerce/Account": "2.3.0",
        "suitecommerce/Address": "2.4.0",
        ...
    }
}
```

- Include the module definition ("OrderWizard.Module.PaymentMethod.PayPal") in the dependencies array of the Shopping application of the **JavaScript** object.

Your distro.json file should look similar to the following:

```
"tasksConfig": {
//...
"javascript": [
//...
{
    "entryPoint": "SC.Shopping.Starter",
    "exportFile": "shopping.js",
    "dependencies": [
//...
        "Newsletter",
        "ProductDetailToQuote",
//...
        "OrderWizard.Module.PaymentMethod.PayPal.Extension"
    ],
}
]
```



Note: Best practice is to place any new modules at the bottom of the list in the **dependencies** array.

- Save the distro.json file.

Step 3: Test and Deploy Your Extension

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)).

If you are currently running SCA on a local server, your changes should appear on your local site immediately.

2. Confirm your results.

An order placed in your web store using PayPal as the payment method should not display an error after the customer clicks **Place Order**. The customer should see an order confirmation and summary.

Canonical Tags Populated With Relative Paths

 **Applies to:** SuiteCommerce Advanced | Vinson

This section applies to the **Vinson** release of SuiteCommerce Advanced only.

In Vinson releases of SuiteCommerce Advanced, canonical URLs for commerce categories are generated as **relative** paths. Search engines do not index relative canonical URLs. Therefore, to ensure optimal SEO rankings, you must apply the patch described here to change your commerce category canonical URLs to use absolute paths. Best practices require customizing the existing source code using the `.extend` method, as detailed below.



Note: Before proceeding, familiarize yourself with the [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend the Facets.Browse.View.js File

This step explains how to extend the `Facets.Browse.View.js` file, which is located in the **Facets** module.

1. If you have not done so already, create a directory to store your custom module.
2. Open this directory and create a subdirectory to maintain your customizations.

Give this directory a name similar to the module being customized. For example:

`Modules/extensions/FacetsExtension@1.0.0`

3. In your new `FacetsExtension@1.0.0` module, create a subdirectory called `JavaScript`.

`Modules/extensions/FacetsExtension@1.0.0/JavaScript`

4. In your new `JavaScript` subdirectory, create a `JavaScript` file to extend `Facets.Browse.View.js`.

Name this file according to best practices. For example:

`Facets.Browse.View.Extension.js`

5. Open this file and extend the `getPath` method to modify the url as follows.

```
define('Facets.Browse.View.Extension'
, [
    'Facets.Browse.View'
, 'underscore'
]
, function (

```

```

FacetsBrowseView
'
  -
)
{
'use strict';
_.extend(FacetsBrowseView.prototype,
{
  getPath: function ()
  {
    var base_url = window.location.protocol + '//' + window.location.hostname;
    if (this.model.get('category'))
    {
      var category_canonical = this.model.get('category').get('canonical')
      || this.model.get('category').get('fullurl');
      return (category_canonical.indexOf('/') === 0 ? base_url : '') + category_canonical;
    }
    else
    {
      var canonical = base_url + '/' + Backbone.history.fragment
      , index_of_query = canonical.indexOf('?');
      // !~ means: indexOf == -1
      return !~index_of_query ? canonical : canonical.substring(0, index_of_query);
    }
  }
});
}
);

```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Extension

1. Open the FacetsExtension@1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.
Modules/extensions/FacetsExtension@1.0.0/ns.package.json
3. Build the ns.package.json file using the following code

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Save the ns.package.json file.
5. Open the distro.json file.
This file is located in the top-level directory of your SuiteCommerce Advanced source code.
6. Add your custom module to the **modules** object.

This ensures that the Gulp tasks include your extension when you deploy. In this example, the **extensions/FacetsExtension** module is added at the beginning of the list of modules.

However, you can add the module anywhere in the **modules** object. The order of precedence in this list does not matter.

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/FacetsExtension": "1.0.0",
    "suitecommerce/Account": "2.3.0",
    "suitecommerce/Address": "2.4.0",
    ...
  }
}
```

- Include the module definition ("Facets.Browse.View.Extension") in the dependencies array of the Shopping application of the `JavaScript` object.

Your distro.json file should look similar to the following:

```
"tasksConfig": {
//...
"javascript": [
//...
{
  "entryPoint": "SC.Shopping.Starter",
  "exportFile": "shopping.js",
  "dependencies": [
//...
  "Newsletter",
  "ProductDetailToQuote",
//..
  "Facets",
  "Facets.Browse.View.Extension"
]
},
//...
]
```



Note: Best practice is to place any new modules at the bottom of the list in the `dependencies` array. In this case, FacetsBrowseExtension must be placed after Facets as there are dependencies on that module.

- Save the distro.json file.

Step 3: Test and Deploy Your Extension

- Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)).

If you are currently running SCA on a local server, your changes should appear on your local site immediately.

- Confirm your results.

All commerce category canonical URLs have absolute paths.

Shopping Cart Not Scrolling (Mobile)

ⓘ Applies to: SuiteCommerce Advanced | Denali | Mont Blanc | Vinson

This section applies to the **Vinson** release of SuiteCommerce Advanced and earlier.

In some cases, mobile users encounter an issue where their Cart ceases to scroll after removing an item from the **Saved for Later** product list. This issue can occur in the following scenario:

- The user adds more than one item to their Cart.
- The user adds items to their **Saved for Later** product list.
- The user later removes an item from the **Saved for Later** product list and returns to their Cart.
- The user can no longer scroll their Cart.

If you experience this issue on your SCA site, perform the following steps to correct. Best practices require customizing the existing source code using the **.extend** method, as detailed below.



Note: Before proceeding, familiarize yourself with the [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend the ProductList.DetailsLater.View.js File

This step explains how to extend the `ProductList.DetailsLater.View.js` file, which is located in the **ProductList** module.

1. If you have not done so already, create a directory to store your custom module.
2. Open this directory and create a subdirectory to maintain your customizations.

Give this directory a name similar to the module being customized. For example:

`Modules/extensions/ProductListExtension@1.0.0`

3. In your new `ProductListExtension@1.0.0` module, create a subdirectory called `JavaScript`.

`Modules/extensions/ProductListExtension@1.0.0/JavaScript`

4. In your new `JavaScript` subdirectory, create a `JavaScript` file to extend `ProductList.DetailsLater.View.js`.

Name this file according to best practices. For example:

`ProductList.DetailsLater.View.Extension.js`

5. Open this file and extend the `deleteListItemHandler` method to include the following line:

```
self.$('[data-action="pushable"]').scPush();
```

Assuming that you have followed best practices outlined in this procedure, your extension should look like this:

```
define('ProductList.DetailsLater.View.Extension'
, [
    'ProductList.DetailsLater.View'
, 'underscore'
, 'jQuery'
```

```

    ]
  , function (
    ProductListDetailsLaterView
  ,
  -
  , jQuery
  )
{
  'use strict';

  _.extend(ProductListDetailsLaterView.prototype,
  {
    deleteListItemHandler: function (target)
    {
      var self = this
      , itemid = jQuery(target).closest('article').data('id')
      , product_list_item = this.model.get('items').findWhere({
        internalid: itemid + ''
      })
      , success = function ()
      {
        if (self.application.getLayout().updateMenuItemsUI)
        {
          self.application.getLayout().updateMenuItemsUI();
        }

        self.deleteConfirmationView.$containerModal.modal('hide');
        self.render();
        self.$('[data-action="pushable"]').scPush();
        self.showConfirmationMessage(_('The item was removed from your product
list')).translate(), true);
      };

      self.model.get('items').remove(product_list_item);
      self.deleteListItem(product_list_item, success);
    }
  });
}
);

```

6. Save the file.

Step 2: Download the jQuery.scPush.js File

1. Download the following file:

[jQuery.scPush.zip](#)

2. Open the .zip file and extract jQuery.scPush.js to the JavaScript subdirectory of your custom module.

`Modules/extensions/ProductListExtension@1.0.0/JavaScript/jQuery.scPush.js`

Step 3: Prepare the Developer Tools for Your Customizations

1. Open the ProductListExtension@1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.

Modules/extensions/ProductListExtension@1.0.0/ns.package.json

3. Build the ns.package.json file using the following code

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
  "overrides": {
    "Modules/suitecommerce/jQueryExtras@x.y.z/JavaScript/jQuery.scPush.js": "JavaScript/jQuery.scPush.js"
  }
}
```



Note: In the above code example, replace the string x.y.z with the current version of your source module.

4. Save the ns.package.json file.

5. Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

6. Add your custom module to the `modules` object to ensure that the Gulp tasks include your extension when you deploy.

In this example, the `extensions/ProductListExtension` module is added at the beginning of the list of modules. However, you can add the module anywhere in the `modules` object. The order of precedence in this list does not matter.

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/ProductListExtension": "1.0.0",
    "suitecommerce/Account": "2.3.0",
    "suitecommerce/Address": "2.4.0",
    ...
  }
}
```

7. Include the file definition ("`ProductList.DetailsLater.View.Extension`") in the dependencies array of the Shopping and MyAccount applications of the `JavaScript` object.

Your distro.json file should look similar to the following:

```
"tasksConfig": {
  //...
  "javascript": [
    //...
  ]
}
```

```
{
  "entryPoint": "SC.Shopping.Starter",
  "exportFile": "shopping.js",
  "dependencies": [
    //...
    "Newsletter",
    "ProductDetailToQuote",
    "ProductList.DetailsLater.View.Extension"
  ],
  //...
  {
    "entryPoint": "SC.MyAccount.Starter",
    "exportFile": "myaccount.js",
    "dependencies": [
      //...
      "Location.Model",
      "StoreLocator.Model",
      "ProductList.DetailsLater.View.Extension"
    ],
    //...
  }
}
```



Note: Best practice is to place any new modules at the bottom of the list in the `dependencies` array.

8. Save the distro.json file.

Step 4: Test and Deploy Your Extension

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.
2. Confirm your results.

Upon successful deployment, you can remove items from the **Save for Later** list, return to the Cart, and scroll through your list of items using a mobile interface.

Error When Adding Items to Categories in Site Management Tools

Applies to: SuiteCommerce Advanced | Vinson

This patch contains a fix for an error on a category or subcategory that contains more than 10 items in Site Management Tools. SMT generates an error if you try to add more items to a category or subcategory that already contains more than 10 items. This error occurs because the default query sent to Items API to retrieve the items includes facets and the Items API returns no more than 10 results for queries that include facets.

This patch adds a Search API fieldset named **CmsAdapterSearch**. To install the patch, you must override files in the **Configuration**, **Facets**, and **CMSAdapter** modules and add the new **CmsAdapterSearch** fieldset in the SuiteCommerce Advanced configuration. For an example of the changes needed for this patch, see [AddingItemstoCategoriesinSMT.zip](#).



Note: In general, NetSuite best practice is to extend JavaScript using the JavaScript `prototype` object. This improves the chances that your customizations continue to work when migrating to a newer version of SuiteCommerce Advanced. However, this patch requires you to modify files that you cannot extend, and therefore requires you to override the existing module files in a custom module. For more information, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1. Override the ItemsSearchAPI.json File

This step shows how to override the `ItemsSearchAPI.json` file, located in the **Configuration** module. For more information about module configuration with JSON, see [Modify JSON Configuration Files](#).

1. If you have not done so already, create a directory to store your custom modules, for example, `extensions`.

2. Open this directory and create a subdirectory to maintain your customizations.

Give this directory a name similar to the module being customized. For example:

`Modules/extensions/ConfigurationExtension@1.0.0`

3. In your new `ConfigurationExtension@1.0.0` directory, create a subdirectory called `Configuration`.

4. Copy the following file into this directory:

`Modules/suitecommerce/Configuration@1.0.0/Configuration/ItemsSearchAPI.json`

5. Open `ItemsSearchAPI.json` and make the changes in steps 6 and 7.

6. In `ItemsSearchAPI.json`, replace the following line:

```
"enum": ["Facets", "itemDetails", "relatedItems", "correlatedItems", "merchandisingZone", "typeAhead", "itemsSearcher"],
```

With the following JSON object:

```
"enum": ["Facets", "itemDetails", "relatedItems", "correlatedItems", "merchandisingZone", "typeAhead", "itemsSearcher", "CmsAdapterSearch"],
```

7. In `ItemsSearchAPI.json`, add the following object to the `default` object:

```
{ "id": "CmsAdapterSearch", "fieldset": "search" }
```

The `default` object should look similar to the following code after this step:

```
"default": [
    {
        "id": "Facets",
        "fieldset": "search",
        "include": "facets"
    },
    ...
    {
        "id": "CmsAdapterSearch",
        "fieldset": "search"
    }
]
```

8. Save the file.

Step 2. Override the `Facets.Model.js` File

This step shows how to override the `Facets.Model.js` file, located in the **Facets** module.

1. Create a directory in the `extensions` directory to store the custom module.
Give this directory a name similar to the module being customized. For example:
`Modules/extensions/FacetsExtension@1.0.0`
2. In the `FacetsExtension@1.0.0` module, create a subdirectory called `JavaScript`.
3. Copy the following file into this directory:
`Modules/suitecommerce/Facets@2.3.0/JavaScript/Facets.Model.js`
4. Open `Facets.Model.js` and make the following change.
Replace the existing `initialize` function:

```
initialize: function ()
{
    // Listen to the change event of the items and converts it to an ItemDetailsCollection
    this.on('change:items', function (model, items)
    {
        if (!(items instanceof ItemDetailsCollection))
        {
            // NOTE: Compact is used to filter null values from response
            model.set('items', new ItemDetailsCollection(_.compact(items)));
        }
    });
}
```

With the following JavaScript code:

```
initialize: function ()
{
    if (options && options.searchApiMasterOptions)
    {
        this.searchApiMasterOptions = options.searchApiMasterOptions;
    }

    // Listen to the change event of the items and converts it to an ItemDetailsCollection
    this.on('change:items', function (model, items)
    {
        if (!(items instanceof ItemDetailsCollection))
        {
            // NOTE: Compact is used to filter null values from response
            model.set('items', new ItemDetailsCollection(_.compact(items)));
        }
    });
}
```

5. Save the file.

Step 3. Override the `CMSadapterImpl.Categories.js` File

This step shows how to override the `CMSadapterImpl.Categories.js` file, located in the **CMSadapter** module.

1. Create a directory in the `extensions` directory to store the custom module.

Give this directory a name similar to the module being customized. For example:

`Modules/extensions/CMSadapterExtension@1.0.0`

2. In the `CMSadapterExtension@1.0.0` module directory, create a subdirectory called `JavaScript`.

3. Copy the following file into this directory:

`Modules/suitecommerce/CMSadapter@3.0.0/JavaScript/
CMSadapterImpl.Categories.js`

4. Open `CMSadapterImpl.Categories.js` and make the changes in steps 5 to 7.

5. In `CMSadapterImpl.Categories.js`, add '`SC.Configuration`' to the list of dependencies. For example:

```
define('CMSadapterImpl.Categories'
  [
    'Facets.Model'
    ...
    , 'SC.Configuration'
  ])
```

6. In `CMSadapterImpl.Categories.js`, add the `Configuration` function parameter. For example:

```
, function (
  FacetsModel
  ...
  , Configuration
)
```

7. In `CMSadapterImpl.Categories.js`, replace the following line:

```
var model = new FacetsModel();
```

With the following JavaScript code:

```
var model = new FacetsModel({ searchApiMasterOptions:  
  Configuration.get('searchApiMasterOptions.CmsAdapterSearch') });
```

8. Save the file.

Step 4. Prepare the Developer Tools for Your Overrides

This step shows how to set up the `ns.package.json` files for your custom module and modify `distro.json` to make sure that the Gulp tasks include your modules when you deploy.

1. Create the `ns.package.json` file in the `ConfigurationExtension@X.X.X` directory. Add the following code to `ns.package.json` in the `Modules/extensions/ConfigurationExtension@X.X.X` directory:

```
{
  "gulp": {
    "configuration": [
      "Configuration/*.json"
    ]
  }
}
```

```
}
```

2. Create the `ns.package.json` file in the `FacetsExtension@X.X.X` directory. Add the following code to `ns.package.json` in the `Modules/extensions/FacetsExtension@X.X.X` directory:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*"
    ]
  }
}
```

3. Create the `ns.package.json` file in the `CMSadapterExtension@X.X.X` directory. Add the following code to `ns.package.json` in the `Modules/extensions/CMSadapterExtension@X.X.X` directory:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*"
    ]
  }
}
```

4. In `distro.json`, add your custom modules to the `modules` object.

This ensures that the Gulp tasks include your module when you deploy. In this example, the custom modules are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

```
{
  "name": "SuiteCommerce Advanced Vinson Release",
  "version": "2.0",
  "buildToolsVersion": "1.2.1",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/ConfigurationExtension": "X.X.X",
    "extensions/FacetsExtension": "X.X.X",
    "extensions/CMSadapterExtension": "X.X.X",
  }
}
```

Step 5. Deploy Your Override

- To test this customization, you must first deploy it to your NetSuite account. You may need to modify the SuiteCommerce Advanced configuration, therefore you must deploy it first. See [Deploy to NetSuite](#).
- After you deploy the customization, log in to NetSuite and go to Setup > SuiteCommerce Advanced > Configuration, and select the appropriate web site and domain.
- On the **Advanced > Search Results** tab, make sure that **CmsAdapterSearch** appears for the search fieldset. If it does not appear, you must add it and click **Save**.

You can now add more than 10 items to a category in Site Builder.

Item Search API Response Data not Cached

ⓘ Applies to: SuiteCommerce Advanced | Vinson

This section applies to the **Vinson** release of SuiteCommerce Advanced.

In **Vinson** releases of SuiteCommerce Advanced, the Item Search API response data is not cached in the Content Delivery Network (CDN) by default. To enable caching of the Item Search API response, you must customize your implementation to include the **pricelvel** input parameter in the Item Search API query. Caching response data in the CDN is critical as it decreases Item Search API response time and improves application performance.



Note: Before proceeding, familiarize yourself with the [Best Practices for Customizing SuiteCommerce Advanced](#).

Step 1: Extend the Session.js File

This step explains how to extend the Session.js file, which is located in the **Session** module. You can download the code samples described in this procedure here: [SearchApiCdnCache.zip](#).

1. If you have not done so already, create a directory to store your custom module.
2. Open this directory and create a subdirectory to maintain your customizations.

Give this directory a name similar to the module being customized. For example:

Modules/extensions/SessionExtension@1.0.0

3. In your new SessionExtension@1.0.0 module, create a subdirectory called JavaScript.

Modules/extensions/SessionExtension@1.0.0/JavaScript

4. In your new JavaScript subdirectory, create a JavaScript file to extend Session.js.

Name this file according to best practices. For example:

Session.Extension.js

5. Open this file and extend the **getSearchApiParams()** method to add the **pricelvel** parameter to the Item Search API GET request.

```
// Session.Fix.js
// When Session.getSearchApiParams is called, this method adds the pricelvel param to the output
define('Session.Extension'
,
[
    'Session'
]
,
function (Session)
{
    'use strict';
    var original_getSearchApiParams = Session.getSearchApiParams;
    Session.getSearchApiParams = function ()
    {
        var search_api_params = original_getSearchApiParams.apply(this, arguments);
        search_api_params.pricelvel = this.get('priceLevel', '');
        return search_api_params;
    }
}
)
```

```
});
```

6. Save the file.

Step 2: Prepare the Developer Tools for Your Extension

1. Open the SessionExtension@1.0.0 module.
2. Create a file in this module and name it **ns.package.json**.
`Modules/extensions/SessionExtension@1.0.0/ns.package.json`
3. Build the ns.package.json file using the following code

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ]
  }
}
```

4. Save the ns.package.json file.

5. Open the distro.json file.

This file is located in the top-level directory of your SuiteCommerce Advanced source code.

6. Add your custom module to the **modules** object.

This ensures that the Gulp tasks include your extension when you deploy. In this example, the **extensions/SessionExtension** module is added at the beginning of the list of modules. However, you can add the module anywhere in the **modules** object. The order of precedence in this list does not matter.

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
    "extensions/SessionExtension": "1.0.0",
    "suitecommerce/Account": "2.3.0",
    "suitecommerce/Address": "2.4.0",
    ...
  }
}
```

7. Include the module definition ("Session.Extension") in the dependencies array of the Shopping application of the **JavaScript** object.

Your distro.json file should look similar to the following:

```
"tasksConfig": {
//...
"javascript": [
```

```
//...
{
    "entryPoint": "SC.Shopping.Starter",
    "exportFile": "shopping.js",
    "dependencies": [
        //...
        "Newsletter",
        "ProductDetailToQuote",
        //...
        "Session.Extension"
    ],
}
```



Note: Best practice is to place any new modules at the bottom of the list in the `dependencies` array.

8. Save the distro.json file.

Step 3: Test and Deploy Your Extension

You are now ready to test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)). If you are currently running SCA on a local server, your changes should appear on your local site immediately.

To verify that the Item Search API output response can now be cached in the CDN:

1. Navigate to the Product Display Page in your web store.
2. Open the browser's Developer Tools.
3. View the Network XHR response data.
4. Refresh your web store page.
5. Verify that the items query contains the `pricellevel` parameter.

Secure Shopping Domains (Elbrus, Vinson, Mont Blanc, and Denali)

Applies to: SuiteCommerce Advanced | Denali | Mont Blanc | Vinson | Elbrus

With the Elbrus release of SuiteCommerce Advanced, a single secure (HTTPS) domain for the checkout and shopping applications is supported. In order to provide this capability to prior releases of SuiteCommerce Advanced, you need to update your implementation by applying the patches provided here. For Elbrus implementations, you also need to apply the provided patch to fix a `ERR_WS_REQUIRE_CUSTOMER_LOGIN` error returned from the `setPurchaseNumber` method.

To update SuiteCommerce Advanced to use SSL correctly:

1. Create a backup of your existing bundle.
2. Download the patch according to your implementation:
 - [Elbrus-ssl.patch](#)
 - [Vinson-ssl.patch](#)
 - [MontBlanc-ssl.patch](#)
 - [Denali-ssl.patch](#)

3. Apply the patch. Details of what the patch is accomplishing are provided below.



Note: The original patches provided for Vinson, Mont Blanc, and Denali have been updated. If you have previously applied a patch for SSL, only the diff between the previous patch and the current one should be applied.

4. Switch the existing non-secure shopping domain to a secure shopping domain or set up a new secure shopping domain from scratch as described under [Secure Web Store](#).

For each implementation patch, instead of using `url.indexOf('https')` to check the domain, multiple methods have been combined and centralized in the `Utils.js` file. The following properties are used to determine which domain is in use:

Utils Property	Domain In Use
<code>Utils.isShoppingDomain()</code>	Shopping domain
<code>Utils.isCheckoutDomain()</code>	Checkout domain
<code>Utils.isSingleDomain()</code>	Single domain
<code>Utils isInShopping()</code>	<code>shopping.ssp</code>
<code>Utils isInCheckout()</code>	<code>checkout.ssp</code> or <code>myaccount.ssp</code>

For the Denali implementation patch, NetSuite also maintains access control by replacing `session.isLoggedIn` with `session.IsLoggedIn2`. In addition, the Denali patch takes advantage of the Serversync touchpoint patch to maintain the identity of the user and cart contents.

Secure Shopping Domain (pre-Denali)

Applies to: SuiteCommerce Advanced | pre-Denali

This section applies to **pre-Denali** releases of SuiteCommerce Advanced.

A secure shopping domain creates an encrypted connection between your web server and your customer's web browser. You can use an SSL certificate to secure the shopping portion of your web store under an HTTPS domain. HTTPS is currently the industry standard for ecommerce services and consumers prefer seeing the secure icon in their browser address bar.

Including secure technology in your shopping area assures your customers that their activities on your site are safe. In addition, search engines tend to rank secure sites higher than non-secure sites.

There are two new SuiteCommerce API methods that help make HTTPS support possible. These methods distinguish between the checkout, shopping, and single domains.

- `isCheckoutSupported()`: returns true if we are in the checkout domain or in a single domain
- `isShoppingSupported()`: returns true if we are in the shopping domain or in a single domain

`!~request.getURL().indexOf('https')` is the legacy domain check and it will no longer work to distinguish between the domains.

The `session.isLoggedIn()` method is the legacy session check for a user's logged in status. This method has been replaced with the new `session.IsLoggedIn2()` method in later versions of SuiteCommerce Advanced. The new session check method is required in this migration to help maintain the identity of the user.

To leverage a secure shopping domain in a pre-Denali version of SuiteCommerce Advanced, you need to:

- Replace all instances of the legacy domain check with the new domain check methods.
- Replace all instances of the legacy session check with the new session check method.

Migration Tasks

This migration is highly technical and involves customizing multiple files. Update the following files in each SSP Application to use the new domain check methods as well as the new session check method where applicable:

- commons.js
- Utils.js
- Models.js
- sc.environment.ssp
- index.ssp and index-local.ssp
- jQuery.ajaxSetup.js
- live-order.ss and live-order-line.ss
- CheckoutSkipLogin.js
- NavigationHelper.js

In addition, complete these tasks.

- Update Legacy Checks and Methods
- Deploy the Migration



Important: This document is intended to be a guide only. It is not a comprehensive procedure that guarantees a correct implementation of HTTPS support for your web store.

To modify files for HTTPS:



Important: Before you begin making changes, back up any custom files that already exist. Be careful not to overwrite existing custom files.

For each file, go to the SSP Application under Documents > Files > File Cabinet > Web Site Hosting Files > Live Hosting Files > SSP Applications and copy the Reference files to the same location in the Custom folder.

1. Open each file at the specified location in the Custom folder.
2. Search each file for the appropriate location using the line number or surrounding text from the code snippet.

Use the following conventions to determine when you need to add or remove code.

```
--- = Remove this line of code.  
+++ = Add this line of code.
```

Note: Surrounding text and line numbers vary depending on each SSP Application as well as existing customizations. They are included as reference points only.

3. Add or remove statements as indicated in the code snippet and save.

commons.js

File locations:

- ShopFlow > Custom ShopFlow > ssp_libraries > commons.js
- MyAccount > Custom MyAccount > ssp_libraries > commons.js
- Checkout > Custom Checkout > ssp_libraries > commons.js

Add the entire **Utils** variable to the Utilities section of each **commons.js** file. The **Utils** variable contains the new methods that distinguish between the domains.

```

...
// Utilities
+++ var Utils = {
+++
+++     // @method isCheckoutDomain determines if we are in a secure checkout
+++     // domain or in a secure single domain environment
+++     // @return {Boolean} true if in checkout or in single domain
+++ , isCheckoutDomain: function isCheckoutDomain()
+++
+++     {
+++         return session.isCheckoutSupported();
+++     }
+++     // @method isShoppingDomain determines if we are in shopping domain (secure or non secure)
+++     // or in a secure single domain environment
+++     // @return {Boolean} true if in shopping or single domain
+++ , isShoppingDomain: function isShoppingDomain()
+++
+++     {
+++         return session.isShoppingSupported();
+++     }
+++     // @method isSingleDomain determines if we are in a single domain environment
+++     // @return {Boolean} true if single domain
+++ , isSingleDomain: function isSingleDomain()
+++
+++     {
+++         return this.isShoppingDomain() && this.isCheckoutDomain();
+++     }
+++     // @method isInShopping determines if we are in shopping ssp
+++     // @return {Boolean} true if in shopping domain, false if in checkout or myaccount
+++ , isInShopping: function isInShopping (request)
+++
+++     {
+++         return this.isShoppingDomain() && (request.getHeader('X-SC-Touchpoint') === 'shopping'
|| request.getParameter('X-SC-Touchpoint') === 'shopping');
+++     }
+++     // @method isInCheckout determines if we are in checkout ssp or my account ssp
+++     // @return {Boolean} true if in checkout domain
+++ , isInCheckout: function isInCheckout (request)
+++
+++     {
+++         var self = this;
+++
+++         if (!self.isSingleDomain())
+++         {
+++             return self.isCheckoutDomain();
+++         }
+++         else
+++         {
+++             var paypal_complete = ModelsInit.context.getSessionObject('paypal_complete') === 'T'
+++             , is_in_checkout = request.getHeader('X-SC-Touchpoint') === 'checkout' ||
+++                 request.getHeader('X-SC-Touchpoint') === 'myaccount' ||
+++                 request.getParameter('X-SC-Touchpoint') === 'checkout' ||
+++                 request.getParameter('X-SC-Touchpoint') === 'myaccount'
+++         }
+++     }
+++ }
+++ 
```

```

+++             request.getParameter('X-SC-Touchpoint') === 'myaccount';
+++         return self.isCheckoutDomain() && (is_in_checkout || paypal_complete);
+++     }
+++ }
...

```

Call the new `session.isLoggedIn2()` method to maintain user identity.

```

...
999:   function isLoggedIn ()
{
    'use strict';
--- // MyAccount (We need to make the following difference because isLoggedIn is always false in
Shopping)
--- if (request.getURL().indexOf('https') === 0)
---
--- {
---     return session.isLoggedIn();
--- }
--- else // Shopping
--- {
---     return parseInt(nlapiGetUser() + '', 10) > 0 && !session.getCustomer().isGuest();
--- }
...
+++ return session.isLoggedIn2();
}
...

```

Modify the `getEnvironment()` method to add the new domain check methods.

```

...
, init: function () {}
220: , getEnvironment: function (session, request)
{
    'use strict';
    // Sets Default environment variables
    var context = nlapiGetContext()
--- , isSecure = request.getURL().indexOf('https:') === 0;
+++ , isSecure = Utils.isCheckoutDomain();

    , siteSettings = session.getSiteSettings(['currencies', 'languages'])
    , result = {
---         baseUrl: session.getAbsoluteUrl(isSecure ? 'checkout' : 'shopping', '/{{file}}')
+++         baseUrl: session.getAbsoluteUrl(Utils.isInCheckout(request) ? 'checkout'
: 'shopping', '/{{file}}')
            , currentHostString: request.getURL().match('http(s)?://(.*)/')[2]
            , availableHosts: SC.Configuration.hosts || []
            , availableLanguages: siteSettings.languages || []
            , availableCurrencies: siteSettings.currencies || []
            , companyId: context.getCompany()
            , casesManagementEnabled: context.getSetting('FEATURE', 'SUPPORT') === 'T'
            , giftCertificatesEnabled: context.getSetting('FEATURE', 'GIFTCERTIFICATES') === 'T'
    };
}

```

```

        // If there are hosts associated in the site we iterate them to check which we are in
        // and which language and currency we are in
---  if (result.availableHosts.length && !isSecure)
+++  if (result.availableHosts.length && Utils.isShoppingDomain())
{
    for (var i = 0; i < result.availableHosts.length; i++)
...

```

Utils.js

File locations:

- ShopFlow > Custom ShopFlow > js > src > core > Utils.js
- MyAccount > Custom MyAccount > js > src > core > Utils.js
- Checkout > Custom Checkout > js > src > core > Utils.js

Update the `getDownloadPdfUrl()` method to add the new domain check methods.

```

...
635:  function getDownloadPdfUrl (params)
{
---  params = params || {};
---  params.n = SC.ENVIRONMENT.siteSettings.siteid;
---  var origin = window.location.origin ? window.location.origin :
---      (window.location.protocol + '/' + window.location.hostname + (window.location.port ? ':' +
window.location.port) : '');
---  return _.addParamsToUrl(origin + _.getAbsoluteUrl('download.ssp'), params);

+++  params = params || {};
+++  params.n = SC && SC.ENVIRONMENT && SC.ENVIRONMENT.siteSettings && SC.ENVIRONMENT.siteSettings.siteid
|| '';
+++
+++  if(_.isSingleDomain())
+++
{   return _.addParamsToUrl(_.getAbsoluteUrl('download.ssp'), params);
+++
}
+++
else
+++
{
+++
    var origin = window.location.origin ? window.location.origin :
+++
        (window.location.protocol + '/' + window.location.hostname + (window.location.port ? ':' +
window.location.port) : '');
+++
    return _.addParamsToUrl(origin + _.getAbsoluteUrl('download.ssp'), params);
+++
}

}
...
797:  function reorderUrlParams (url)
{
    var params = []
    , url_array = url.split('?');

```

```

        if (url_array.length > 1)
        {
            params = url_array[1].split('&');
            return url_array[0] + '?' + params.sort().join('&');
        }
        return url_array[0];
    }

    // @method isShoppingDomain determines if we are in shopping domain (secure or non secure)
    // or single domain
    // @return {Boolean} true if in checkout or in single domain
    function isShoppingDomain ()
    {
        return SC.ENVIRONMENT.siteSettings.shoppingSupported;
    }

    // @method isCheckoutDomain determines if we are in a secure checkout
    // domain or in a secure single domain environment
    // @return {Boolean} true if in checkout or in single domain
    function isCheckoutDomain ()
    {
        return SC.ENVIRONMENT.siteSettings.checkoutSupported;
    }

    // @method isSingleDomain determines if we are in a single domain environment
    // @return {Boolean} true if single domain
    function isSingleDomain ()
    {
        return SC.ENVIRONMENT.siteSettings.isSingleDomain;
    }

    // @method isInShopping determines if we are in shopping ssp
    // used when there are frontend features only shown in the shopping domain
    // @return {Boolean} true if in shopping domain, false if in checkout or myaccount
    function isInShopping ()
    {
        return _.isShoppingDomain() && (SC.ENVIRONMENT.SCTouchpoint === 'shopping' || SC.ENVIRONMENT.siteSettings.sitetype === 'STANDARD');
    }

    // @method isInCheckout determines if we are in checkout or my account ssp
    // @return {Boolean} true if in checkout domain
    function isInCheckout ()
    {
        return !_.isSingleDomain() ? _.isCheckoutDomain() : _.isCheckoutDomain() && (SC.ENVIRONMENT.SCTouchpoint === 'checkout' || SC.ENVIRONMENT.SCTouchpoint === 'myaccount');
    }

    ...

```

Models.js

File locations:

- ShopFlow > Custom ShopFlow > ssp_libraries >Models.js

- MyAccount > Custom MyAccount > ssp_libraries > Models.js
- Checkout > Custom Checkout > ssp_libraries > Models.js

Verify the domain using the new domain check methods.

```
...
19:     Application.defineModel('SiteSettings', {
...
121:     settings.is_logged_in = session.isLoggedIn();
        settings.touchpoints = basic_settings.touchpoints;
        settings.shopperCurrency = session.getShopperCurrency();

...
+++     settings.checkoutSupported = Utils.isCheckoutDomain();
+++     settings.shoppingSupported = Utils.isShoppingDomain();
+++     settings.isSingleDomain = settings.checkoutSupported && settings.shoppingSupported;

...
}
```

Call the new `session.isLoggedIn2()` method and update the `getFieldValues()` method to use the new domain check methods.

```
...
167:     Application.defineModel('LiveOrder', {
...
309:     --- , isSecure: request.getURL().indexOf('https') === 0
        --- , isLoggedIn: session.isLoggedIn()

        +++ , isLoggedIn: session.isLoggedIn2()
        +++ , isSecure: Utils.isCheckoutDomain()

...
681:     , getFieldValues: function ()
    {
        'use strict';
        ---         var order_field_keys = this.isSecure ? SC.Configuration.order_checkout_field_keys :
SC.Configuration.order_shopping_field_keys;
        +++         var order_field_keys = Utils.isInCheckout(request) && session.isLoggedIn2() ?
SC.Configuration.order_checkout_field_keys : SC.Configuration.order_shopping_field_keys;

...
}
```

Update the `verifySession()` method to remove the legacy domain check.

```
...
1726:     Application.defineModel('ProductList', {
```

```

...
// Returns a product list based on a given userId and id
1759: , get: function (user, id)
{
---     // Verify session if and only if we are in My Account...
---     if (request.getURL().indexOf('https') === 0)
---     {
---         this.verifySession();
---     }
+++     this.verifySession();

...
// Retrieves all Product Lists for a given user
1896: , search: function (user, order)
{
---     // Verify session if and only if we are in My Account...
---     if (request.getURL().indexOf('https') === 0)
---     {
---         this.verifySession();
---     }

var filters = [new nlobjSearchFilter('isinactive', null, 'is', 'F')
, new nlobjSearchFilter('custrecord_ns_pl_pl_owner', null, 'is', user)]
, template_ids = []

...

```

sc.environment.ssp

File locations:

- Checkout > Custom Checkout > sc.environment.ssp
- MyAccount > Custom MyAccount > sc.environment.ssp
- ShopFlow > Custom ShopFlow > sc.environment.ssp

Add a touchpoint check for Checkout and update the `session.GetAbsoluteUrl()` parameters to replace the legacy domain check.

```

...
94: <% if (SiteSettings) { %>
    // Site Settings Info
    <%
    // under some wired cases the terms and conditions bring a script tag if there is a body tag present
    // This code eliminates it in the case
    var site_settings_json = JSON.stringify(SiteSettings).replace(
/>/ig, '').replace(/<body.*?>/ig, '')
    %>
    SC.ENVIRONMENT.siteSettings = <%= site_settings_json %>;
    // Site site (ADVANCED or STANDARD)
    SC.ENVIRONMENT.siteType = '<%= SiteSettings.sitetype %>';

```

```

+++      // SCTouchpoint indicates the touchpoint the user is effectively in. We can only know with certain
this in the proper ssp
+++      // because there is still code that depends on the touchpoint
+++      // when in single ssp check if this it's necessary
+++      SC.ENVIRONMENT.SCTouchpoint = 'checkout';
<% } %>

...
179:     if(!SC.ENVIRONMENT.baseUrl)
{
---      SC.ENVIRONMENT.baseUrl = '<%= session.getAbsoluteUrl(request.getURL().indexOf('https:') === 0 ?
'checkout' : 'shopping', '/{{file}})' %>';
+++      SC.ENVIRONMENT.baseUrl = '<%= session.getAbsoluteUrl('checkout', '/{{file}})' %>';
}

...

```

Add a touchpoint check for MyAccount and update the `session.GetAbsoluteUrl()` parameters to replace the legacy domain check.

```

...
106: <% if (SiteSettings) { %
// Site Settings Info
SC.ENVIRONMENT.siteSettings = <%= JSON.stringify(SiteSettings) %>;
// Site site (ADVANCED or STANDARD)
SC.ENVIRONMENT.siteType = '<%= SiteSettings.sitetype %>';

+++      // SCTouchpoint indicates the touchpoint the user is effectively in. We can only know with certain
this in the proper ssp
+++      // because there is still code that depends on the touchpoint
+++      // myaccount value is added just in case someone needs it
+++      // when in single ssp check if this it's necessary
+++      SC.ENVIRONMENT.SCTouchpoint = 'myaccount';

<% } %>

...
205:   if(!SC.ENVIRONMENT.baseUrl)
{
---      SC.ENVIRONMENT.baseUrl = '<%= session.getAbsoluteUrl(request.getURL().indexOf('https:') === 0 ?
'checkout' : 'shopping', '/{{file}})' %>';
+++      SC.ENVIRONMENT.baseUrl = '<%= session.getAbsoluteUrl('checkout', '/{{file}})' %>';
}

...

```

Add a touchpoint check for ShopFlow and update the `session.GetAbsoluteUrl()` parameters to replace the legacy domain check.

```

...
73: <% if (SiteSettings) { %>
```

```

// Site Settings Info
SC.ENVIRONMENT.siteSettings = <%= JSON.stringify(SiteSettings) %>

+++      // SCTouchpoint indicates the touchpoint the user is effectively in. We can only know with certain
this in the proper ssp
+++      // because there is still code that depends on the touchpoint
+++      // when in single ssp check if this it's necessary
+++      SC.ENVIRONMENT.SCTouchpoint = 'shopping';

<% } %>

...
115: if (!SC.ENVIRONMENT.baseUrl)
{
---      SC.ENVIRONMENT.baseUrl = '<%= session.getAbsoluteUrl(request.getURL().indexOf('https:') === 0 ?
'checkout' : 'shopping', '/{{file}}') %>';
+++      SC.ENVIRONMENT.baseUrl = '<%= session.getAbsoluteUrl('shopping', '/{{file}}') %>';
}

...

```

index.ssp and index-local.ssp

File locations:

- ShopFlow > Custom ShopFlow > index.ssp
- MyAccount > Custom MyAccount > index.ssp
- Checkout > Custom Checkout > index.ssp
- ShopFlow > Custom ShopFlow > index-local.ssp
- MyAccount Premium> Custom MyAccount Premium > index-local.ssp

Modify each of these files to add the `&X-SC-Touchpoint=` query parameter when requesting `sc.environment.ssp` and `sc.user.environment.ssp`.

ShopFlow > Custom ShopFlow > index.ssp

```

...
146:   <script>
        if (!SC.isCrossOrigin())
        {
            // Do we have SEO Support
            if (SC.isPageGenerator())
            {
                document.body.className = document.body.className + ' seo-support';
            }
            SC.ENVIRONMENT.seoSupport = !!~document.body.className.indexOf("seo-support");
            /* load language and sc.environment.ssp */
            loadScript([
...
            '<%= session.getAbsoluteUrl("shopping", "sc.environment.ssp?lang=' + Language + "&cur=" +
Currency) %>'
+++          '<%= session.getAbsoluteUrl("shopping", "sc.environment.ssp?lang=' + Language + "&cur=" +
Currency + "&X-SC-Touchpoint=shopping") %>'
```

```

        ,   '<%= session.getAbsoluteUrl("shopping", "languages/" + Language + ".js") %>'
    ]);
    if (SC.isPageGenerator())
    {
        SC.ENVIRONMENT.PROFILE = {};
    }
    // Loads the application files, if you need to have a less agresive cacheing you can move them
    // to the sc.environment.ssp (Moderate cacheing) or to the sc.user.environment.ssp (No cache but
    less performant)
    loadScript([
        '<%= session.getAbsoluteUrl("shopping", "js/libs/Libraries-014c760ca5c2.js") %>',
        '<%= session.getAbsoluteUrl("shopping", "templates/Templates-0152658bf631.js") %>',
        '<%= session.getAbsoluteUrl("shopping", "js/Application-01526a8e6d58.js") %>'
        // , '/cms/1/cms.js'
    ]);
    if (SC.ENVIRONMENT.jsEnvironment == 'browser')
    {
        loadScript({
--- url: '<%= session.getAbsoluteUrl("shopping", "sc.user.environment.ssp?lang=' + Language + "&cur=" + Currency)
%>&t=' + new Date().getTime()
+++ url: '<%= session.getAbsoluteUrl("shopping", "sc.user.environment.ssp?lang=' + Language + "&cur=" + Currency
+ "&X-SC-Touchpoint=shopping") %>&t=' + new Date().getTime()
            , async: true
        });
    }
}
</script>

...

```

MyAccount > Custom MyAccount > index.ssp

```

...
<script>
var SC = window.SC = {
    ENVIRONMENT: {
        jsEnvironment: (typeof nsglobal === 'undefined') ? 'browser' : 'server'
    }
    , isCrossOrigin: function() { return '<%= Environment.currentHostString %>' !
== document.location.hostname; }
    , isPageGenerator: function() { return typeof nsglobal !== 'undefined'; }
    , getSessionInfo: function(key)
    {
        var session = SC.SESSION || SC.DEFAULT_SESSION || {};
        return (key) ? session[key] : session;
    }
};
</script>
65: --- <script> src="<%= session.getAbsoluteUrl('checkout', 'sc.environment.ssp?lang=' + Language + '&cur=' +
Currency) %>"</script>
65: +++ <script> src="<%= session.getAbsoluteUrl('checkout', 'sc.environment.ssp?lang=' + Language + '&cur=' +
Currency + "&X-SC-Touchpoint=myaccount") %>"</script>

...

```

Checkout > Custom Checkout > index.ssp

```
...
139: <% if (login) { %>
---     <script src="<%= session.getAbsoluteUrl('checkout', 'sc.environment.ssp?lang=' + Language + '&cur=' +
Currency + (cart_bootstrap ? "&cart-bootstrap=T" : "") ) %>"></script>
+++     <script src="<%= session.getAbsoluteUrl('checkout', 'sc.environment.ssp?lang=' + Language + '&cur=' +
Currency + (cart_bootstrap ? "&cart-bootstrap=T" : "") + "&X-SC-Touchpoint=checkout" ) %>"></script>
<% } else { %>
<script>
    loadScript({
---         url: '<%= session.getAbsoluteUrl("checkout", "sc.environment.ssp?lang=' + Language + '&cur=' +
Currency + (cart_bootstrap ? "&cart-bootstrap=T" : "") ) %>&t=' + (new Date().getTime())
+++         url: '<%= session.getAbsoluteUrl("checkout", "sc.environment.ssp?lang=' + Language +
"&cur=" + Currency + (cart_bootstrap ? "&cart-bootstrap=T" : "") + "&X-SC-Touchpoint=checkout" ) %>&t=' +
(new Date().getTime())
    });
</script>
<% } %>

...

```

ShopFlow > Custom ShopFlow > index-local.ssp

```
...
80: /* load language and sc.environment.ssp */
loadScript([
---     '<%= session.getAbsoluteUrl("shopping", "sc.environment.ssp?v=711&lang=' + Language + '&cur=' +
Currency) %>'
+++     '<%= session.getAbsoluteUrl("shopping", "sc.environment.ssp?v=711&lang=' + Language + '&cur=' +
Currency + "&X-SC-Touchpoint=shopping") %>'
        ,   '<%= session.getAbsoluteUrl("shopping", "languages/" + Language + ".js") %>' 
]);
if (SC.ENVIRONMENT.jsEnvironment === 'server')
{
    SC.ENVIRONMENT.PROFILE = {};
}
loadScript([
    '<%= root %>templates/Templates.php'
    , '<%= root %>js/Application.php'
]);

if (SC.ENVIRONMENT.jsEnvironment == 'browser')
{
    loadScript({
--- url: '<%= session.getAbsoluteUrl("shopping", "sc.user.environment.ssp?lang=' + Language + '&cur=' + Currency) %>&t=' + new Date().getTime()
+++ url: '<%= session.getAbsoluteUrl("shopping", "sc.user.environment.ssp?lang=' + Language + '&cur=' + Currency +
+ "&X-SC-Touchpoint=shopping") %>&t=' + new Date().getTime()
        , async: true
    });
}

```

```
...
```

MyAccount Premium> Custom MyAccount Premium > index-local.ssp

```
...
```

```
<script src="<% root %>js/utils/BootUtilities.js"></script>
70: --- <script src="<% session.getAbsoluteUrl('checkout', 'sc.environment.ssp?lang=' + Language + '&cur=' +
Currency) %>"></script>
+++ <script src="<% session.getAbsoluteUrl('checkout', 'sc.environment.ssp?lang=' + Language + '&cur=' +
Currency + '&X-SC-Touchpoint=myaccount') %>"></script>
```

```
...
```

jQuery.ajaxSetup.js

File locations:

- ShopFlow > Custom ShopFlow > js > src > core > extras > jQuery.ajaxSetup.js
- MyAccount > Custom MyAccount > js > src > core > extras > jQuery.ajaxSetup.js
- Checkout > Custom Checkout > js > src > core > extras > jQuery.ajaxSetup.js

Update the `beforeSend()` method to send the header to the backend for each request.

```
...
```

```
// This registers an event listener to any ajax call
85:  jquery(document)
    // http://api.jquery.com/ajaxStart/
    .ajaxStart(SC.loadingIndicatorShow)
    // http://api.jquery.com/ajaxStop/
    .ajaxStop(SC.loadingIndicatorHide);
// http://api.jquery.com/jQuery.ajaxSetup/

    jquery.ajaxSetup({
93: beforeSend: function (jqXHR, options)
    {
        // BTW: "!"~" means "==" -1"
        if (!~options.contentType.indexOf('charset'))
        {
            // If there's no charset, we set it to UTF-8
            jqXHR.setRequestHeader('Content-Type', options.contentType + '; charset=UTF-8');
        }

        +++ // Add header so that suitescript code can know the current touchpoint
        +++ jqXHR.setRequestHeader('X-SC-Touchpoint', SC.ENVIRONMENT.SCTouchpoint);
    }
}
```

```
...
```

live-order.ss and live-order-line.ss

File locations:

- ShopFlow > Custom ShopFlow > services > live-order.ss
- MyAccount > Custom MyAccount > > live-order.ss
- Checkout > Custom Checkout > services > live-order.ss
- ShopFlow > Custom ShopFlow > services > live-order-line.ss
- MyAccount > Custom MyAccount > services > live-order-line.ss
- Checkout > Custom Checkout > services > live-order-line.ss

In each file, add an **if** statement to verify if the user is in shopping or logged in.

```
...
// If we are not in the checkout OR we are logged in
// When on store, login in is not required
// When on checkout, login is required
--- if (!~request.getURL().indexOf('https') || session.isLoggedIn())
+++ if (Utils.isInShopping(request) || session.isLoggedIn2())

...
```

CheckoutSkipLogin.js

Customizing this file requires that you copy the **CheckoutSkipLogin** module folder from Reference Checkout to Custom Checkout because it is not created automatically.

Update Checkout > Custom Checkout > js > src > app > modules > CheckoutSkipLogin > CheckoutSkipLogin.js to use the new domain check method.

```
return promise;
};

--- // don't wrap on non-secure domains (Site Builder cart is in Checkout :/ )
89: --- if (window.location.protocol !== 'http:')
+++ // Site Builder cart is in Checkout :/ don't wrap if in shopping
+++ if (Utils.isInCheckout())
{
    LiveOrderModel.prototype.save = _wrap(LiveOrderModel.prototype.save, wrapper);
}

...
```

NavigationHelper.js

Customizing this file requires that you copy the **NavigationHelper** module folder from Reference ShopFlow to Custom Shopflow because it is not created automatically.

File locations:

- ShopFlow > Custom ShopFlow >js > src > app > modules > NavigationHelper > NavigationHelper.js
- MyAccount > Custom MyAccount >js > src > app > modules > NavigationHelper > NavigationHelper.js
- Checkout > Custom Checkout >js > src > app > modules > NavigationHelper > NavigationHelper.js

Add the new domain check methods in each of the files.

```

...
    // if we are heading to a secure domain (myAccount or checkout), keep setting the language by url
299: ---  if (target_touchpoint.indexOf('https:') >= 0)
+++  if (Utils.isInCheckout())
{
    var current_language = SC.ENVIRONMENT.currentLanguage;
    if (current_language)
    {
        target_data.parameters = target_data.parameters ?
            target_data.parameters + '&lang=' + current_language.locale :
            'lang=' + current_language.locale;
    }
}
...

```

Update Legacy Checks and Methods

Previous sections have covered customizing multiple files. This procedure provides steps for finding and replacing any other instances of the legacy domain checks, the legacy `getAbsoluteUrl()` methods, and the legacy `session.isLoggedIn()` methods.

To update the legacy checks and methods:

1. Search the files for remaining instances of these legacy domain checks.
 - `!~request.getURL().indexOf('https')`
 - `isSecure`
 - `isSecureDomain`
 - a. If instances are located in files that have not already been copied to the Custom folder, copy as needed.
 - b. For each instance, determine the reason for the domain check.
 - Was it to protect a resource from being accessed through the shopping domain? In this case, call the `Utils.isCheckoutDomain()` method.
 - Does the method return a different result depending on whether we are in the checkout domain or the shopping domain? In this case, call the `Utils.isInCheckout()` method.
 - Is there a visual change in the web store depending on whether we are in the checkout domain or the shopping domain? Call the `Utils.isInCheckout()` method or the `Utils.isInShopping()` method as appropriate for each domain.
 - c. Replace each instance with the appropriate new domain check method.

See the code snippet in [commons.js](#) for the method definitions.

- `Utils.isCheckoutDomain()`
- `Utils.isInCheckout()`
- `Utils.isInShopping()`
- `Utils.isSingleDomain()`

2. Search the files for remaining instances of:

```
session.getAbsoluteUrl(request.getURL().indexOf('https:') === 0 ?
  'checkout' : 'shopping', '/{{file}}') :
```

- a. If instances are located in files that have not already been copied to the Custom folder, copy as needed.
- b. In Checkout or MyAccount, replace the instance with `session.getAbsoluteUrl('checkout', '/{{file}}').`
- c. In Shopping, replace the instance with `session.getAbsoluteUrl('shopping', '/{{file}}').`
- d. You can also replace any instance with:
`session.getAbsoluteUrl(Utils.isInCheckout(request) ? 'checkout' : 'shopping', '/{{file}}').`

This method call will check for either domain.

3. Search the files for remaining instances of the `session.isLoggedIn()` method.
 - a. If instances are located in files that have not already been copied to the Custom folder, copy as needed.
 - b. Replace each instance with the `session.isLoggedIn2()` method.

Deploy the Migration

When all customizations are complete, you need to configure NetSuite to use the custom files instead of the reference files. Complete the following tasks:

- Configure the secure domain.
- Add or update library scripts for each customized SSP Application.
- Add or update touch points for each customized SSP Application.
- Deploy your site to the secure domain.



Important: These customizations are extensive. You should test them in a sandbox account or other test site before deploying to a production account.

To deploy the migration:

1. If you have not already done so, complete the steps to configure a Secure Domain. For details, refer to [Set Up a Web Store Domain](#).
2. Go to Setup > SuiteCommerce Advanced > SSP Applications.
3. Click **Edit** to modify the custom SSP Application.
4. On the **Library** subtab under **Scripts**, add the library scripts as indicated:



Important: The order of these scripts must match exactly.

Web Site Hosting Files/Live Hosting Files/SSP Applications/<application>/...

- ... Reference <SSP Application>/ssp_libraries/underscore.js
- ... Reference <SSP Application>/ssp_libraries/backbone.validation.js
- ... Custom <SSP Application>/ssp_libraries/commons.js
- ... Reference <SSP Application>/ssp_libraries/backend.configuration.js
- ... Custom <SSP Application>/ssp_libraries/Models.js

For example, to update the commons.js file for ShopFlow 1.07.0, go to:

Web Site Hosting Files/Live Hosting Files/SSP Applications/NetSuite Inc. — ShopFlow 1.07.0/Custom ShopFlow /ssp_libraries/commons.js

5. Click on the **Supported Touch Points** subtab.
6. Add the appropriate touch points using the custom .ssp files that were modified as part of this migration. See the help topic [Select Supported Touch Points](#).
7. Click **Save**.
8. Deploy the Custom SSP Applications to your Secure Domain.
 - a. Select the **Save & Link to Domain** button (from the **Save** drop down menu).
 - b. Choose your secure domain.
 - c. Click **Save**.
9. Repeat the Scripts, Touch Points, and Deploy steps for each SSP Application you customized (Custom Checkout, Custom MyAccount, and Custom ShopFlow).

PayPal Address not Retained in Customer Record

 **Applies to:** SuiteCommerce Advanced | Vinson

For Vinson releases and earlier, when a PayPal address is submitted during a web store transaction, the NetSuite customer address is not updated to reflect the PayPal address. Instead, the original customer address within the NetSuite customer record is retained. Apply the patch described in this section to retain the PayPal address in the NetSuite customer record.



Important: Making changes to core JavaScript source files or changing any vital functionality of the application can make migrating to future releases difficult. Before making changes to SuiteCommerce Advanced, see [Customize and Extend Core SuiteCommerce Advanced Modules](#).

Step 1: Create the LiveOrder.Model.js Override file

1. If you have not done so already, create a directory to store your custom module.
2. Open this directory and create a subdirectory to maintain your customizations. Give this directory a name similar to the module being customized. For example:
Modules/extensions/Live.Order.Override@1.0.0
3. In the Live.Order.Override@1.0.0 directory, create a subdirectory called SuiteScript.
4. Go to the SuiteScript directory located under the Modules > suitecommerce > LiveOrder@X.X.X directory.
5. Copy the LiveOrder.Model.js file from Modules > suitecommerce > LiveOrder@X.X.X into the Live.Order.Override@1.0.0 > SuiteScript directory.
6. Update the file to remove the following line from the **submit()** method.

```
this.removePaypalAddress(paypal_address);
```



Note: This customization can also be applied to pre-Denali implementations of SuiteCommerce Advanced. Remove the line of code from the submit method located at **ssp_libraries/Models.js**

Step 2: Prepare the Developer Tools for Your Override

1. Create the ns.package.json file for the LiveOrderOverride@X.X.X directory. Add the following code:

```
{
```

```

    "gulp": {
      "ssp-libraries": [
        "SuiteScript/*.js"
      ],
      "overrides": {
        "suitecommerce/LiveOrder@X.X.X/SuiteScript/LiveOrder.Model.js" :
        SuiteScript/LiveOrder.Model.js"
      }
    }
  
```

2. Update the distro.json file.

```

  },
  "modules": {
...
  "suitecommerce/LiveOrder": "X.X.X",
  "extensions/Live.Order.Override": "X.X.X",

```

Step 3: Test and Deploy Your Override

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)).
Since this customization modifies a file that is stored as an SSP library, changes are not immediately visible in your local environment. You must first deploy your custom module directly to NetSuite. See [Deploy to NetSuite](#) for more information.
2. Confirm your results.

Login Email Address Appears in the Password Reset URL

 **Applies to:** SuiteCommerce Advanced | Elbrus

This patch adds a method named `sendPasswordRetrievalEmail2()`, included in Kilimanjaro and later implementations of SuiteCommerce Advanced. Located in the Commerce API, `sendPasswordRetrievalEmail2()` generates a password reset email message. For added security, the original login email address for the customer does not appear in the password reset URL generated by this method. Other ecommerce solutions commonly use this secure solution.

To implement this patch, you extend JavaScript functions in the `Account` and `LoginRegister` modules and override the `login_register_reset_password.tpl` template file. For an example of the changes needed for this patch, see [EmailAddressPasswordResetURL.zip](#).

This method replaces the `sendPasswordRetrievalEmail()` method. However, the deprecated `sendPasswordRetrievalEmail()` method will continue to operate without change.

 **Note:** Before proceeding, familiarize yourself with [Best Practices for Customizing SuiteCommerce Advanced](#). The following sections show how to [Extend JavaScript](#) to implement the patch.

Step 1: Create and Copy the Required Files

Create the custom directories used by the custom module that you create for the patch, extend the functions in the `Account.Model.js` and `LoginRegister.ResetPassword.View.js` JavaScript files, and override the `login_register_reset_password.tpl` template file.

This section shows how to create custom modules that use the JavaScript **prototype** object to extend the functions that require a code change for the patch.

1. If you have not done so already, create a directory to store your custom modules, for example, create **Modules/extensions**.

2. Open this directory and create the following subdirectories to maintain your customizations.

Give this directory a name similar to the module being customized. For example, create the following directories:

Modules/extensions/AccountExtension@1.0.0

Modules/extensions/LoginRegisterExtension@1.0.0

3. In the **AccountExtension@X.X.X** directory, create a **SuiteScript** subdirectory. In the **SuiteScript** subdirectory, create a JavaScript file.

To follow best practices, name the JavaScript file **Account.Model.Extension.js**.

4. Open this file and extend the **forgotPassword** method as shown in the following code snippet:

```
define(
  'Account.Model.Extension'
  [
    'SC.Model',
    'Application',
    'Models.Init'

    , 'underscore'
  ]
  , function (
    SCModel
    , Application
    , ModelsInit

    ,
    -
  )
{
  'use strict';

  _.extend(AccountModelExtension.prototype,
  {
    forgotPassword: function (email)
    {
      try
      {
        // this API method throws an exception if the email doesn't exist
        // 'The supplied email has not been registered as a customer at our Web store.'
        ModelsInit.session.sendPasswordRetrievalEmail2(email);
      }
      catch (e)
      {
        var error = Application.processError(e);
        // if the customer failed to log in previously
        // the password retrieval email is sent but an error is thrown
        if (error.errorCode !== 'ERR_WS_CUSTOMER_LOGIN')
        {
          throw e;
        }
      }
    }
  });
}
```

```

        }

        return {
            success: true
        };
    }

});
});

```

5. In the `LoginRegisterExtension@X.X.X` directory, create a `JavaScript` subdirectory. In the `JavaScript` subdirectory, create a `JavaScript` file named `LoginRegister.ResetPassword.View.Extension.js`.
6. Open this file and extend the `function name` method as shown in the following code snippet:

```

define('LoginRegister.ResetPassword.View.Extension'
, [
    'SC.Configuration'
, 'Account.ResetPassword.Model'
, 'Backbone.FormView'

, 'Backbone'
, 'underscore'
]
, function (
    Configuration
, AccountResetPasswordModel
, BackboneFormView

, Backbone
, _
)
{
    'use strict';

    _.extend(LoginRegisterResetPasswordViewExtension.prototype,
    {
        initialize: function ()
        {
            this.model = new AccountResetPasswordModel();
            this.model.set('params', {'cb':_.parseUrlOptions(location.search).cb});
            this.model.on('save', _.bind(this.showSuccess, this));

            BackboneFormView.add(this);
        }
    });
}
);

```

7. In the `LoginRegisterExtension@X.X.X` directory, create a `Templates` subdirectory. Copy the `Modules/suitecommerce/LoginRegister@2.3.0/Templates/login_register_reset_password.tpl` template file into the `Templates` directory.
8. Open `login_register_reset_password.tpl` and make the following change. Replace this HTML:

```

<p class="login-register-reset-password-description">
    {{translate 'Enter a new password below for <b>$0</b>' email}}

```

```
</p>
```

With this HTML:

```
<p class="login-register-reset-password-description">
    {{translate 'Enter a new password below'}}
</p>
```

Step 2. Prepare the Developer Tools for Your Overrides

1. Create the `ns.package.json` file for the `AccountExtension@X.X.X` directory. Add the following code to `ns.package.json` in the `Modules/extensions/Account@X.X.X` directory:

```
{
  "gulp": {
    "ssp-libraries": [
      "SuiteScript/*.js"
    ],
  }
}
```

2. Create the `ns.package.json` file for the `LoginRegisterExtension@X.X.X` directory. Add the following code to `ns.package.json` in the `Modules/extensions/LoginRegister@X.X.X` directory:

```
{
  "gulp": {
    "javascript": [
      "JavaScript/*.js"
    ],
    "templates": [
      "JavaScript/*.js"
    ]
  },
  "overrides": {
    "suitecommerce/LoginRegister@X.X.X/Templates/login_register_reset_password.tpl" :
    Templates/login_register_reset_password.tpl
  }
}
```

3. In `distro.json`, add your custom modules to the `modules` object.

This ensures that the Gulp tasks include your extension when you deploy. In this example, the extension `modules` are added at the beginning of the list of modules. However, you can add the modules anywhere in the `modules` object. The order of precedence in this list does not matter.

```
{
  "name": "SuiteCommerce Advanced Elbrus",
  "version": "2.0",
  "buildToolsVersion": "1.3.0",
  "folders": {
    "modules": "Modules",
    "suitecommerceModules": "Modules/suitecommerce",
    "extensionsModules": "Modules/extensions",
    "thirdPartyModules": "Modules/third_parties",
    "distribution": "LocalDistribution",
    "deploy": "DeployDistribution"
  },
  "modules": {
```

```
"extensions/AccountExtension": "X.X.X",
"extensions/LoginRegisterExtension": "X.X.X",
...
```

Step 3. Test and Deploy Your Override

1. Test your source code customizations on a local server (see [SCA on a Local Server](#)) or deploy them to your NetSuite account (see [Deploy to NetSuite](#)).

Since this customization modifies a file that is stored as an SSP library, changes are not immediately visible in your local environment. You must first deploy your custom module directly to NetSuite. See [Deploy to NetSuite](#) for more information.

2. Confirm your results.

How to Apply .patch Files

Some releases of SuiteCommerce Advanced include a series of .patch files, which contain the complete diffs between new and previous releases across all source files. In some cases, a patch might be available for specific purposes, such as for backporting code to previous releases to support new features. If you encounter a .patch file, refer to this section to learn how to interpret and apply it.

A patch file contains the differences between two identically titled files. The patch designates these folders as **a** and **b**. Distributing the patch to an identically titled path/folder allows the patch to transform changes in **b** (the source of the changes) to the matching file in **a** (the target of the changes).

Follow the appropriate procedure to apply the diff, depending on how you customized the source code in previous releases.

- If you have not altered any SuiteCommerce Advanced source files in `Modules/suitecommerce/`, you can use third-party software to run the patch.
- If you have modified any source files in `Modules/suitecommerce/`, you must implement changes manually.



Important: Applying a diff adds and removes lines of code. This can break your site if you are not careful. Do not apply any diffs using third-party software unless you have followed NetSuite's best practices for customizing JavaScript files as outlined in [Best Practices for Customizing SuiteCommerce Advanced](#). Do not patch your live site directly. Create a test site or test files locally using the dev tools. Failure to test any changes thoroughly before deploying to your live site can result in data loss.

For a list of available patches, see [Patches](#).

To apply a diff using third-party software:

1. Download the correct diff file to the target directory.
2. Make a backup of your original source code.
3. Review the diff as applicable. See [Diff File Structure](#) for details.
4. Ensure that you have not altered any source code provided by NetSuite and apply the diff using your preferred third-party method.

Various programs are available that can apply a diff file to your SCA source code. The following example applies a Mont Blanc release diff using Git.

```
cd montblanc
```

```
git init
git apply /path/montblanc-sensors.patch
rm -rf .git
```

To apply a diff manually:

1. Download the correct diff file.
2. Open the diff.
3. Make a backup of your local source code.
4. Make all changes manually in your local source code. For a detailed description of a diff file and its components, see [Diff File Structure](#)

Diff File Structure

This section explains how SuiteCommerce Advanced diff files are structured. The information in this section should help you understand any diff files provided by NetSuite for SuiteCommerce Advanced.



Note: Surrounding code and estimated line numbers vary depending on each file due to existing customizations. These instructions include them here as reference points only.

Example Diff File

The following example shows a simple diff (.patch file). This example only compares one file, although most diffs include more than one file. One diff file typically contains many comparisons, each with its own components, as described in this section.

```
diff --git a/myDirectoryPath/myFile.js b/myDirectoryPath/myFile.js
index 5d7cd7e..fca4091 100755
--- a/myDirectoryPath/myFile.js
+++ b/myDirectoryPath/myFile.js
@@ -76,7 +77,7 @@ define('Sensors'

{
  _.each(arguments, function (argument)
  {
-    if(argument)
+    if (argument)
    {
      _.extend(data, argument);
    }
}
```

File Comparison Header

The File Comparison Header is a statement explaining two files being compared. The two files being compared include leading directory names **a** and **b** to distinguish the two files, which typically have identical names and locations.

This example shows changes between two versions of myFile.js. Note the file names and paths are identical. **--git** simply informs you that this diff is written in Git-specific diff format. Each comparison within the diff begins with a header.

```
diff --git a/myDirectoryPath/myFile.js b/myDirectoryPath/myFile.js
```

Index

Technically part of the Header, the index is a line of metadata describing technical information about the given files. The first two numbers (separated by ..) represent the IDs of the two files being compared. In this case, each version of the project file is an object at a specific revision.

The last number is a mode identifier for the file containing the changes:

- 100644 specifies a normal, non-executable file.
- 100755 specifies an executable file.
- 120000 specifies a symbolic link.

In this example, this part of the diff is comparing two different file IDs (**5d7cd7e** and **fca4091**) of myFile.js, which is an executable file.

```
index 5d7cd7e..fca4091 100755
```

Change Markers

The next two lines describe the two files (and paths) being compared, but with **+++** or **---** prefixes. The file with the minus symbols shows lines that exist within the **a** version but are missing from the **b** version. Likewise, the file with the plus symbols contains lines missing in the **a** version but are present in **b**.

In this example, **a** is the existing (old) file and **b** is the updated (new) file.

```
--- a/myDirectoryPath/myFile.js
+++ b/myDirectoryPath/myFile.js
```

Range Information

Diff files show only the portions of the file being changed. These portions are called hunks. One hunk of data containing changes is depicted within two sets of **@@** symbols. This information declares the lines within the hunk that are affected by the change. One index can contain multiple hunks, depicting the various changes throughout the file.

In this example, the diff contains one hunk of differences. Seven lines are extracted from the **a** file, beginning at line 76. Likewise, seven lines are displayed in the **b** file, beginning at line 77.

```
@@ -76,7 +77,7 @@
```

Changes

The changes to each hunk immediately follow the range information. Each changed line is prepended with either a **+** or a **-** symbol. These symbols act in the same way as they do in the Change Markers section. The diff removes lines prepended with a **-** and adds lines prepended with a **+** sign. If making changes to your code manually, any lines bearing the **-** prefix need to be removed. Any lines bearing the **+** prefix need to be added.

Additionally, a hunk typically contains unchanged lines before and after the modification to provide context for the change. In this example, the diff removes **if(argument)** and adds the corrected code: **if (argument)**.

```
define('Sensors'
```

```
{  
  _.each(arguments, function (argument)  
  {  
-    if(argument)  
+    if (argument)  
    {  
      _.extend(data, argument);  
    }  
}
```



Note: In diffs provided by NetSuite, **a** files and lines marked with **-** are old, and **b** files and lines marked with **+** are updates.

Upgrade SuiteCommerce to SuiteCommerce Advanced

 **Applies to:** SuiteCommerce Web Stores

This topic explains how to upgrade from SuiteCommerce to SuiteCommerce Advanced (SCA).

To upgrade from SuiteCommerce to SCA, follow these steps:

1. [Test Your Site on a Development Domain](#)
2. [Set Up Your Production Domain for SCA](#)
3. [Verify Production Domain Operation](#)

Before you begin

Before you begin upgrading your site:

- Contact your account representative to provision the appropriate modules for your NetSuite account.
- In NetSuite, go to Setup > SuiteCommerce Advanced > Extension Management, select your web site and domain, and then click **Next**. Make a note of the active themes and extensions on your SuiteCommerce site.

Test Your Site on a Development Domain

It is a best practice to use a development domain to test your existing themes and extensions on an SCA implementation before upgrading your live production domain to SCA. Testing your existing NetSuite and third-party themes and extensions helps ensure that your SCA domain performs as expected when you upgrade your production domain.

To test SCA on your development domain:

1. If you have not already done so, create a development domain in your existing SuiteCommerce web site record.
See the help topic [Set Up a Web Store Domain](#).
2. Go to Setup > SuiteCommerce Advanced > Set Up Web Site.
3. Click **Edit** to access the web site record.
4. On the **Domains** tab, clear the **Primary Web Site URL** checkbox for all of the listed domains.
5. Copy the configuration record from the existing SuiteCommerce production domain to the development domain.
See [Copy a SuiteCommerce Configuration Record](#).
6. Install the Theme and Extension SuiteApp.
See the help topic [Install Theme and Extension SuiteApps](#).
7. Link your development domain to the SuiteCommerce Advanced Dev application.
See the help topic [Link a Domain to an SSP Application](#). Use the latest release **SuiteCommerce Advanced — Dev** implementation for all touch points.

8. Activate the themes and extensions on the development domain.
See the help topic [Manage Themes and Extensions](#).
9. Clear (invalidate) the cache for the development **and** production domains.
See the help topic [Cache Invalidation Request](#).

Thoroughly test your themes and extensions on your development domain. When you are satisfied, set up your production domain for SCA.

Set Up Your Production Domain for SCA

After confirming that your site works as expected on the development domain, schedule the optimal time to upgrade your production domain to SCA. It is a best practice to schedule upgrades during a period of off-peak site usage.

To set up your production domain for SCA:

1. Assign the latest release **SuiteCommerce Advanced – Dev** implementation for all touch points (the same touch points you assigned on your development domain).
See [Selecting Supported Touch Points](#).



Note: As an alternative, you can also assign touch points directly on the domain record. Using this method overrides the touch points assigned on the Touch Points subtab on the Web Site Setup record.

2. Link your production domain to the **SuiteCommerce Advanced Dev** SSP application.
See the help topic [Link a Domain to an SSP Application](#).
3. Activate the themes and extensions that were previously active in your production domain.
See the help topic [Manage Themes and Extensions](#).
4. Clear (invalidate) the cache for the development and production domains.
See the help topic [Cache Invalidation Request](#).

Verify Production Domain Operation

Go to your production site URL in your browser to test operation of your SCA site.

- Confirm that your touch points (listed below) are correct and accessible in your browser:
 - Log In
 - Log Out
 - Register
 - Customer Center (My Account)
 - Checkout
 - View Cart
 - Home Page
- Confirm that your theme and extension activations work as expected.

Update SuiteCommerce Advanced

Applies to: SuiteCommerce Advanced

This section explains the versioning and release system governing all SuiteCommerce Advanced releases and how to advantage of updates.

This section includes the following topics:

- [Commerce Releases and Versioning](#) – Outlines the basic structure of the Commerce release cycle and versioning nomenclature.
- [Migrate to the Latest Release of SuiteCommerce Advanced](#) – Explains how to migrate to the latest release.



Important: SuiteCommerce Advanced releases are unmanaged bundles and require manual migration of customizations. Before migrating to a new release of SuiteCommerce Advanced, refer to the [SuiteCommerce Release Notes](#) for information on enhancements and fixes.

Commerce Releases and Versioning

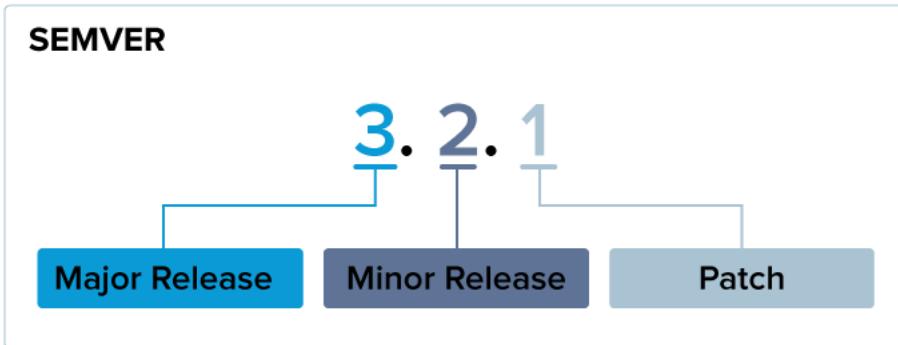
Both SuiteCommerce (SC) and SuiteCommerce Advanced (SCA) releases follow the same general Commerce Release Schedule, explained below. The main difference between the two products is that SC releases are managed with automated updates, while SCA releases are generally available as unmanaged SuiteApps with unique bundle IDs.

Each release includes a version number using a nomenclature that is based on, but does not completely adhere to, the semantic versioning schema, or SEMVER.

Both of these Commerce offerings provide the following releases, each generally occurring within an annual cycle:

- **Major Release** – These releases include significant product enhancements and new features or bug fixes.
- **Minor Releases** – These releases generally include bug fixes and periodic updates to core SCA developer tools. The effort to migrate existing customizations is minimal.

An example of an SC or SCA version is **2019.1.1**. This nomenclature correlates with SEMVER, with slightly different distinctions, as shown in the following image.



Migrate to the Latest Release of SuiteCommerce Advanced

To ensure that you implement the latest features, bug fixes, and changes to SuiteCommerce Advanced (SCA), install the latest major or minor release of the active version, whichever is highest. See the help topic [SuiteCommerce Release Notes](#) for the list of changes and bundle ID of the latest version.

Because SuiteCommerce Advanced SuiteApps are **unmanaged** bundles, updates are not automatically pushed to your site. This structure is in place to ensure that existing customizations are not impacted unexpectedly with each update. Therefore, you must manually update the bundle to the latest major or minor release to receive the latest changes.

The correct procedures to follow depend on the current release of your SCA site. Follow the correct procedures, described below:

- [Migrate from Aconcagua and Later](#)
- [Migrate From Kilimanjaro and Earlier](#)



Important: Performing a bundle update causes all files in the existing SCA Source directory to be replaced with the updated set of files. The new source file changes and your existing customizations must be re-deployed from your local environment before changes take effect on a domain pointing to the Development SSP application. Even though both major and minor releases maintain the same Bundle ID, you must still re-link your site record and domain to the updated SSP application.

These topics also apply to administrators testing a new major or minor release by deploying to a development SSP within a site record.

Migrate from Aconcagua and Later

The Aconcagua release of SCA and later all support themes and extensions and the Extension Framework (also referred to as the Extensibility API). Follow the instructions in this section if you are currently implementing the Aconcagua release of SCA or higher.



Important: If you have modified any core SCA files in your existing implementation, you should recreate these changes as extensions instead of migrating your customizations manually. If your existing customizations modify objects not accessible through the Extensibility API, you must migrate this custom code manually to the new version. See the help topic [Extensibility API](#) for more information.



Important: Before migrating to a new release, review the [SuiteCommerce Release Notes](#) for important notifications and details regarding changes included in the release. Each major release might include bug fixes and changes in the form of periodic minor releases. Make sure you implement the latest minor release of the active version.

To migrate to the latest release:

1. Install the new bundle.
 1. Go to Customization > SuiteBundler > Search & Install Bundles.
 2. Enter the bundle ID for the latest version of SCA you want to install. Bundle IDs are listed in the [SuiteCommerce Release Notes](#).
 3. Select the bundle in the results list and then click **Install**. Follow any on-screen prompts to continue the install.

The bundle installation may take a while. A pending message displays in the status column of the Installed Bundles page. Refresh the screen during the process to see when the installation is complete.



Note: Minor releases to the current SCA release do not receive new Bundle IDs.

2. Download the new source code from the file cabinet to your local environment.
The source zip file is located at Web Site Hosting Files > Live Hosting Files > SSP Applications > NetSuite Inc. — SCA [version] > Source > _Sources. After downloading is complete, unzip the file into a new working directory.
3. All customizations should be implemented using themes and extensions. However, if your existing implementation includes customizations not accessible using the Extensibility API, migrate these customizations to your new source code directory.
 - a. Create a duplicate of your custom module folder in the new source code directory.



Note: This applies to custom modules that are not standard SCA source modules. Any source SCA modules that have been customized may require comparison against the new version plus any required manual edits to the code.

- b. Copy your existing custom modules into the new custom module folder.
- c. Update the new distro.json file to include references to any custom modules and to define any custom application dependencies.



Important: Do not copy the existing distro.json file contents directly into the new distro.json file. Doing this causes overwrites to changes made in the distro.json file to support the new release. For example, a new module may be referenced. NetSuite recommends evaluating the DIFF between files to ensure that you migrate all changes from your local distro.json file into the new distro.json file.

- d. Any module that changed in the new release bears a higher version number from its previous release. Compare module version numbers to assess which modules have been updated and migrate the changes to any existing customizations as necessary. Refer to the [SuiteCommerce Release Notes](#) for details on which modules were changed and why.



Important: Migrating to a new release requires testing to ensure any existing customizations remain viable. Evaluate differences in code against your existing customizations and test thoroughly before deploying to your live site. See [Best Practices for Customizing SuiteCommerce Advanced](#).

4. Link your site record and domain to an SSP application as required. This can be the SSP Application for the latest version of SCA or one specifically intended for development.
 - a. Go to Setup > SuiteCommerce Advanced > SSP Applications and click **View** next to the SuiteCommerce Advanced Dev <version> application.
 - b. Click **Link to Domain**.
 - c. Select the development domain for this SSP Application.
5. Examine all customizations to your existing source code of the previous implementation. If all customizations of your previous implementation were accomplished using themes and extensions, then reactivate them using the Extension Manager in NetSuite. See the help topic [Manage Themes and Extensions](#) for details.
6. If your previous implementation included customizations of SCA source code that were not part of the Extensibility API (implemented without extensions), then complete the following:
 - a. Create a duplicate of your custom module folder in the new source code directory.



Note: This applies to custom modules that are not standard SCA source modules. Any source SCA modules that have been customized may require comparison against the new version plus any required manual edits to the code.

- b. Copy your existing custom modules into the new custom module folder.
- c. Update the new distro.json file to include references to any custom modules and to define any custom application dependencies.



Important: Do not copy the existing distro.json file contents directly into the new distro.json file. Doing this causes overwrites to changes made in the distro.json file to support the new release. For example, a new module may be referenced. NetSuite recommends evaluating the DIFF between files to ensure that you migrate all changes from your local distro.json file into the new distro.json file.

- d. Any module that changed in the new release bears a higher version number from its previous release. Compare module version numbers to assess which modules have been updated and migrate the changes to any existing customizations as necessary. Refer to the [SuiteCommerce Release Notes](#) for details on which modules were changed and why.



Important: Migrating to a new release requires testing to ensure any existing customizations remain viable. Evaluate differences in code against your existing customizations and test thoroughly before deploying to your live site. See [Best Practices for Customizing SuiteCommerce Advanced](#).

- e. Deploy your local files and test appropriately.

Testing may consist of various scenarios depending on your setup. You can deploy your local files to any of the following locations:

- Core source code customizations require the core SCA developer tools. This requires setting up the appropriate developer environment. See [Core SuiteCommerce Advanced Developer Tools](#) for details. Core SCA customizations can be deployed to:
 - A primary SSP Application linked to a primary domain (live site)
 - Development SSP Application linked to a development domain
 - Local server. See [SCA on a Local Server](#).
 - NetSuite sandbox account. See [Deploy to a NetSuite Sandbox](#).
 - Custom SSP Application. See [Deploy to a Custom SSP Application](#)

For example, you can deploy your customizations to a local server or to a NetSuite sandbox account for testing. If you do not have a Sandbox account, you can deploy to an SSP Application linked to a development domain in NetSuite. After testing, deploy your files to the SSP Application linked to the primary domain.

Migrate From Kilimanjaro and Earlier

Follow the instructions in this section if you are currently implementing any of the following releases of SCA. Migrating from these implementations requires adaptation of any current customizations into themes and extensions:

- Kilimanjaro
- Elbrus
- Vinson
- Mont Blanc
- Denali



Important: The best practice to customize SCA is through themes and extensions. Earlier versions of SCA (prior to Aconcagua) do not support themes and extensions, which do not require modification of source code. If you have modified any core SCA files in your existing implementation, best practice is to re-create these changes as themes or extensions. If your customizations modify objects not accessible through the Extensibility API, you must migrate this custom code manually to the new version. See the help topic [Extensibility API](#) for more information.

To migrate to the latest release:



Important: Before migrating to a new release, review the [SuiteCommerce Release Notes](#) for important notifications and details regarding changes included in the release. Each major release might include bug fixes and changes in the form of periodic minor releases. Make sure you implement the latest minor release of the active version.

1. Install the new bundle.
 1. Go to Customization > SuiteBundler > Search & Install Bundles.
 2. Enter the bundle ID for the latest version of SCA. Bundle IDs are listed in the [SuiteCommerce Release Notes](#).
 3. Select the bundle in the results list and then click **Install**. Follow any on-screen prompts to continue the install.

The bundle installation may take a while. A pending message displays in the status column of the Installed Bundles page. Refresh the screen during the process to see when the installation is complete.
2. Download the new source code from the file cabinet to your local environment.
The source zip file is located at Web Site Hosting Files > Live Hosting Files > SSP Applications > NetSuite Inc. — SCA [version] > Source > _Sources. After downloading is complete, unzip the file into a new working directory.
3. Examine all customizations to your existing source code of the previous implementation.



Important: The latest version of SuiteCommerce Advanced does not include Sass or HTML source and also utilizes the Extensibility API to customize most JavaScript, SuiteScript, and JSON. You must recreate existing customizations as themes or extensions to carry them over to your new implementation of SCA.

- a. Create one or more themes to implement any existing Sass and HTML template customizations. See [Themes](#) for details.
 - b. Create one or more extensions to implement any existing JavaScript, SuiteScript, or JSON customizations. See [Extensions](#) and [Extensibility API](#) for details.
4. All customizations should be implemented using themes and extensions. However, if your existing customizations include objects not accessible using the Extensibility API, migrate these customizations to the new source code directory.
 - a. Create a duplicate of your custom module folder in the new source code directory.



Note: This applies to custom modules that are not standard SCA source modules. Any source SCA modules that have been customized may require comparison against the new version plus any required manual edits to the code.

- b. Copy your existing custom modules into the new custom module folder.

- c. Update the new distro.json file to include references to any custom modules and to define any custom application dependencies.



Important: Do not copy the existing distro.json file contents directly into the new distro.json file. Doing this causes overwrites to changes made in the distro.json file to support the new release. For example, a new module may be referenced. NetSuite recommends evaluating the DIFF between files to ensure that you migrate all changes from your local distro.json file into the new distro.json file.

- d. Any module that changed in the new release bears a higher version number from its previous release. Compare module version numbers to assess which modules have been updated and migrate the changes to any existing customizations as necessary. Refer to the [SuiteCommerce Release Notes](#) for details on which modules were changed and why.



Important: Migrating to a new release requires testing to ensure any existing customizations remain viable. Evaluate differences in code against your existing customizations and test thoroughly before deploying to your live site. See [Best Practices for Customizing SuiteCommerce Advanced](#).

5. Link your site record and domain to an SSP application as required. This can be the SSP Application for the latest version of SCA or one specifically intended for development.
 - a. Go to Setup > SuiteCommerce Advanced > SSP Applications and click **View** next to the SuiteCommerce Advanced Dev <version> application.
 - b. Click **Link to Domain**.
 - c. Select the development domain for this SSP Application.
6. Deploy your local files and test appropriately.

Testing may consist of various scenarios depending on your setup. You can deploy your local files to any of the following locations:

- Themes and extensions require the theme or extension developer tools. These are included with the SuiteCommerce Extension Management bundle. This requires setting up the appropriate developer environment. See [Theme Developer Tools](#) and [Extension Developer Tools](#) for details.
- Core source code customizations require the core SCA developer tools. This requires setting up the appropriate developer environment. See [Core SuiteCommerce Advanced Developer Tools](#) for details. Core SCA customizations can be deployed to:
 - A primary SSP Application linked to a primary domain (live site)
 - Development SSP Application linked to a development domain
 - Local server. See [SCA on a Local Server](#).
 - NetSuite sandbox account. See [Deploy to a NetSuite Sandbox](#).
 - Custom SSP Application. See [Deploy to a Custom SSP Application](#)

For example, you can deploy your customizations to a local server or to a NetSuite sandbox account for testing. If you do not have a Sandbox account, you can deploy to an SSP Application linked to a development domain in NetSuite. After testing, deploy your files to the SSP Application linked to the primary domain.