Shopware 6 Storefront-Templates

Shopware 6 Storefront-Templates

Shopware 6: Templates

Twig templates in the Shopware 6 storefront

shopware/platform

Shopware 6 is an open source eCommerce platform realised by the ideas and the spirit of its community. – shopware/platform

[expand title="mehr lesen…"]

Templates

The storefront theme is using Boostrap. Therefore the template structure is a derivative of the bootstrap starter template. The templating engine used is Twig. For styling SASS is used as CSS preprocessor. The bundling and transpiling of the javascript is done with Webpack.

The templates can be found in /src/Storefront/Resources/views/storefront/

Template Top Level

<platform/src/Storefront/Resources/views/storefront/>
└── block
└── component
└── element
└── layout
└── page
└── section
└── utilities
└── base.html.twig

block, element, sectionParts of the experience worldscomponentShared content templates form the basis of the pages.layoutLayout templates. Navigation, header and footer content templates are located here.pageThe concrete templates rendered by the page controllers. This directory contains full page templates as well as private local includes and the pagelet ajax response templates if necessary. utilitiesTechnical necessities used across the content and across all domain concepts.base.html.twigBase page layout of the storefront. This file mainly includes header and footer templates from /layout and provides blocks for the /page templates to overwrite.

Page templates

The page directory contains the entry points which are referenced by page controllers and rendered through the Twig engine. The structure is derived from the page controller naming.

```
<platform/src/Storefront/Resources/views/storefront/page>
└── account
└── checkout
└── content
└── error
└── newsletter
└── product-detail
└── search
```

Inside the directories are the actual templates rendered by the storefront. The inner structure depends on the domain context.

Override and extend templates

Due to the plugin and theme system in shopware it is possible that one storefront template gets extended by multiple plugins or themes, but Twig does not allow multi inheritance out of the box. Therefore we created custom twig functions sw_extends and sw_include, that work like twigs native extends or include, except that they allow for multi inheritance. So it is really important to use the sw_extends and sw_include, instead of the native extends and include.

sw_extends

To inherit a template file, you need to use {% sw_extends %}.

Example:

```
{#
YourPlugin/Resources/views/storefront/layout/header/logo.html.
twig #}
```

```
{%                                                  sw_extends
'@Storefront/storefront/layout/header/logo.html.twig' %}

{# Add an <h2> with underneath the logo image block #}
{% block layout_header_logo_image %}
    {{ parent() }}
    <h2>Additional headline</h2>
{% endblock %}
```

sw_include

If you build your own feature and you need e.g. an element to display the price of the current product you can include existing partials with sw_include like this.

Example:

```
{#     MyPlugin/Resources/views/storefront/page/product-
detail/index.html.twig #}

<div class="my-theme an-alternative-product-view">
    ...

    {% block component_product_box_price %}
        {# use sw_include to include template partials #}
                                                {%     sw_include
'@Storefront/storefront/component/product/card/price-
unit.html.twig' %}
    {% endblock %}

    ...
</div>
```

Inheritance order

The order of the inheritance is determined by the order you set in the theme.json of your active theme.

Styles Top Level

The stylesheets are written in SASS. The organization is

inspired by the 7-1 pattern structure.

```
<platform/src/Storefront/Resources/app/storefront/src/scss>
└── abstract
└── base
└── component
└── layout
└── page
└── skin
└── vendor
└── base.scss
```

The base.scss is the global include file which references styles that are written as an extension of the bootstrap base. For further information just take a look at the excellent description at sass-guidelines.es.

Scripts Top Level

The storefront includes a set of JavaScript plugins providing different functionalities to the storefronts templates on the client side. The plugins are written as **ES6 classes** in **vanilla JavaScript**. Additionally since bootstrap is distributed with the jQuery library the storefront also contains this library.

The script root looks like this:

```
<platform/src/Storefront/Resources/app/storefront/src/script>
└── config
└── helper
└── plugin
└── service
└── utility
└── vendor
└── base.js
```

Sanitize content

Filters tags and attributes from a given string.

The filter can be found in /src/Storefront/Framework/Twig/Extension/SwSanitizeTwigFilter.phpExamples:{{ unfilteredHTML|sw_sanitize }}Uses the default config{{ unfilteredHTML|sw_sanitize({'div': ['style', ...]}, true) }}**allow only** div tags + style attribute for div

[/expand]

---

sass

sass

Sass: Syntactically Awesome Style Sheets

Syntactically Awesome Style Sheets

[expand title="mehr lesen…"]

[/expand]

Sales channel –

Verkaufskanäle – Forum-slack – 23-10-2020

Shopware 6 Sales channel – Verkaufskanäle – Forum-slack – 23-10-2020

[expand title="mehr lesen…"]

Hendrik11:20 Uhr

What is the best way to create a SalesChannelContext, eg. when using a CLI command? I loaded a SalesChannelEntity and used SalesChannelContext::createFrom($loadedSalesChannelEntity), but the SalesChannelContext generated this way is missing the generic Context and doesn't contain a setter to add one.
What is the best way to add it there?

there is a sales channel context factory service

```
$salesChannelContext                                    =
$this->salesChannelContextFactory->create(
    Uuid::randomHex(),
    $bid->getSalesChannel()->getId(),
    [
            SalesChannelContextService::LANGUAGE_ID =>
$bid->getLanguage()->getId(),
            SalesChannelContextService::CUSTOMER_ID =>
$bid->getCustomer()->getId(),
             SalesChannelContextService::COUNTRY_ID =>
$bid->getCustomer()->getDefaultBillingAddress()->getCountry()-
>getId()
    ]
);
```

[/expand]

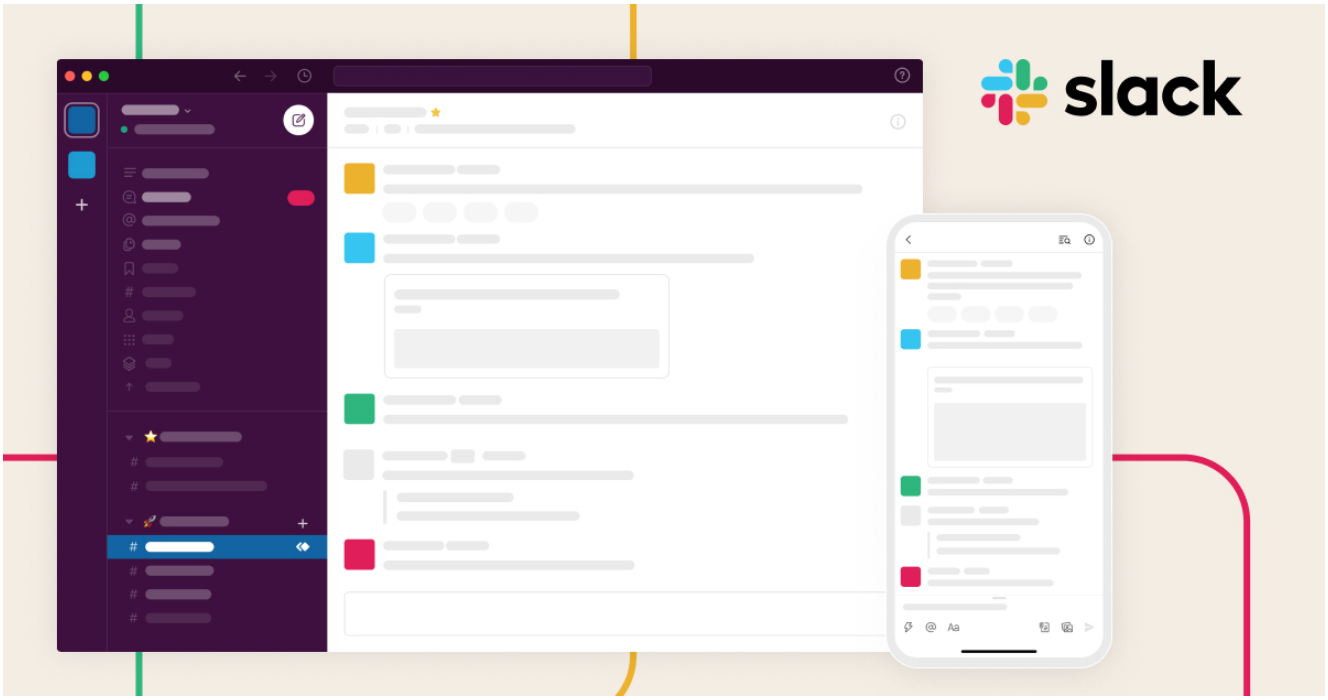
---

roles and permissions – ACL – https://slack.shopware.com/

roles and permissions – ACL – https://slack.shopware.com/

[expand title="mehr lesen…"]

Hannes Wernery 13:40 Uhr
Hey! The ACL system is supposed to be upwards and downwards compatible https://hi.shopware.com/ACL
But as I see it, if I activate this feature and set up a role for a user, any models that are not listed in the role/permission management are *not allowed*. Meaning: entities that are not explicitly listed and activated in the management cannot be used by that user. This requires changes in our plugin.
code: „0“
detail: „Missing privilege my_custom_model:read“
meta: {,…}
status: „403“
title: „Forbidden“
This is not quite downwards compatible, is it? Or am I missing something?



Join Shopware Community

Slack is where work flows. It’s where the people you need, the information you share, and the tools you use come together to get things done.
[Meteor] Brent Robert10:55 UhrWe seem to be having an issue with a template of the B2B suite:This is default shopware:

```
{% block   page_checkout_confirm_address_billing_actions_link
%}
<a   href="{{   path('frontend.account.address.edit.page',
{'addressId': billingAddress.id}) }}"
title="{{ "account.overviewChangeBilling"|trans|striptags }}"
class="btn btn-light"
data-address-editor="true"
data-address-editor-options='{{
addressEditorOptions|json_encode }}'>
{{ "account.overviewChangeBilling"|trans|sw_sanitize }}
</a>
{% endblock %}
```

This is B2B suite:

```
{% block   page_checkout_confirm_address_shipping_actions_link
%}
{% if b2bSuite %}
```

```
<a
title="{{ "account.overviewChangeShipping"|trans|striptags }}"
class="btn     btn-light     ajax-panel-link     {{
b2b_acl('b2bcontactaddress', 'list') }}"
data-target="address-select"
href="{{ path('frontend.b2b.b2baddressselect.index', {'type':
'shipping', 'selectedId': shippingAddress.id}) }}"
>
{{ "account.overviewChangeShipping"|trans|sw_sanitize }}
</a>
{%  else %}
{{ parent() }}
{%  endif %}
{% endblock %}
```

The B2B suite is losing the data-addresss-editor(-options) tags and breaking the layout?

[/expand]

---

roles and permissions – november 2020

WHRGDr0pQG5E%253D

[expand title="mehr lesen…"]

Roles & Permissions is planned for NovemberPrepare your plugin now:

In November we plan to release the highly demanded feature „Roles & Permissions“ (ACL). With this feature you will be able to assign rights and roles to users in the administration.

Prepare your plugins:
In order for store operators to be able to use the feature „Roles & Permissions“ to its full extent, you should prepare the rights in your plugins now.

What you have to do for that?
The feature „Roles & Permissions“ is already completely implemented on the master (GitHub) and can be tested by activating a feature flag. A detailed instruction on how to activate the feature flag can be found below in the mailing.

Important to know:
All customizations are upward and downward compatible. That means you can make the changes now and your plugin is still compatible in older Shopware 6 versions and of course in future versions.

Documentation
Activate feature flagDevelopment Template

If the Development Template is used, the feature flag can be activated as follows:

1.) The following code should be integrated in the *.psh.yaml.override*

Check out the Code on GitHub

2.) After that the cache should be cleared with the following command:

./psh.phar cache

Activate feature flagProduction Template and all other installation types

For all other types of installation, proceed as follows:

1.) An environment variable should be created with the following values:

FEATURE_NEXT_3722=1
You can also use e.g. the .env file.

2.) The cache should be cleared with the following command:

bin/console cache:clear

1000 +GitHub Stars

800,000 +Downloads

2000 +Developers on Slack

shopware AG
Ebbinghoff 10
Schöppingen

DE: +49 (0) 2555 928850
UK: +44 (0) 203 095 2445
World: 00 800 746 7626 0
info@shopware.com

Board: Stefan Hamann, Sebastian Hamann
Supervisory Board: Reinhold Wellers (Chairman), Christoph Hertz, Christoph Pliete
Register Court: Amtsgericht Coesfeld HRB 11471

[/expand]

Shopware 6 Theme entwickeln – Livestream #4

Shopware 6 Theme entwickeln – Livestream #4

[expand title="mehr lesen…"]

Your browser does not support HTML5 video.

[/expand]

Shopware 6 Theme entwickeln – Livestream #3

Shopware 6 Theme entwickeln – Livestream #3

[expand title="mehr lesen…"]

Your browser does not support HTML5 video.

[/expand]

Shopware 6 Theme entwickeln – Community store/Compile a theme

Shopware 6 Theme entwickeln – Community store/Compile a theme

[expand title="mehr lesen…"]

Community store

To publish your plugins in the Shopware Community Store, you need to register your own developer prefix in your Shopware account.

After that process you can compile your created theme, then create a zip file of the folder and upload your theme to the Shopware Community Store.

Compile a theme

```
# run this to compile your theme
$ bin/console theme:compile
```

[/expand]

Shopware 6 Theme entwickeln –

JavaScript

Shopware 6 Theme entwickeln – Shopware 6 Theme entwickeln – JavaScript

[expand title="mehr lesen…"]

You can add JavaScript to theme to interact with the DOM, make some API calls or change the behavior of the storefront.

By default, Shopware 6 look inside the <plugin root>/src/Resources/app/storefront/src folder of your plugin to load a main.js file.

You can simple put your own JavaScript code in here. Shopware 6 support the ECMAScript 6 and will transpile your code to ES5 for legacy browser support.

The recommended way to add JavaScript is to write a JavaScript Plugin for the storefront.

Writing a JavaScript plugin

Storefront JavaScript plugins are vanilla JavaScript ES6 classes that extend from our Plugin base class. For more background information on JavaScript classes, take a look here. To get started create a src/Resources/app/storefront/src/example-plugin folder and put an example-plugin.plugin.js file in there. Inside that file create and export a ExamplePlugin class that extends the base Plugin class:

```
//  src/Resources/app/storefront/src/example-plugin/example-plugin.plugin.js

import Plugin from 'src/plugin-system/plugin.class';

export default class ExamplePlugin extends Plugin {
}
```

Each Plugin has to implement the init() method. This method will be called when your plugin gets initialized and is the entrypoint to your custom logic. In your case you add an callback to the onScroll event from the window and check if the user has scrolled to the bottom of the page. If so we display an alert. Your full plugin now looks like this:

```
//  src/Resources/app/storefront/src/example-plugin/example-plugin.plugin.js

import Plugin from 'src/plugin-system/plugin.class';

export default class ExamplePlugin extends Plugin {
    init() {
        window.onscroll = function() {
             if ((window.innerHeight + window.pageYOffset) >= document.body.offsetHeight) {
                 alert('seems like there\'s nothing more to see here.');
             }
        };
    }
}
```

Registering your plugin

Next you have to tell Shopware that your plugin should be loaded and executed. Therefore you have to register your plugin in the PluginManager. Open up main.js file inside your src/Resources/app/storefront/src folder and add the following example code to register your plugin in teh PluginManager.

```
// src/Resources/app/storefront/src/main.js

// import all necessary storefront plugins
import ExamplePlugin from './example-plugin/example-plugin.plugin';

// register them via the existing PluginManager
const PluginManager = window.PluginManager;
PluginManager.register('ExamplePlugin', ExamplePlugin);
```

You also can bind your plugin to an DOM element by providing an css selector:

```
// src/Resources/app/storefront/src/main.js

// import all necessary storefront plugins
import ExamplePlugin from './example-plugin/example-plugin.plugin';

// register them via the existing PluginManager
const PluginManager = window.PluginManager;
PluginManager.register('ExamplePlugin', ExamplePlugin, '[data-example-plugin]');
```

In this case the plugin just gets executed if the HTML document contains at least one element with the data-scroll-detector attribute. You could use this.el inside your plugin to access the DOM element your plugin is bound to.

Loading your plugin

You bound your plugin to the css selector „[data-example-plugin]“ so you have to add DOM elements with this attribute on the pages you want your plugin to be active. Therefore create an Resources/views/storefront/page/content/ folder and create an index.html.twig template. Inside this template extend from the @Storefront/storefront/page/content/index.html.twig and overwrite the base_main_inner block. After the parent content of the blog add an template tag that has the data-example-

plugin attribute. For more information on how template extension works, take a look here.

```
{% sw_extends '@Storefront/storefront/page/content/index.html.twig' %}

{% block base_main_inner %}
    {{ parent() }}

    <template data-example-plugin></template>
{% endblock %}
```

With this template extension your plugin is active on every content page, like the homepage or category listing pages.

More about Storefront JavaScript: Storefront JavaScript reference documentation

[/expand]

Shopware 6 Theme entwickeln – SCSS and Styling

Shopware 6 Theme entwickeln – SCSS and Styling

Shopware 6: SCSS and Styling

In our documentation you will find all the information you

need for your daily work with Shopware.

[expand title="mehr lesen…"]

The stylesheets are written in SASS. The folder structure is inspired by the 7-1 pattern structure.

```
<platform/src/Storefront/Resources/app/storefront/src/scss>
└── abstract
└── base
└── component
└── layout
└── page
└── skin
└── vendor
└── base.scss
```

Shopware 6 looks inside your theme.json file to find a „style“ array which contains all SCSS files which should be loaded by your theme. By default you get the Shopware Storefront SCSS plus an additional entry point for your own SCSS. You can also extend this array with more SCSS files.

```
{
  "name": "Just another theme",

  ...

  "style": [
    "@Storefront",
    "app/storefront/src/scss/base.scss" <-- Theme SCSS entry
  ],

  ...

}
```

To try it out, open up the base.scss file from your theme.

Inside of the .scss file, add some styles like this:

```
// src/Resources/app/storefront/src/scss/base.scss
```

```
body {
    background: blue;
}
```

To see if it’s working you have to re-build the storefront. Use the following command to do that.

```
# run this to re-compile the current storefront theme
$ ./psh.phar storefront:build

# or run this to start a watch-mode (the storefront will re-compile when you make sytle changes)
$ ./psh.phar storefront:hot-proxy
```

In this example, the background of the body will be changed.

Working with variables

In case you want to use the same color in several places, but want to define it just one time you can use variables for this.

Create a abstract/variables.scss file inside your „scss“ folder and define your background color variable.

```
// in variables.scss
$sw-storefront-assets-color-background: blue;
```

Inside your base.scss file you can now import your previously defined variables and use them:

```
// in base.scss
@import 'abstract/variables.scss';

body {
    background: $sw-storefront-assets-color-background;
}
```

This has the advantage that when you want to change the values of your variables you just have one location to change them and the hard coded values are not cluttered all over the codebase.

Bootstrap

The storefront theme is implemented as a skin on top of the Boostrap toolkit.

Override default SCSS variables

To override default variables like for example $border-radius from Bootstrap you should use a slightly different approach then explained in the storefront assets how-to.

Bootstrap 4 is using the !default flag for it’s own default variables. Variable overrides have to be declared beforehand.

More information: https://getbootstrap.com/docs/4.0/getting-started/theming/#variable-defaults

To be able to override Bootstrap variables you can define an additional SCSS entry point in your theme.json which is declared before @Storefront. This entry point is called overrides.scss:

```
{
  "name": "Just another theme",
  "author": "Just another author",
  "views": [
     "@Storefront",
     "@Plugins",
     "@JustAnotherTheme"
  ],
  "style": [
     "app/storefront/src/scss/overrides.scss", <-- Variable overrides
    "@Storefront",
    "app/storefront/src/scss/base.scss"
  ],
  "script": [
    "@Storefront",
    "app/storefront/dist/storefront/js/just-another-theme.js"
```

```
  ],
  "asset": [
    "app/storefront/src/assets"
  ]
}
```

In the overrides.scss you can now override default variables like $border-radius globally:

```
/*
Override variable defaults
==========================================================
This file is used to override default SCSS variables from the Shopware Storefront or Bootstrap.

Because of the !default flags, theme variable overrides have to be declared beforehand.
https://getbootstrap.com/docs/4.0/getting-started/theming/#variable-defaults
*/

$disabled-btn-bg: #f00;
$disabled-btn-border-color: #fc8;
$font-weight-semibold: 300;
$border-radius: 0;
$icon-base-color: #f00;
$modal-backdrop-bg: rgba(255, 0, 0, 0.5);
```

Please only add variable overrides in this file. You should not write CSS code like .container { background: #f00 } in this file. When running storefront:hot or storefront:hot-proxy SCSS variables will be injected dynamically by webpack. When writing selectors and properties in the overrides.scss the code can appear multiple times in your built CSS.

Using Bootstrap SCSS only

The Shopware default theme is using Bootstrap with additional custom styling.

If you want to build your theme only upon the Bootstrap SCSS you can use the @StorefrontBootstrap placeholder instead of the @Storefront bundle in the style section of your theme.json. This gives you the ability to use the Bootstrap SCSS without the Shopware Storefront „skin“. Therefore all the SCSS from src/Storefront/Resources/app/storefront/src/scss/skin wil l not be available in your theme.

```
{
  "style": [
    "@StorefrontBootstrap",
    "app/storefront/src/scss/base.scss"
  ]
}
```

- This option can only be used in the style section of the theme.json. You must not use it in views or script.
- All theme variables like $sw-color-brand-primary are also available when using the Bootstrap option.
- You can only use either @StorefrontBootstrap or @Storefront. They should not be used at the same time. The @Storefront bundle **includes** the Bootstrap SCSS already.

[/expand]