

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 the 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 `parent::index.html.twig` 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](#).

```
{% 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 [Bootstrap 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/#var
iable-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 will 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]

Shopware 6 Theme entwickeln – Snippets

Shopware 6 Theme entwickeln – Snippets

```
[expand title="mehr lesen..."]
```

To extend a language in Shopware 6 you can add your own snippets in your theme. You can also add a completely new language to Shopware 6.

General snippet structure

To organize your snippets you can add them to .json files, so structuring and finding snippets you want to change is very easy.

Adding snippets

You can add new snippets by adding files to the follow structure like this:

```
# move into your theme folder
$ cd custom/plugins/MyTheme
```

```
# structure of theme
└─ src
    └─ Resources
```

```

├── config
│   └── services.xml
├── snippet
│   ├── de_DE
│   │   ├── SnippetFile_de_DE.php
│   │   └── storefront.de-DE.json
│   └── en_GB
│       ├── SnippetFile_en_GB.php
│       └── storefront.en-GB.json
└── MyTheme.php

```

There is no explicit syntax for variables in the storefront. It is nevertheless recommended to encompass them with % symbols to be extra clear on what their purpose is. Pluralization works for any natural number. Just remember to explicitly define the intervals' amounts and ranges for that snippet.

Example of src/Resources/snippet/storefront.en-GB.json:

```

{
  "my-theme": {
    "productDetail": {
      "headLineText": "There are %count% discounts
available for %product%:",
      "description": "Lorem ipsum dolor sit amet,
consetetur sadipscing elitr, sed diam ..."
    },
    "cart": {
      "itemCounter": "{1} 1 discount item | ]1,Inf[
%count% discount items"
    }
  }
}

```

Adding Storefront snippets

Storefront snippets additionally require a class that extends the SnippetFileInterface. A suitable name would e.g. be SnippetFile_en_GB.php. Having created that file, you will

have to implement the following five methods:

- **getName:** Returns the name of the snippet file as a string. By referring to this name, you can access the translations later. It is **required** to use `messages.en-GB`, if you provide a whole new language. By default, an extension should call its Storefront extension `storefront.en-GB`. Otherwise a describing domain, like shopware's PayPal plugin using `paypal.en-GB`, is also okay.
- **getPath:** Each `SnippetFile` class has to point to the `.json` file, that contains the actual translations. Return its path here. We suggest using the name already chosen in `getName` for your file name.
- **getIso:** Return the ISO string of the supported locale here. This is important, because the Translator collects every snippet file with this locale and merges them to generate the snippet catalogue used by the storefront.
- **getAuthor:** Return your vendor name here. This can be used to distinguish your snippets from all the other available ones. The Administration snippet set module offers a filter, so users are able to easily find plugin specific snippets.
- **isBase:** Return `true` here, if your theme implements a whole new language, such as providing french snippets for the whole Shopware 6 system. Don't forget to watch your `getName` method then. Most of the time, you're just adding your own snippets to an existent language, then `false` will be your way to go.

Example for the language `en-GB` :

```
<?php declare(strict_types=1);
```

```
namespace MyTheme\Resources\snippet\en_GB;
```

```
use Shopware\Core\System\Snippet\Files\SnippetFileInterface;
```

```
class SnippetFile_en_GB implements SnippetFileInterface
```

```

{
    public function getName(): string
    {
        return 'storefront.en-GB';
    }

    public function getPath(): string
    {
        return __DIR__ . '/storefront.en-GB.json';
    }

    public function getIso(): string
    {
        return 'en-GB';
    }

    public function getAuthor(): string
    {
        return 'Enter developer name here';
    }

    public function isBase(): bool
    {
        return false;
    }
}

```

Registering your service

Lastly you have to register your snippet files by creating file `src/Resources/config/services.xml`.

Example:

```

<!-- src/Resources/config/services.xml -->
<?xml version="1.0" ?>

<container xmlns="http://symfony.com/schema/dic/services"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://symfony.com/schema/dic/services
    http://symfony.com/schema/dic/services/services-1.0.xsd">

```

```

<!-- Translations -->
<services>
    <service
id="MyTheme\Resources\snippet\en_GB\SnippetFile_en_GB"
public="true">
        <tag name="shopware.snippet.file"/>
    </service>
</services>

<services>
    <service
id="MyTheme\Resources\snippet\de_DE\SnippetFile_de_DE"
public="true">
        <tag name="shopware.snippet.file"/>
    </service>
</services>
</container>

```

Using your snippets in your templates

With the trans filter you can use your snippets in the twig templates and they will be translated automatically. You also can pass values to replace the placeholders in the snippets.

```

<div class="product-detail-headline">
    {{ 'my-theme.productDetail.headLineText' |
trans({'%count%': count, '%product%': product}) }}
</div>

```

[/expand]

**Shopware 6 Theme entwickeln –
Livestream – theme – github**

**Shopware 6 Theme entwickeln –
Livestream – theme – github**



8mylez

Shopware Specialists – Support / Hosting / Themes / Plugins /
Blog / Tutorials – 8mylez



8mylez Developer

Hallo und herzlich willkommen auf unserem neuen YouTube Kanal:

8mylez Developer. Wir sind 8mylez, eine reine Shopware Agentur und wollen hier über die techni...

[expand title="mehr lesen..."]

[/expand]

Shopware 6 Theme entwickeln – Twig templates

Shopware 6 Theme entwickeln – Twig templates

[expand title="mehr lesen..."]

In Shopware 6 we use the [Twig](#) template engine to render the HTML in the storefront.

The templates can be found in [platform/src/Storefront/Resources/views/storefront/](#) Here is a brief overview of the structure and the most important template sections:

```
#
vendor/shopware/platform/src/Storefront/Resources/views/storefront
├── base.html.twig
├── block                # Part of the content management system
```

- cms-block-category-navigation.html.twig
 - cms-block-center-text.html.twig
 - cms-block-form.html.twig
 - ...
- element # Part of the content management system
 - cms-element-category-navigation.html.twig
 - cms-element-form
 - cms-element-form.html.twig
 - ...
- section # Part of the content management system
 - cms-section-block-container.html.twig
 - cms-section-default.html.twig
 - cms-section-sidebar.html.twig
- component # Shared content templates form the basis of the pages.
 - account
 - address
 - analytics.html.twig
 - checkout
 - ...
- layout # Layout templates. Navigation, header and footer content templates are located here.
 - breadcrumb.html.twig
 - cookie
 - footer
 - header
 - ...
- page # The 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.
 - account
 - checkout
 - content
 - error
 - newsletter

```
|
| ── product-detail
| ── search
| ── sitemap
|
└─ utilities    # Technical necessities used across the
content, header and footer sections across all domain
concepts.
    ── alert.html.twig
    ── form-violation.html.twig
    ── icon.html.twig
    ── ...
```

The `base.html.twig` is the root file of the storefront which holds every rendered component.

So, for instance, if you would like to modify the header, you would want to recreate the specific directory structure in order to be able to overwrite or extend the already existing elements. The storefront header in the `header.html.twig` file (which is later included into the `base.html.twig`) is located inside
the `platform/src/Storefront/Resources/views/storefront/layout/header` directory.

Blocks

Almost every HTML chunk is wrapped into a corresponding Twig-Block, so it is easy to overwrite or extend existing blocks.

Adding new content by extending the template

In this example we want add some extra content like the shop name below the logo. To do so, you need to extend the existing `logo.html.twig` file which is located inside the `platform/src/Storefront/Resources/views/storefront/layout/header` directory.

First we create a new file inside our theme.

```
# move into your theme folder
$ cd custom/plugins/MyTheme
```

```
# create coresponding folder structure
$ mkdir -p src/Resources/app/storefront/views/layout/header
```

```
# create a new file
$ touch
src/Resources/app/storefront/views/layout/header/logo.html.twig
```

Then we extend the existing file through the `sw_extends` command.

```
#
src/Resources/app/storefront/views/layout/header/logo.html.twig
```

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

After that you save this new file. Everything should be rendered like before.

To add new content into to template you can override the `layout_header_logo_link` block from the `logo.html.twig` like this.

```
#
src/Resources/app/storefront/views/layout/header/logo.html.twig
```

```
{# extend the original twig file #}
{% sw_extends
'@Storefront/storefront/layout/header/logo.html.twig' %}
```

```
{# override the original twig block #}
{% block layout_header_logo_link %}
```

```
{#
```

```

        call the `parent` function to keep the old behavior,
        otherwise the block gets overridden
    #}
    {{ parent() }}

    {# modified content added to the block #}
    <span>Company name of shop owner</span>

{% endblock %}

```

Reuse of existing elements

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.

```

<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>

```

Template multi inheritance

Due to the plugin and theme system in Shopware 6 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 our own twig functions `sw_extends` and `sw_include`, that work like twigs native [extends](#) or [include](#), except that they allow multi inheritance. So it is really important to use the `sw_extends` and `sw_include`, instead of the

native extends and include.

[/expand]

Shopware 6 Theme entwickeln – Theme configuration

Shopware 6 Theme entwickeln – Theme configuration

[expand title="mehr lesen..."]

Structure of the theme.json

Open up the `src/Resources/theme.json` file with your favorite code-editor. The configuration looks like this.

```
# src/Resources/theme.json
{
  "name": "MyTheme",
  "author": "Shopware AG",
  "views": [
    "@Storefront",
    "@Plugins",
    "@MyTheme"
  ],
  "style": [
    "app/storefront/src/scss/overrides.scss",
    "@Storefront",
```

```

    "app/storefront/src/scss/base.scss"
  ],
  "script": [
    "@Storefront",
    "app/storefront/dist/storefront/js/my-theme.js"
  ],
  "asset": [
    "app/storefront/src/assets"
  ]
}

```

Installing Let's have a closer look at each section.

```

# src/Resources/theme.json
{
  "name": "MyTheme",
  "author": "Shopware AG",
  "description": {
    "en-GB": "Just another description",
    "de-DE": "Nur eine weitere Beschreibung"
  },
  ...
}

```

Here change the name of your theme and the author. The description section is optional and as you notice it is also translatable.

Theme template inheritance

The inheritance of the templates can be controlled via the views option. Here you define the order in which the templates are to be loaded. To illustrate this, here is an example for the views configuration:

```

# src/Resources/theme.json
{
  ...
  "views": [
    "@Storefront",
    "@Plugins",

```

```

        "@PayPal",
        "@MyTheme"
    ],
    ...
}

```

Defining the above configuration results in the following behavior:

- Templates are first searched in @MyTheme.
- The specification of @PayPal allows to control the order for a specific plugin more precisely
- @Plugins serves as a placeholder and defines that the @MyTheme and @PayPal should be searched for in all other plugins for the templates
- @Storefront then defines that the Shopware 6 storefront theme should be used as the last inheritance level.

Styles

```

# src/Resources/theme.json
{
    ...
    "style": [
        "app/storefront/src/scss/overrides.scss",
        "@Storefront",
        "app/storefront/src/scss/base.scss"
    ],
    ...
}

```

The style section determines the order of the CSS compilation. In the app/storefront/src/scss/base.scss file you can apply your changes you want to make to the @Storefront standard styles or add other styles you need. The app/storefront/src/scss/overrides.scss file is used for a special case. Maybe you need to override some defined variables or functions defined by Shopware or Boostaop, you can implement your changes here.

Assets

```
# src/Resources/theme.json
{
  ...
  "asset": [
    "app/storefront/src/assets"
  ]
  ...
}
```

The asset option you can configure you paths to your assets like images, fonts, etc. The standard location to put your assets to is the app/storefront/src/assets folder.

Config fields

One of the benefits of creating a theme is that you can overwrite the theme configuration of the default theme or add your own configurations.

```
# src/Resources/theme.json
{
  ...
  "assest": [
    ...
  ],
  "config": {
    "fields": {
      "sw-color-brand-primary": {
        "value": "#00ff00"
      }
    }
  }
  ...
}
```

In the example above, we change the primary color to green. You always inherit from the storefront config and both configurations are merged. This also means that you only have

to provide the values you actually want to change. You can find a more detailed explanation of the configuration inheritance in the next section.

The `theme.json` contains a `config` property which consists a list of tabs, blocks, sections and fields.

The key of each config fields item is also the technical name which you use to access the config option in your theme or scss files. config entries will show up in the administration and can be customized by the enduser (if `editable` is set to `true`, see table below).

The following parameters can be defined for a config field item:

Name	Meaning
label	Array of translations with locale code as key
type	Type of the config. Possible values: color, text, number, fontFamily, media, checkbox and switch
editable	If set to false, the config option will not be displayed (e.g. in the administration)
tab	Name of a tab to organize the config options
block	Name of a block to organize the config options
section	Name of a section to organize the config options
custom	The defined data will not be processed but is available via API
scss	If set to false, the config option will not be injected as a SCSS variable

Field types

You can use different field types in your theme manager:

- A text field example:

```
"modal-padding": {
```

```
"label": {
  "en-GB": "Modal padding",
  "de-DE": "Modal Innenabstand"
},
"type": "text",
"value": "(0, 0, 0, 0)",
"editable": true
}
```

A number field example:

```
"visible-slides": {
  "label": {
    "en-GB": "Number of visible slides",
    "de-DE": "Anzahl an sichtbaren Slider Bildern"
  },
  "type": "number",
  "custom": {
    "numberType": "int",
    "min": 1,
    "max": 6
  },
  "value": 3,
  "editable": true
}
```

Two boolean field examples:

```
"navigation-fixed": {
  "label": {
    "en-GB": "Fix navigation",
    "de-DE": "Navigation fixieren"
  },
  "type": "switch",
  "value": true,
  "editable": true
}
```

or

```
"navigation-fixed": {
  "label": {
    "en-GB": "Fix navigation",
```

```

    "de-DE": "Navigation fixieren"
  },
  "type": "checkbox",
  "value": true,
  "editable": true
}

```

Examples for custom config fields

- A custom single-select field example

```

{
  "name": "Just another theme",
  "author": "Just another author",
  "description": {
    "en-GB": "Just another description",
    "de-DE": "Nur eine weitere Beschreibung"
  },
  "views": [
    "@Storefront",
    "@Plugins",
    "@SelectExample"
  ],
  "style": [
    "app/storefront/src/scss/overrides.scss",
    "@Storefront",
    "app/storefront/src/scss/base.scss"
  ],
  "script": [
    "@Storefront",
    "app/storefront/dist/storefront/js/select-example.js"
  ],
  "asset": [
    "app/storefront/src/assets"
  ],
  "config": {
    "blocks": {
      "exampleBlock": {
        "label": {
          "en-GB": "Example block",
          "de-DE": "Beispiel Block"
        }
      }
    }
  }
}

```

```

    }
  },
  "sections": {
    "exampleSection": {
      "label": {
        "en-GB": "Example section",
        "de-DE": "Beispiel Sektion"
      }
    }
  },
  "fields": {
    "my-single-select-field": {
      "label": {
        "en-GB": "Select a font size",
        "de-DE": "Wähle ein Schriftgröße"
      },
      "type": "text",
      "value": "24",
      "custom": {
        "componentName": "sw-single-select",
        "options": [
          {
            "value": "16",
            "label": {
              "en-GB": "16px",
              "de-DE": "16px"
            }
          },
          {
            "value": "20",
            "label": {
              "en-GB": "20px",
              "de-DE": "20px"
            }
          },
          {
            "value": "24",
            "label": {
              "en-GB": "24px",
              "de-DE": "24px"
            }
          }
        ]
      }
    }
  }
}

```

```

        }
    ]
},
"editable": true,
"block": "exampleBlock",
"section": "exampleSection"
}
}
}
}
}

```



A custom multi-select field example

```

{
  "name": "Just another theme",
  "author": "Just another author",
  "description": {
    "en-GB": "Just another description",
    "de-DE": "Nur eine weitere Beschreibung"
  },
  "views": [
    "@Storefront",
    "@Plugins",
    "@SelectExample"
  ],
  "style": [
    "app/storefront/src/scss/overrides.scss",
    "@Storefront",
    "app/storefront/src/scss/base.scss"
  ],
  "script": [
    "@Storefront",
    "app/storefront/dist/storefront/js/select-example.js"
  ],
  "asset": [
    "app/storefront/src/assets"
  ],
  "config": {
    "blocks": {
      "exampleBlock": {
        "label": {

```

```

        "en-GB": "Example block",
        "de-DE": "Beispiel Block"
    }
}
},
"sections": {
    "exampleSection": {
        "label": {
            "en-GB": "Example section",
            "de-DE": "Beispiel Sektion"
        }
    }
},
"fields": {
    "my-multi-select-field": {
        "label": {
            "en-GB": "Select some colours",
            "de-DE": "Wähle Farben aus"
        },
        "type": "text",
        "editable": true,
        "value": [
            "green",
            "blue"
        ],
        "custom": {
            "componentName": "sw-multi-select",
            "options": [
                {
                    "value": "green",
                    "label": {
                        "en-GB": "green",
                        "de-DE": "grün"
                    }
                },
                {
                    "value": "red",
                    "label": {
                        "en-GB": "red",
                        "de-DE": "rot"
                    }
                }
            ]
        }
    }
}

```

```

        },
        {
            "value": "blue",
            "label": {
                "en-GB": "blue",
                "de-DE": "blau"
            }
        },
        {
            "value": "yellow",
            "label": {
                "en-GB": "yellow",
                "de-DE": "gelb"
            }
        }
    ]
},
"block": "exampleBlock",
"section": "exampleSection"
}
}
}
}
}

```



Tabs, blocks and sections

You can use tabs, blocks and sections to structure and group the config options.



In the picture above are four tabs. In the „Colours“ tab there is one block „Theme colours“ which contains two sections named „Important colors“ and „Other“. You can define the block and section individually for each item. Example:

```

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

```

```

"config": {
  "fields": {
    "sw-color-brand-primary": {
      "label": {
        "en-GB": "Primary colour",
        "de-DE": "Primär"
      },
      "type": "color",
      "value": "#399",
      "editable": true,
      "tab": "colors",
      "block": "themeColors",
      "section": "importantColors"
    }
  }
}

```

The tab and section property is not required.

You can extend the config to add translated labels for the tabs, blocks and sections:

```

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

  "config": {
    "blocks": {
      "colors": {
        "themeColors": {
          "en-GB": "Theme colours",
          "de-DE": "Theme Farben"
        }
      }
    },
    "sections": {
      "importantColors": {
        "label": {
          "en-GB": "Important colors",
          "de-DE": "Wichtige Farben"
        }
      }
    }
  }
}

```

```

    }
  },
  "tabs": {
    "colors": {
      "label": {
        "en-GB": "Colours",
        "de-DE": "Farben"
      }
    }
  },
  "fields": {
    "sw-color-brand-primary": {
      "label": {
        "en-GB": "Primary colour",
        "de-DE": "Primär"
      },
      "type": "color",
      "value": "#399",
      "editable": true,
      "tab": "colors",
      "block": "themeColors",
      "section": "importantColors"
    }
  }
}

```

[/expand]

Shopware 6 Theme entwickeln – ein neues Theme entwickeln

Shopware 6 Theme entwickeln – ein neues Theme entwickeln

[expand title="mehr lesen..."]

Creating a new theme

Version

6.0.0 or newer

Difference between „themes“ and „regular“ plugins

There are basically two ways to change the appearance of the storefront. You can have „regular“ plugins which main purpose is to add new functions and change the behavior of the shop. These Plugins might also contain scss/css and javascript to be able to embed their new features into the storefront.

A shop owner can install your plugin over the plugin manager and your scripts and styles will automatically be embedded. The theme which is currently selected by the shop owner will be recompiled with your custom styles.

The second way to change the appearance of the storefront is to create a theme. The main purpose of themes is to change the appearance of the storefront and they behave a bit different compared to „regular“ plugins.

Technically a theme is also a plugin but it will not only appear in the plugin manager of the administration, it will also be visible in the theme manger once activated in the plugin manager. To distinguish a theme plugin from a „regular“

plugin you need to implement the Interface Shopware\Storefront\Framework\ThemeInterface A theme can inherit from other themes, overwrite the default configuration (colors, fonts, media) and add new configuration options.

Creating a new theme

Open your terminal and run any of these commands to create a new theme.

```
# run this inside the project directory to create a new theme
$ bin/console theme:create MyTheme
```

```
# you should get an output like this:
```

```
Creating theme structure under
.../development/custom/plugins/MyTheme
```

Installing your theme

```
# run this to let shopware know about your plugin
$ bin/console plugin:refresh
```

```
# you should get an output like this
```

```
[OK] Plugin list refreshed
```

Shopware Plugin Service

=====

Plugin	Author	Label	Installed	Version
Upgrade version				Active
Upgradeable				

```
MyTheme      Theme MyTheme plugin      6.2
```

No

No

No

Now you can install your theme and activate it with the following command.

```
# run this to install and activate your plugin
$ bin/console plugin:install --activate MyTheme
```

you should get an output like this

Shopware Plugin Lifecycle Service
=====

```
Install 1 plugin(s):
* Theme MyTheme plugin (6.2)
```

Plugin "MyTheme" has been installed and activated successfully.

[OK] Installed 1 plugin(s).

Changing current theme

```
# run this to change the current storefront theme
$ bin/console theme:change
```

you will get an interactive prompt to change the
current theme of the storefront like this

```
Please select a sales channel:
[0] Storefront | 64bbbe810d824c339a6c191779b2c205
[1] Headless | 98432def39fc4624b33213a56b8c944d
> 0
```

```
Please select a theme:
[0] Storefront
[1] MyTheme
> 1
```

Set "MyTheme" as new theme for sales channel "Storefront"
Compiling theme 13e0a4a46af547479b1347617926995b for sales
channel MyTheme

Now your theme is fully installed and you can start your
customization.

Directory structure of a theme

In the file tree below you can see the basic file structure of
the generated theme:

```
# move into your theme folder
$ cd custom/plugins/MyTheme
```

```
# structure of theme
```

```
├── composer.json
├── src
│   ├── MyTheme.php
│   └── Resources
│       ├── app
│       │   └── storefront
│       │       ├── dist
│       │       │   ├── storefront
│       │       │   │   └── js
│       │       │   │       └── my-theme.js
│       │       └── src
│       │           ├── assets
│       │           ├── main.js
│       │           └── scss
│       │               ├── base.scss
│       │               └── overrides.scss
│       └── theme.json
```

Commands

The theme system can be controlled via CLI with the following
commands.

Theme refresh

Normally new themes are detected automatically but if you want to trigger this process run the command

```
# run
$ bin/console theme:refresh
```

Change a theme

After scanning for themes these can be activated using

```
# run this to interactively change a theme
$ bin/console theme:change
```

```
#run this to change a theme for all sales channels
$ bin/console theme:change MyCustomTheme --all
```

Notice: theme:change will also compile the theme for all new assignments.

Compile a theme

Calling the theme:compile command will recompile all themes which are assigned to a sales channel.

```
# run this to compile the scss and js files
$ bin/console theme:compile
```

Browser Compatibility Note

Administration: All browsers in the latest version (Edge, (no IE), Chrome, Firefox, Safari, Opera)

Storefront: All browsers in the latest version and additionally IE 11

[/expand]

Shopware 6 Theme entwickeln – Livestream #2

Shopware 6 Theme entwickeln – Livestream #2

[expand title="mehr lesen..."]

Your browser does not support HTML5 video.

[/expand]

Shopware 6 Theme entwickeln – Livestream #1

Shopware 6 Theme entwickeln – Livestream #1

[expand title="mehr lesen..."]

Console

```
/Applications/MAMP/bin/php/php7.3.1/bin/php bin/console
```

shopware plugin service

```
/Applications/MAMP/bin/php/php7.3.1/bin/php bin/console  
plugin:list
```

shopware plugin deinstallieren

```
/Applications/MAMP/bin/php/php7.3.1/bin/php bin/console  
plugin:uninstall ...
```

shopware cache leeren

```
/Applications/MAMP/bin/php/php7.3.1/bin/php bin/console  
cache:clear
```

theme skelett erstellen

```
/Applications/MAMP/bin/php/php7.3.1/bin/php bin/console  
theme:create name (zB. EightTheme)
```

```
/Applications/MAMP/bin/php/php7.3.1/bin/php bin/console  
theme:create --help
```

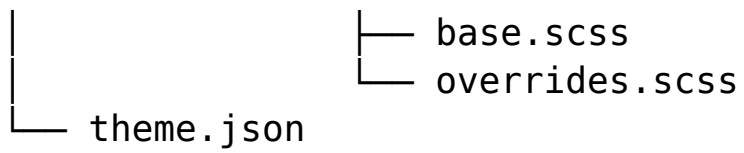
Your browser does not support HTML5 video.

move into your theme folder

```
$ cd custom/plugins/MyTheme
```

structure of theme

```
├── composer.json  
├── src  
│   ├── MyTheme.php  
│   └── Resources  
│       ├── app  
│       │   ├── storefront  
│       │   │   ├── dist  
│       │   │   │   ├── storefront  
│       │   │   │   │   ├── js  
│       │   │   │   │   └── my-theme.js  
│       │   └── src  
│       │       ├── assets  
│       │       ├── main.js  
│       │       └── scss
```



[/expand]