BELVG
FOR SUCCESSFUL
E-COMMERCE

AWARD WINNING
ECOMMERCE AGENCY

Magento 2 Certified Professional
Front End Developer Guide

www.belvg.com Phone: +1 650 353 23 01 E-mail: contact@belvg.com

Section 1: Create Themes

1.1 Describe folder structure for local and Composer-based themes

In which folders can themes be located?

If we create a theme manually, it should be located in the ***app/design*** folder.

If we install a theme through Composer, it is located in the ***vendor/*** catalog and can be stored anywhere in **root**.

What determines where a theme is installed?

As it was mentioned above, a place where a new theme is installed depends only on how the theme is created - manually or using Composer.

Magento uses the Composer autoloader. When the application is launched, Composer executes each file in the autoload.files section. registration.php and then it is registered as a theme.

To install a theme you need to log in the admin panel or reload it (if you have already logged in to it).

What is the difference if a theme is installed in one or the other of the possible directories?

If a theme is intended for the admin panel, then it should be in the folder ***admin/<VendorName>/<theme>*** (for a theme created manually the correct folder is ***app/design/admin/<VendorName>/<theme>***).

If the theme is intended for the front-end of the website, it should be in the ***frontend/<VendorName>/<theme>*** directory (for a theme created manually the correct folder is ***app/design/frontend/<VendorName>/<theme>***).

Composer-based themes are loaded from external sources and cannot be edited directly, whereas local themes are part of the project source code and therefore can be edited directly.

Composer-based themes are usually purchased themes which are installed to a website. And if we want to create a theme from scratch, or on the basis of a default theme, then we create the theme locally (in the app/design folder).

1.2. Describe the different folders of a theme

Which folders can exist within a theme?

A theme can contain the following folders: folders for module extension, etc folder, i18n, media, web. As a rule, web folder contains css, js, fonts, images, template folders. The image below demonstrates the theme folders structure.
A theme can contain the following folders: folders for module extension, etc folder, i18n, media, web. As a rule, web folder contains css, js, fonts, images, template folders. The image below demonstrates the theme folders structure.


etc
i18n
Magento_AdvancedCheckout
Magento_Banner
Magento_Braintree
Magento_Bundle
Magento_Catalog
Magento_CatalogEvent
Magento_CatalogSearch
Magento_Checkout
Magento_Cms
Magento_Customer
Magento_Downloadable
Magento_GiftCard
Magento_GiftCardAccount
Magento_GiftMessage
Magento_GiftRegistry
Magento_GiftWrapping
Magento_GroupedProduct
Magento_Invitation
Magento_LayeredNavigation
Magento_Msrp
Magento_MultipleWishlist
Magento_Multishipping
Magento_Newsletter
Magento_Paypal
Magento_ProductVideo
Magento_Reports
Magento_Review
Magento_Reward
Magento_Rma
Magento_Sales
Magento_SalesRule
Magento_SendFriend
Magento_Swatches
Magento_Theme
Magento_Vault
Magento_VersionsCms
Magento_Weee
Magento_Wishlist
media
web

http://prntscr.com/oc2igr

Which folders are optional and which are required?

In Magento 2 custom theme only one folder is required- "**/etc**", and only in case the parent theme does not contain this folder with "**/etc/view.xml**" file. Therefore, if the parent theme contains it, then it becomes unnecessary in the custom theme. Let us examine the table below:

Folder	Necessity	Folder example
/**<name_Vendor>_<name_Module>**	Not necassary	Magento_Theme, Magento_Catalog, Magento_Checkout and others
/<name_Vendor>_<name_Module>/**layout**	Not necassary	Magento_Catalog/**layout**
/<name_Vendor>_<name_Module>/layout/**base**	Not necassary	Magento_Catalog/layout/**base**
/<name_Vendor>_<name_Module>/**templates**	Not necassary	Magento_Catalog/**templates**
/<name_Vendor>_<name_Module>/**email**	Not necassary	Magento_Sales/**email**
/<name_Vendor>_<name_Module>/**email/i18n**	Not necassary	Magento_Customer/email/**i18n**/ar_SA
/etc	Necessary, if the parent folder does not have the /etc/view.xml file	/app/design/frontend/Magento/blank/**etc**
/i18n	Not necassary	/app/design/frontend/Magento/blank/**i18n**
/media	Not necassary	/app/design/frontend/BelVG/caskers_theme/**media**
/web	Not necassary	This folder will be described below

What is the purpose of each of the folders?

To learn the purpose of each of the folders, examine the table below:

Folder	Folder description
/**<name_Vendor>_<name_Module>**	The folder stores templates, layouts and web folder.
/<name_Vendor>_<name_Module>/**layout**	The folder stores the extending layout files. Files examples: **default.xml, default_head_blocks.xml, catalog_category_view.xml**
/<name_Vendor>_<name_Module>/layout/override/**base**	The folder stores layout files that override the default module templates.
/<name_Vendor>_<name_Module>/layout/override/**<parent_theme>**	The folder contains the layout files that override parent theme module layouts.
/<name_Vendor>_<name_Module>/**templates**	The folder stores template files of the module.
/<name_Vendor>_<name_Module>/**email**	The folder stores email templates files.
/<name_Vendor>_<name_Module>/**email/18n**	The folder stores localized files of email templates.
/<name_Vendor>_<name_Module>/**web**	The folder stores static module files. Below, we will describe in details the folder contents.
/**etc**	The file stores configurational file view.xml
/**i18n**	The folder stores .csv translations.
/**media**	The folder stores a draft theme image.
/**web**	The folder contains theme static files. Below, we will describe in detail the folder contents.

We will consider the web folder and its subfolders separately. Apart from the theme root, the folder can contain the extending module. Let us examine the table below:

Folder	Folder description

/<name_Vendor>_<name_Module>/**web**/**css**	The folder stores css and less files of the module. Folder example: /Magento_Checkout/**web**/**css**
/<name_Vendor>_<name_Module>/**web**/**js**	The folder stores JS files of the module. Folder example: /Magento_Checkout/**web**/**js**
/<name_Vendor>_<name_Module>/**web**/**template**	The folder stores templates for JS components and widgets. File format is .html. Folder example: /Magento_Checkout/**web**/**template**
/<name_Vendor>_<name_Module>/**web**/**i18n**	The folder stores localized folders for the module, such as css/, js/ and other. Folder example: Magento_Customer/web/**i18n/ar_SA/js/**
/<theme_dir>/**web**/**i18n**	The folder stores the localized folders for the theme, like css/, js/ and other. Folder example: /app/design/frontend/Magento/blank**/web/i18n/ar_SA/css/source/_extend.less**
/<theme_dir>/**web**/**css**	The folder stores css and less files of the theme Folder example: /app/design/frontend/Magento/blank/**web/css**
/<theme_dir>/**web**/**fonts**	The folder stores fonts files. Folder example: /app/design/frontend/Magento/luma/**web**/**fonts**
/<theme_dir>/**web**/**images**	The folder stores image files for different sprites, theme logo, etc. Folder example: /app/design/frontend/Magento/luma/**web**/**images**
/<theme_dir>/**web**/**mage**	The folder contains Magento libraries files, such as jquery, jqueryui and different cms widgets. It is not advised to use this folder to substitute the components. Instead, use requireJS.

1.3 Describe the different files of a theme

Which files are required to be present in a theme?

The following files are required to be present in each theme:

- /**registration.php** - the file contains the information required for theme registration.

- /**theme.xml** - the file is necessary for recognizing a theme in Magento. It contains the following information about the theme: its name (available in the list of available themes on the admin panel), parent theme name (if the theme is inherited from another theme) and image path (used as a theme preview on the admin panel).

- **/etc/view.xml** - the file contains configurations for all the product images and thumbnails, configurations of the product images gallery and Js Bundle settings. The parameter is required only in case when the theme does not have a parent theme (otherwise the file is not required).

Which files are optional?

Except for the required files we described above, other theme files are optional. Let us describe them as well:

- /**composer.json** - the file describes the main instructions and dependencies of the theme, as well as theme information. The file is required if your theme is the Composer package.
- /**requirejs-config.js** - the file contains RequireJS configuration.

Apart from these two files located at the theme root directory, there is a number of different file types (.xml, .phtml, .html, .js, .less, .css, .cs, etc.), located at the folders and subfolders of the theme. We will examine them in detail further on.

What is the purpose of each of the different file types?

Below, we will enumerate all the file types and define their purposes:

- **.xml** - layout files that define the overall page structure, for example, header, footer, sidebar, menu, etc position.
- **.phtml** - template files with the code elements PHP and HTML markup. They add the functionality and content for the elements.
- **.html** - markup files, applied for UI components.
- **.js** - JavaScript files. This can be both third-party libraries and custom scripts. Their purpose is to add logic and interaction elements to the website.
- **.less** and **.css** - style files, responsible for the website layout. As a rule, LESS files are applied to create new styles for the theme, while CSS files - for the connected side libraries. All style files get the .css extension after the compilation.
- **.csv** - applied for language packs; translation to different languages are stored in .csv files in i18n folder of a theme or a module.
- **Image files** - images used for theme styles, for theme preview, etc. They can be in different formats (.png, .jpg, .svg, etc.).
- **Fonts files** - custom fonts, used in the style files; as a rule, each connected font has several files with different extensions so that it would be supported in different browsers - .otf, .ttf, .woff, etc.).

1.4 Understand the usage of Magento areas: adminhtml/base/frontend

Which design areas exist?

Magento 2 has two main areas:

- **Magento Admin** (**adminhtml)** is used for admin panel, module display and the resources applied solely at the admin panel. The files are located at **/app/design/adminhtml,** and as for the module, it is located at the **<module_dir>/view/adminhtml** path.
- **Storefront (frontend)** is used for the external part of the online store; it displays modules and resources that are used for the visible to the customer area. The files are located at **/app/design/frontend,** and as for the module, it is located at the **<module_dir>/view/frontend** path.

Modules contain the **Basic** (**base)** area, applied for module's basic files. The files are located at the **<module_dir>/view/base** folder of the module.

What is the difference between them, and what do design areas have in common?

The files from **Basic** (**base)** can be used in both **adminhtml** and **frontend** areas**.** At the same time, it is impossibe to use **adminhtml** files at the **frontend** and vice versa**.** We can also use UI components in **adminhtml**, while from the **frontend** such ability is missing. What the areas have in common is the structure (to learn more, return to paragraphs 1.2 and 1.3).

What are design areas used for?

- **adminhtml** is used to create a new admin theme or customize the existing one.
- **frontend** is used to create a new external theme or customize the existing one.

How can design areas be utilized for custom themes or customizations?

Design areas can be applied to achieve the following:

- Create or customize admin panel theme design
- Add the abilities to edit modules parameters via the admin panel
- Create or customize the design for **Storefront** theme
- Modify the functionality of the modules' external areas (displayed at the **Storefront**).
- Separate letters' templates from the **Admin panel** and **Storefront.**

Section 2: Magento Design Configuration System

2.1 Describe the relationship between themes

What type of relationships can exist between themes?

In Magento 2, the relationships between themes are defined by inheritance fallback mechanism. The mechanism functions the following way: in case view files are not found in the current theme, then the system searches for the files in parent themes, modules or libraries. For example, let's imagine we created a theme with several external modifications, which will be inherited from the existing theme. Then, instead of copying all theme files, we need to copy only the files that will change. In other words, there is a parent theme (the one from which the child theme is inherited) and child theme (the one from which the parent theme is inherited).

What is the difference between a parent theme and a child theme?

Parent theme is the theme from which all the files for the child theme are inherited. **Parent theme** can have its own parent theme, which makes it a child theme of the original parent theme.

Child theme is the theme, the files of which are inherited from Parent theme.

Modification of **Parent theme** files can have an effect on **Child theme**, in case when they are not overridden in Child theme. In contrast, the modification of **Child theme** files has no influence on **Parent theme**.

How can the relationship between themes be defined and influenced?

A parent theme in Child theme.xml file is set:
/app/design/frontend/(vendor)/(theme_name)/theme.xml

```
(<parent>Magento/Blank</parent>).
```

```
<?xml version="1.0"?>
<theme xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:Config/etc/theme.xsd">
   <title>My Theme</title>
   <parent>Magento/blank</parent>
   <media>
       <preview_image>media/preview.jpg</preview_image>
   </media>
</theme>
```

How is that taken into account when creating a custom theme or customizing an existing theme?

There can be an unlimited amount of inheritance levels in Magento 2.

Let us consider several examples of extending or overriding the child theme files.

Layout files

The mechanism of layout files processing is not enabled in fallback system. The system gathers layout in the following order:

1. Current theme layouts: **<theme_dir>/<Vendor>_<Module>/layout/**
2. Ancestor themes layouts, starting from the most distant ancestor, recursively until a theme with no parent is reached: **<parent_theme_dir>/<Vendor>_<Module>/layout/**
3. Module layouts for the frontend area: **<module_dir>/view/frontend/layout/**
4. Module layouts for the base area: **<module_dir>/view/base/layout/**

Layout files can be easily enhanced, unlike the template files. All you need is to create a new xml. file in child theme in the layout folder of the needed module. For example, if you need to alter the position of the heading block at the product catalog page, you need to create a **catalog_product_view.xml** file at the **\app\design\frontend\vendor_name\theme_name\Magento_Catalog\layout\catalog_category_view.xml** path and add the following content into it:

```
<?xml version="1.0"?>
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
layout="2columns-left"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page_c
onfiguration.xsd">
    <body>
        <move element="page.main.title" as="page.main.title"
destination="content" before="-"/>
    </body>
</page>
```

This will extend the layout without overriding other rules.

But if you need to override an xml.files, then create a file with the same name as the overridden file at the path:
theme_dir/vendor_module/layout/override/theme/vendor_name/<ancestor_theme>

Templates files

The fallback scheme for templates is the following:

1. Current theme templates:
   <theme_dir>/<Namespace>_<Module>/templates
2. Ancestors themes templates, recursively, until a theme with no ancestor is reached:
   <parent_theme_dir>/<Namespace>_<Module>/templates
3. Module templates: **<module_dir>/view/frontend/templates**

Unlike the layouts, template files can not be extended, only overridden. In order to override the product catalog file, you need to copy:
Magento_Catalog_module_dir\view\frontend\templates\product\list.phtml
At the path:
app\design\frontend\vendor_name\theme_name\Magento_Catalog\templates\product\list.phtml
and enter the needed modifications.

The same system applies to static files (js, css, images, fonts and .less).

The certain catalogs that the system searches for during the backup depend on whether the module context is known for the file.

If module context is not defined for the file:

1. Current theme static files for a specific locale (the locale set for the storefront): **theme_dir/web/i18n/<locale>**

2. Current theme static files: **theme_dir/web/**
3. Ancestor's static files, recursively, until a theme with no parent is reached:
    - **parent_theme_dir/web/i18n/<locale>**
    - **parent_theme_dir/web/**
4. Library static view files: **lib/web/**

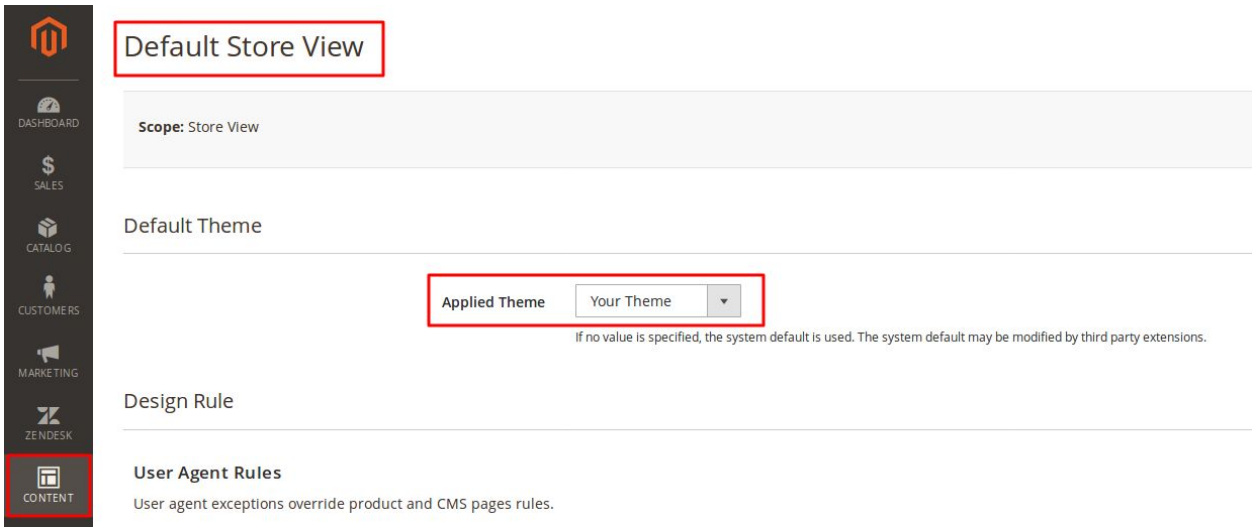
If module context is defined:

1. Current theme and current locale module static files: **theme_dir/Namespace_Module/web/i18n/<locale>/**
2. Current theme module static files **theme_dir/Namespace_Module/web/**. Example: **app/design/frontend/BelVg/BelvgTheme/Magento_Catalog/web/**
3. Ancestor themes module static files, recursively, until a theme with no ancestor is reached:
    - **parent_theme_dir/Namespace_Module/web/i18n/<locale>/**
    - **parent_theme_dir/Namespace_Module/web/**
4. Module static view files for the current locale and frontend area: **module_dir/view/frontend/web/i18n/<locale>/**
5. Module static view files for the current locale and base area: **module_dir/view/base/web/i18n/<locale>/**
6. Module static view files for the frontend area: **module_dir/view/frontend/web/**
7. Module static view files for the base area: **module_dir/view/base/web/**

2.2 Configure the design system using the options found in the Admin UI under Content > Design > Configuration

How do the configuration settings affect theme rendering?

First, let's log in the Magento admin panel and navigate to the theme settings at Content — Design — Configuration.
Next, choose the necessary store. Here is what you will see on the page:

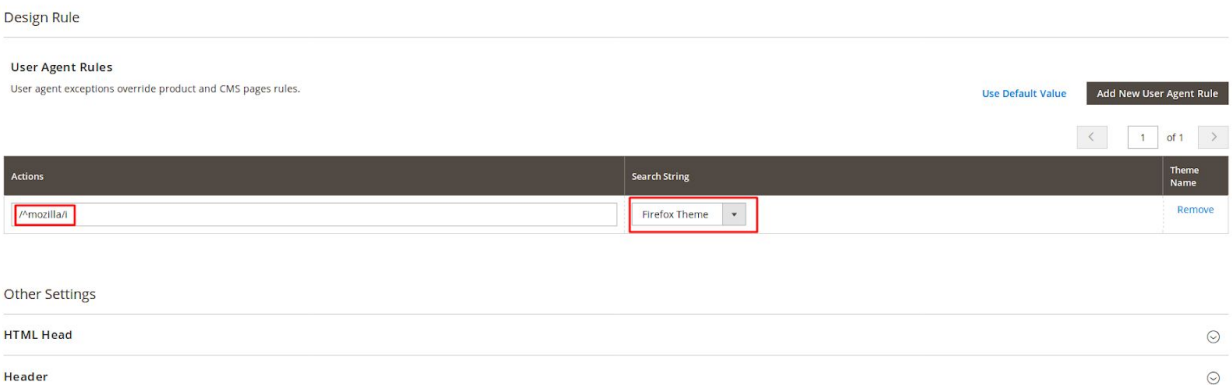


The current theme is applied by default, you can change it for any of the installed ones.
Click "save", clean the cache, and a new theme is displayed for the selected store.

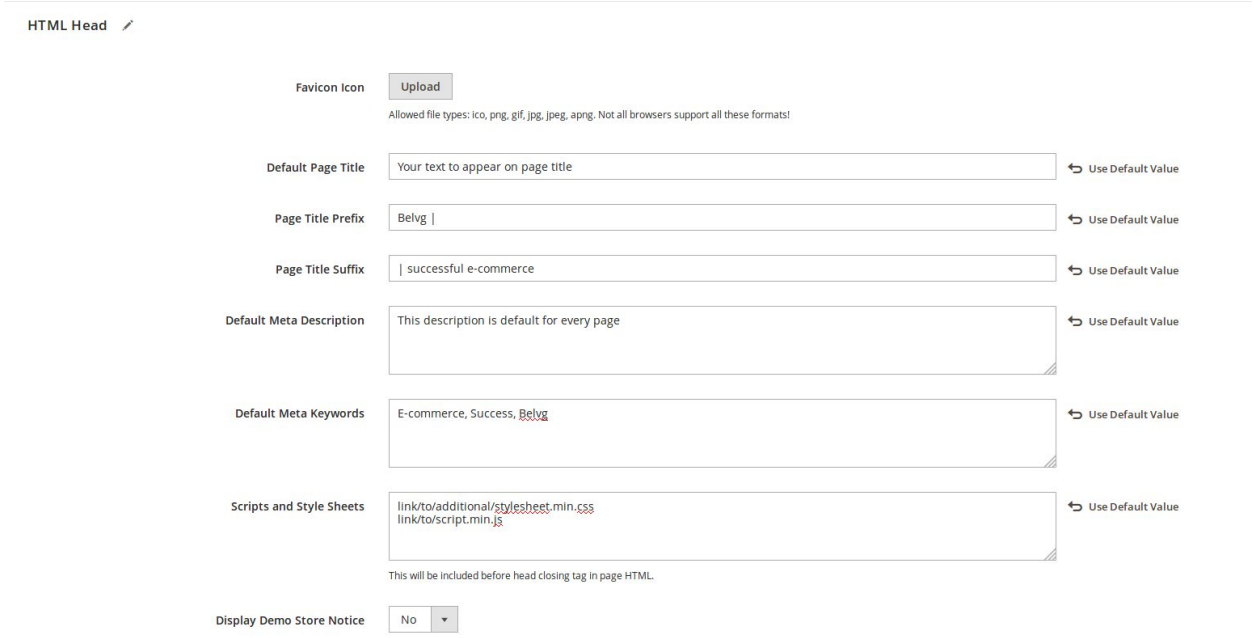
The next field of the menu are the user agent rules. Here you can set a theme for the necessary user agent. For example, let's apply a special theme with an animated favicon and CSS-grid for the Firefox browser:

- create a rule for the user agent;
- we write /^mozilla/i for its definition;
- choose a special theme;
- save and clean the cache.

Now the users with a Firefox browser will have their own theme.



Next are the settings for head, header, footer, search robots, pagination, watermarks for images and letters. Let's examine them in more detail.



Let's take a quick look at each of the fields:

- Favicon is an icon for quick identification of a site in the browser tab and in bookmarks. The supported formats are: ICO, PNG, APNG, GIF, and JPG (JPEG). Not all browsers support all types of favicons, so read the specifications carefully!
- The page title, displayed in a browser tab, is used for a variety of purposes, including SEO. If no header is set for each particular page, a default title is applied to all pages of a webstore (which is not good).
- Header prefix. It is used for 2 and 3-fragment headings. It is separated from the main header with a vertical dash or hyphen. It’s also applied on all pages, unless otherwise specified for a particular page.
- Header suffix. It’s similar to the prefix, but goes after the main header.
- Meta description. It’s used for SEO of the site, carries a brief information about the visited page and can not be longer than 160

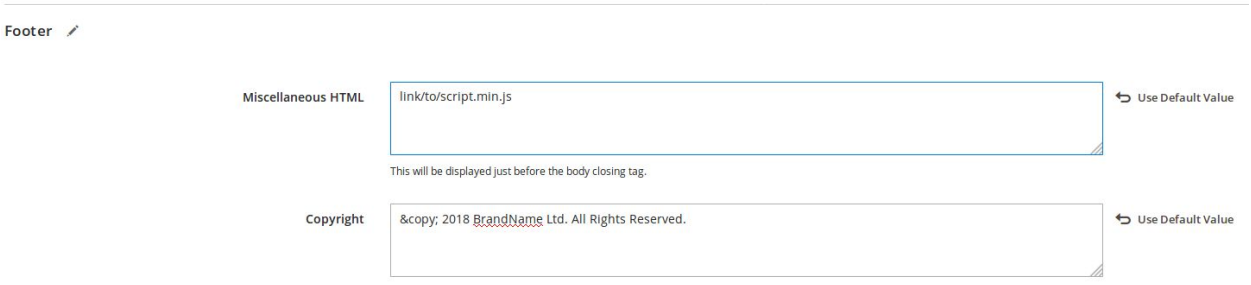
characters. Similar to headings, it's applied to all pages, unless otherwise specified for a particular page.

- Meta tags. A list of comma-separated tags for SEO of the page. Since Google abandoned it a long time ago, meta tags play a very small part in search optimization of your page.
- Scripts and style sheets. Additional styles and scripts that will be placed before the closing **</head>** tag.
- Display demo store notice. Enables/disables the demo store message.

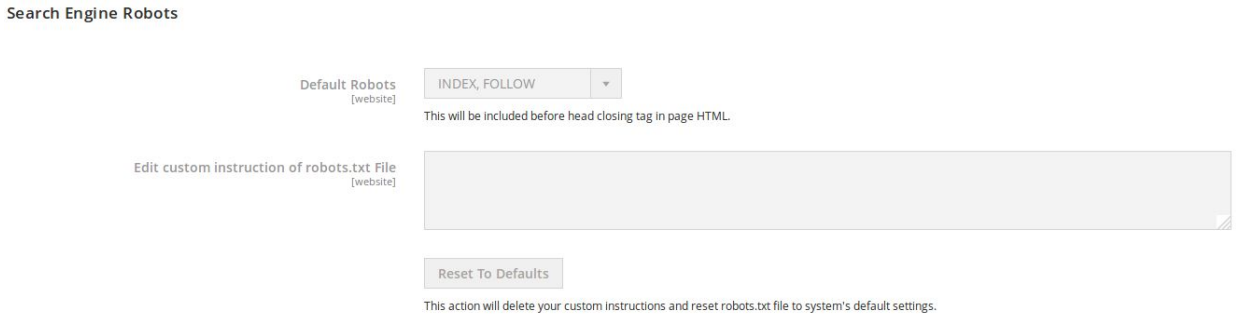
Next come the header settings:

- Logo image. The supported file types are PNG, GIF, JPG (JPEG).
- Logo attribute width in pixels.
- Logo attribute height in pixels.
- Welcome text, which also contains the name of a logged-in user.
- Logo image alt. Alt text for the logo.

Footer settings:

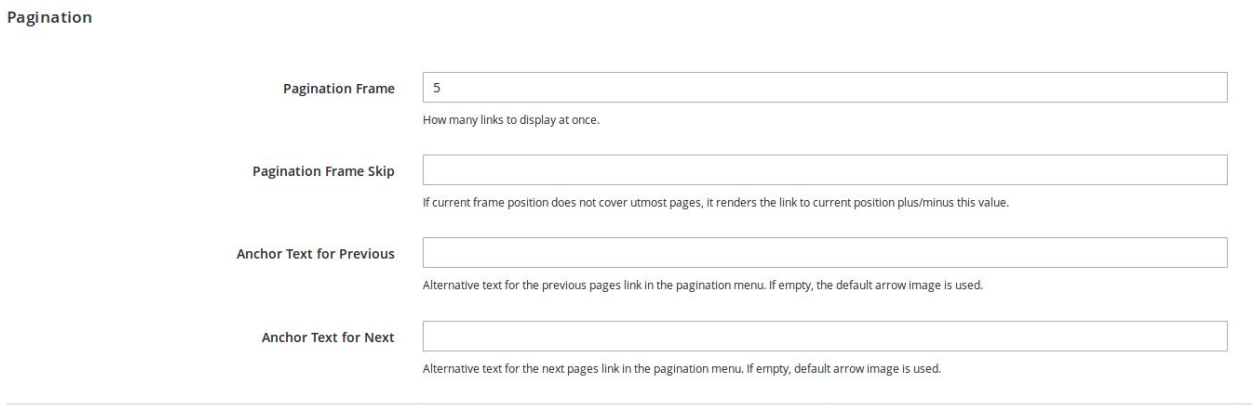


- A field for scripts that run before the closing **</body>** tag.
- Copyright. Supports &copy for the "©" symbol.
  Search engine robots settings:



Here it is written how robots should behave in terms of indexing and returning to the site on schedule and upon editing, as well as special rules, such as banning certain directories that should not be indexed.

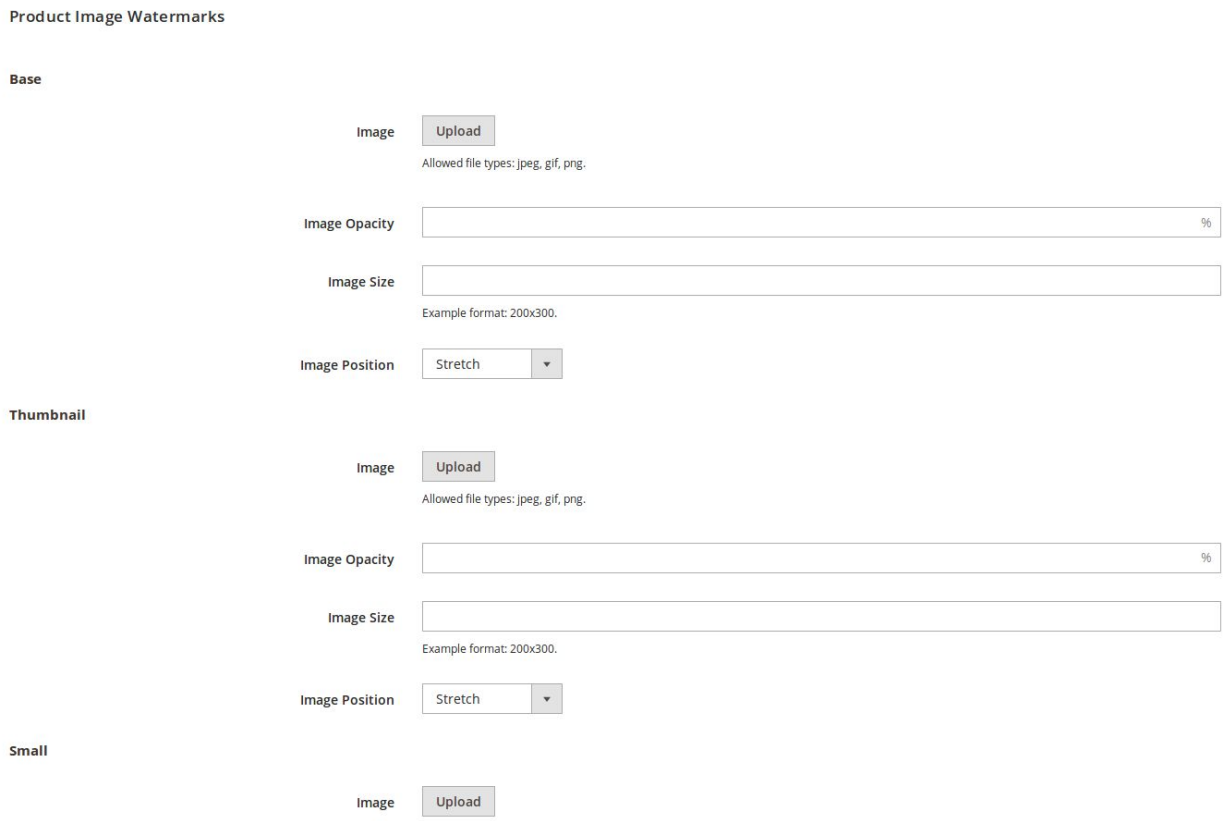
Pagination settings



Here you can specify:

- The number of links to next pages
- The number of links that will be skipped. If you need to skip the list of 5 links, and the step is set to 4, then the last link of the skip will be the first link on the new list.
- The text for the forward and back buttons, arrows are shown by default.

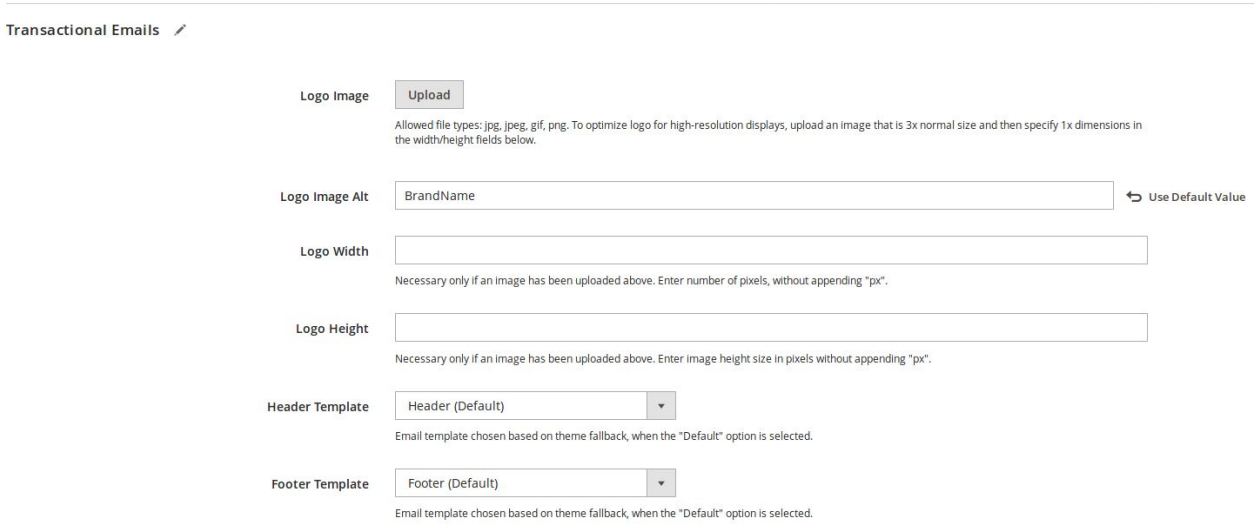
Product image watermarks



Here you can set the watermark parameters for the main image sizes:

- Image (in jpeg, gif, png);
- Image opacity in %;
- Image size in pixels;
- Image position. The position of the watermark on the image.

Transactional emails



Here you can set:

- Logo image (in jpg, jpeg, gif, png);
- Logo image alt;
- Logo width and logo height in pixels;
- Header and footer templates.

What happens if a theme is added or removed?

We can install a theme manually, either as a component or via composer. If, after the theme installation and admin panel reload, it appeared in the list of the available themes, then the process went successful and we can use it.

The process of theme removal differs depending on the way it was installed:

- Before the theme removal, set the website into dev mode and make sure that the theme is active as a parent or a child one.
- If the theme was installed manually, then all you need is to delete theme files and its data from the database.

- If the theme was installed via composer, we remove it via a CLI command; for example “magento theme:uninstall --backup-code frontend/BrandName/default”.
- If the theme was installed via the repository, then the course of action is the same as described in the previous point, only we delete the dependencies from the composer file.
- If the theme was installed as a component, remove the theme from the admin via uninstall in the installed components list.

What common mistakes can be made in regard to these settings?

Let us consider the most common mistakes which can be made in regard to these settings:

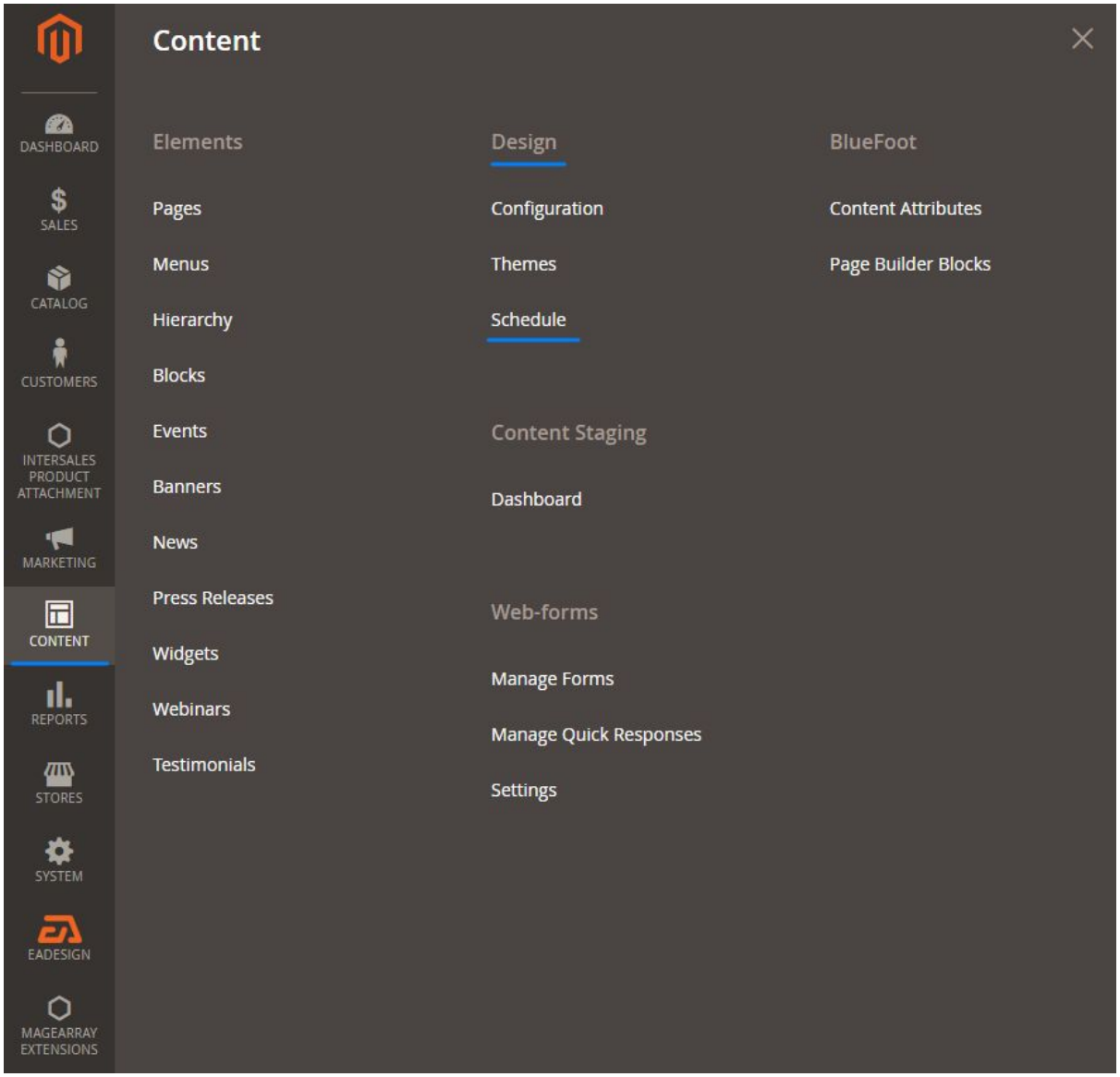
- The changes may not display if you did not clear cache or deploy static content
- The unsupported logo file type was loaded
- The theme was deleted incorrectly (the theme, installed via composer, was removed manually, etc.)
- The same meta description is used for all the pages
- If the rights for reading/recording was set wrong, then graphic files may not load.

2.3 Apply a temporary theme configuration to a store view using the options found in the Admin UI under Content > Design > Schedule

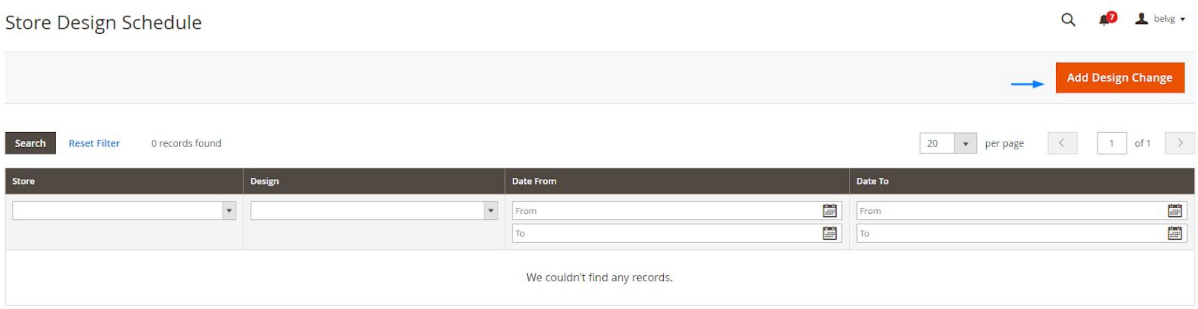
What is the purpose of this feature?

Store Design Schedule feature in Magento 2 was created to manage store view themes change according to the schedule. This feature is very useful if you need to change the store theme depending on the season, the upcoming holiday, discounts, etc.

Store Design Schedule can be found at the admin panel at the following path:
Content > Design > Schedule

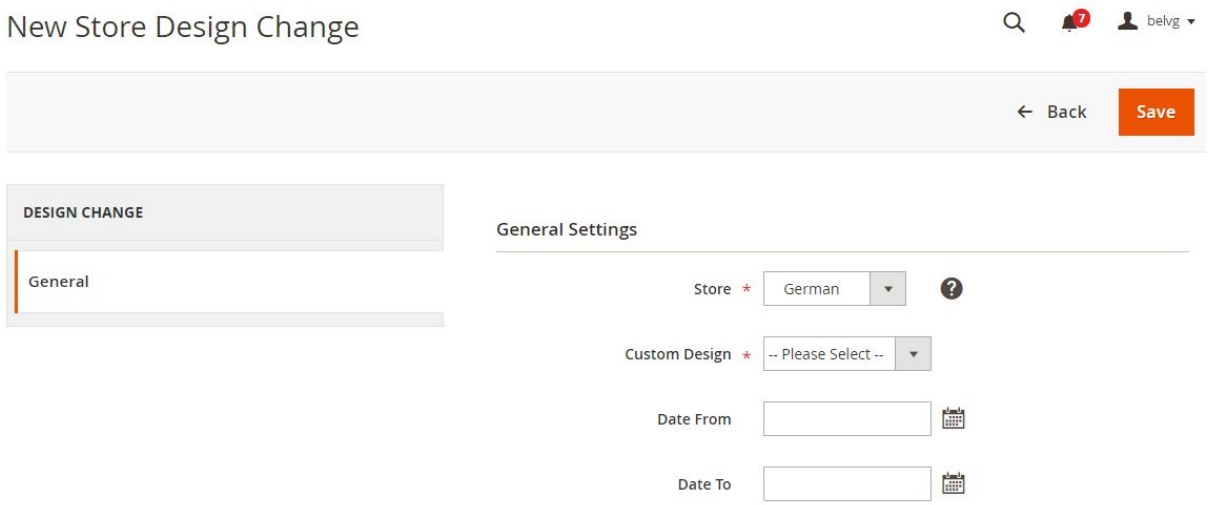


The page contains a list of all the scheduled design modifications. There is also a button for adding a new design on schedule.

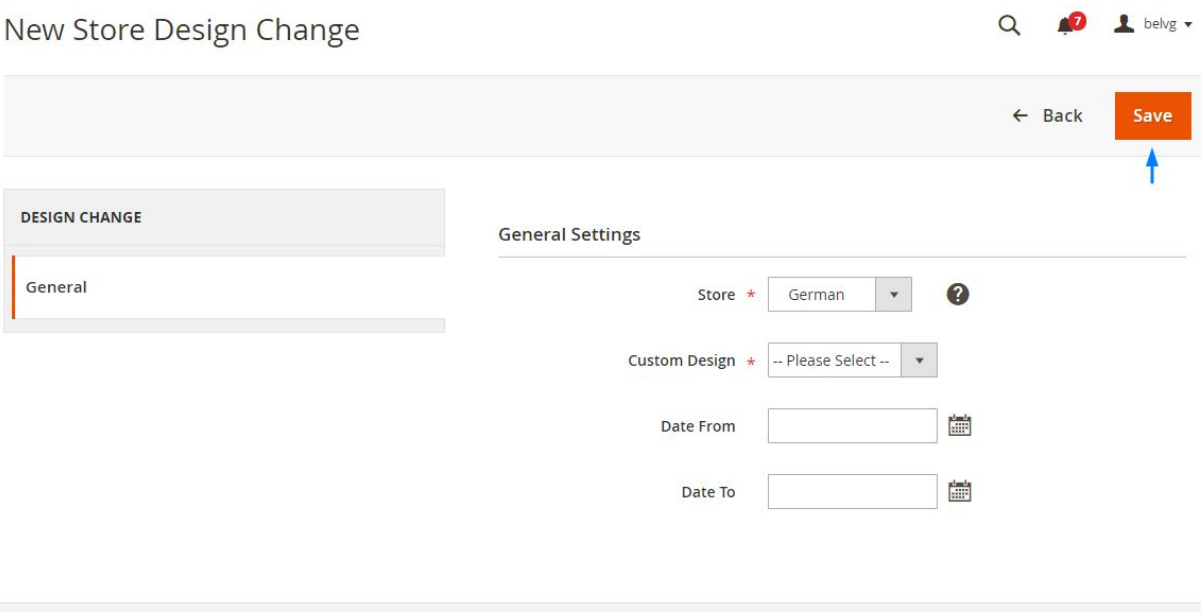


To create a new store design change, you need to specify the following settings:

- Store - on which of the current Stores the design will change.
- Custom design - design (theme) selection.
- Date from/Date to - the time range during which the selected design is applied and removed.



After all the options are selected, press the button in the upper right corner to save the changes.



If the time range of the created design modification overlaps the time range of the set design modification, it will be impossible to save changes until the overlap is resolved.

How does it influence rendering if a design change is scheduled?

When comes the date, set in **Date From** field, the selected theme is applied to the selected store instead of the current theme.

What happens at the end of a scheduled design?

When comes the date, specified in **Date To** field, then applies the theme that was the current one before the planned changes (selected in Content > Design > Configuration).

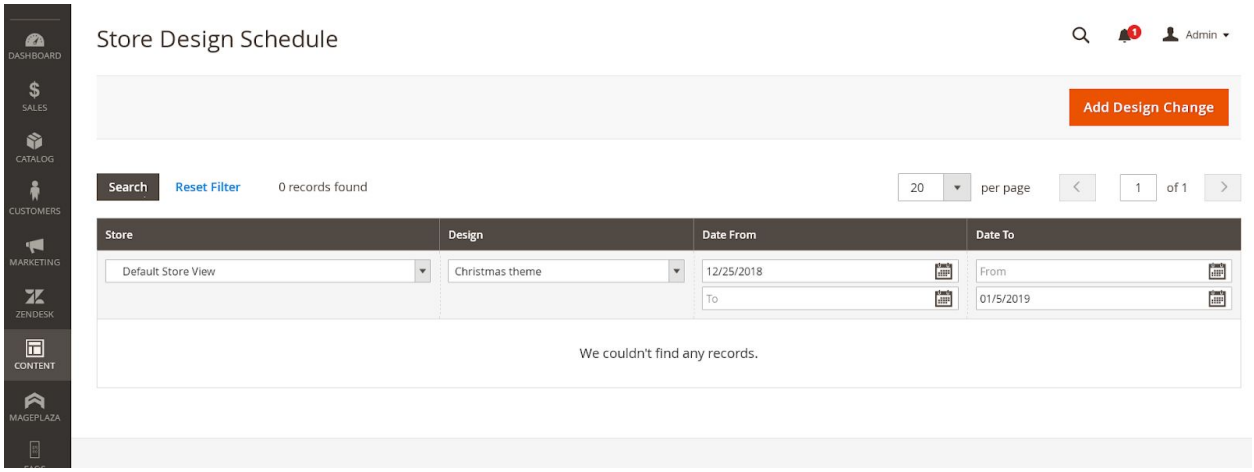
How is full page caching involved?

When full page caching is enabled, then the planned theme modifications will not be enabled until the cache is cleared.

2.4 Understand the differences and similarities between Content > Design > Configuration and > Schedule to configure the design fallback

What is the effect if both options are used at the same time?

When the design and schedule settings are set simultaneously, then the schedule will override the design settings for the date range, specified in it. With the help of this feature, you can prepare a special website design, postpone the design application for a certain term, and it will automatically apply on the date you specified.



The screenshot above demonstrates a special Christmas theme, set for the holiday's time range. When the festive season is over, the theme will automatically disable, and the content design settings will take force again.

Section 3: Layout XML in Themes

3.1 Demonstrate knowledge of all layout XML directives and their arguments

What layout XML elements exist and what is their purpose?

To connect different components and manage their interactions, Magento 2 uses XML templates and XML configurations for the page.

XML template serves to position the containers on the page, as well as set the view and structure options.
XML configuration distributes each block separately among the containers and assigns its configurations to each.

Below is the example of XML file:

```
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
layout="2columns-left"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_configuration.xsd">
   <body>
       <referenceContainer name="columns.top">
           <container name="category.view.container" htmlTag="div"
htmlClass="category-view" after="-">
               <block class="Magento\Catalog\Block\Category\View"
name="category.image"
template="Magento_Catalog::category/image.phtml"/>
               <block class="Magento\Catalog\Block\Category\View"
```

```
name="category.description"
template="Magento_Catalog::category/description.phtml"/>
                <block class="Magento\Catalog\Block\Category\View"
name="category.cms" template="Magento_Catalog::category/cms.phtml"/>
            </container>
        </referenceContainer>
        <referenceContainer name="content">
            <block class="Magento\Catalog\Block\Category\View"
name="category.products"
template="Magento_Catalog::category/products.phtml">
                <block
class="Magento\Catalog\Block\Product\ListProduct"
name="category.products.list" as="product_list"
template="Magento_Catalog::product/list.phtml">
                    <container name="category.product.list.additional"
as="additional" />
                    <block
class="Magento\Framework\View\Element\RendererList"
name="category.product.type.details.renderers"
as="details.renderers">
                        <block
class="Magento\Framework\View\Element\Template"
name="category.product.type.details.renderers.default" as="default"/>
                    </block>
                    <block
class="Magento\Catalog\Block\Product\ProductList\Item\Container"
name="category.product.addto" as="addto">
                        <block
class="Magento\Catalog\Block\Product\ProductList\Item\AddTo\Compare"
                                name="category.product.addto.compare"
as="compare"

template="Magento_Catalog::product/list/addto/compare.phtml"/>
                    </block>
                    <block
class="Magento\Catalog\Block\Product\ProductList\Toolbar"
name="product_list_toolbar"
```

```
template="Magento_Catalog::product/list/toolbar.phtml">
                        <block class="Magento\Theme\Block\Html\Pager"
name="product_list_toolbar_pager"/>
                    </block>
                    <action method="setToolbarBlockName">
                        <argument name="name"
xsi:type="string">product_list_toolbar</argument>
                    </action>
                </block>
            </block>
            <block class="Magento\Cookie\Block\RequireCookie"
name="require-cookie"
template="Magento_Cookie::require_cookie.phtml">
                <arguments>
                    <argument name="triggers" xsi:type="array">
                        <item name="compareProductLink"
xsi:type="string">.action.tocompare</item>
                    </argument>
                </arguments>
            </block>
        </referenceContainer>
        <referenceBlock name="page.main.title">
            <arguments>
                <argument name="id"
xsi:type="string">page-title-heading</argument>
                <argument name="add_base_attribute_aria"
xsi:type="string">page-title-heading toolbar-amount</argument>
            </arguments>
            <block class="Magento\Catalog\Block\Category\Rss\Link"
name="rss.link" template="Magento_Catalog::category/rss.phtml"/>
        </referenceBlock>
    </body>
</page>
```

These files have the .xml extension and are located at the layout folder of the module (*module_name/view/frontend/layout)*.

The main layout elements are **blocks** and **containers**.

Template containers can contain blocks, and their aim is to effectively position them inside the page. The following DOM elements are the most common containers:

- head
- header
- main
- aside (left or right)

These elements divide the page template into the containers, in which we will place our blocks. By default, Magento 2 has five page template types:

- empty (page without containers)
- 1 column (one container for content)
- 2 column with left bar (container for content and a left sidebar)
- 2 column with right bar (container for content and a right sidebar)
- 3 column (3 containers optionally)

The highlighted elements on the picture above are the containers in which we can insert any blocks we wish. The placement of blocks within itself is the main distinguishing feature of the <container> element, however, it is worth noting that, if necessary, the <block> element can also contain other elements and even containers, and thereby it automatically becomes a container for its child element.

Based on everything said above, one can get an impression that <container> and <block> elements are identical to each other. However,

there is sufficient difference between them in realization and operating methods (this will be explained further in the text).

```
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
layout="2columns-left"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_configuration.xsd">
    <body>
        <referenceBlock name="page.main.title">
            <arguments>
                <argument name="id"
xsi:type="string">page-title-heading</argument>
                <argument name="add_base_attribute_aria"
xsi:type="string">page-title-heading toolbar-amount</argument>
            </arguments>
            <block class="Magento\Catalog\Block\Category\Rss\Link"
name="rss.link" template="Magento_Catalog::category/rss.phtml"/>
        </referenceBlock>
    </body>
</page>
```

In the piece of code above we used xml tags <page> and <body> to configure containers for our page, and their syntax is very similar to <html> syntax. The same as in HTML markup, all tags must be closed (in the example above, we used </page> and </body> closing tags) or they can be self-closing, like <block name="blockName"/> tag.

All elements can have special attributes for their parameters management. For blocks and containers, the list of attributes and their purpose is similar:

Block and container common attributes

Attribute	Value	Description
name	**0-9, A-Z, a-z and (-)(_)(.)** *Should begin with a letter	A unique name for addressing is recorded.

	name=”current_name”	
before	**(-)** displays the element **before** all in the current block or container **(element name)** element display **before** the element with the specified name **()**when the parameter is absent, the element is considered not positioned and is displayed in the order set by the current template before=”-”	Applied to position elements inside the page template or inside the container itself.
after	**(-)** element display **after** all in the current block or container **(element name)** element display **after** the element with the specified name. **()**when the parameter is absent, the element is considered not positioned and is displayed in the order set by the current template after=”target_elemement”	Applied to position elements inside the page template or inside the container itself. Has priority over **before**
as	**0-9, A-Z, a-z and (-)(_)(.)** *Should begin with a letter\ name=”current_allias”	The set name serves to identify the current element in the parent
cacheable	**true, false** cacheable=”false”	Enables and disables the caching of the pages which contain elements with the current attribute; the attribute is necessary for creation of the dynamic elements, widgets and pages.

Attribute values and their functional load is identical for blocks and containers as well.
However, containers have the methods that are inherent for them solely.
Below is the table of the main attributes together with their unique values:

Container attributes

Attribute	Value	Description
output	**0 / 1** or **true / false** output=”1”	Is set to determine whether it is necessary to render the parent container, which contains a **false** element by default
HtmlTag	Html 5 tags (aside, main, div ...) HtmlTag=”div”	Everything that is inside the container will be displayed to the user inside the specified tag.
htmlId	**0-9, A-Z, a-z and (-)(_)(.)** *Functions only when the **HtmlTag** value is set htmlId=”current_id_name”	html **Id** selector is set for the specified wrap.
htmlClass	**0-9, A-Z, a-z and (-)(_)(.)** *Functions only when the **HtmlTag** value is set htmlClass=”current_class_name”	html **class** selector is set for the specified wrap.
label	Any value	A voluntary container name is set for display in the browser
layout	The name value of the page layout template layout=”grid_name”	It is written to indicate an explicit indication of the applicable template grid for the current layout.

*the values are not required

Let us also consider the parameters for specifying in the attributes, inherent to **blocks** only:

Block attributes

Attribute	Value	Description
class	The path to the class and the class name:	The path to the **class** that is responsible for processing information for the current

	class="Vendor\Folder_Name\Block\Class_Folder\Class_Name"	block of the template is indicated. It is applied to indicate its handler for a new element or to overwrite the current handler for a program block with a user one.
template	The path to the template and the template name template="Vendor_Name::folder_template/name_template.phtml"	Specified the path to the **template,** which is responsible for rendering the information for the current layout block. It is applied to specify a template for the new element or override the default template for the user one.

*For **blocks**, the "**class**" attribute is required for Magento 2.1 version and earlier.*
*For **containers,** the "**name**" attribute is required for Magento 2.1 version and earlier.*

What is the purpose of the attributes that are available on blocks and other elements?

An additional tool for working with layout XML files and page configurations in Magento 2 are the so-called instructions, which can be attributes of elements or new elements. These include the <block> and <container> tags, as well as a number of additional tags. Below is a list of the main ones:

referenceBlock and referenceContainer

The instruction that applies to a block or container correspondingly in order to pass the necessary parameters using the following attributes:

Attribute	Value	Description
name	The name of the	Specified the name of the block or container to which the

	destination block or container	instructions will be passed.
remove	true/false	Removes or cancels removal of the current block or container. When a container is removed, all child elements are deleted as well.
display	true/false	Turns on or off the rendering off on the page of the current container or block. When an element is disabled, the further possibility of configuring it and its child elements remains.

Nested attributes.

Magento 2 has several attributes that can be nested; their syntax is similar to the containers and blocks declaration and looks the following way:

```
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_layout.xsd">
   <move element="category.cms" destination="page.wrapper"
after="category.view.container"/>
</layout>
```

Example of move instructions to override the position of the ‘category.view.container’ container

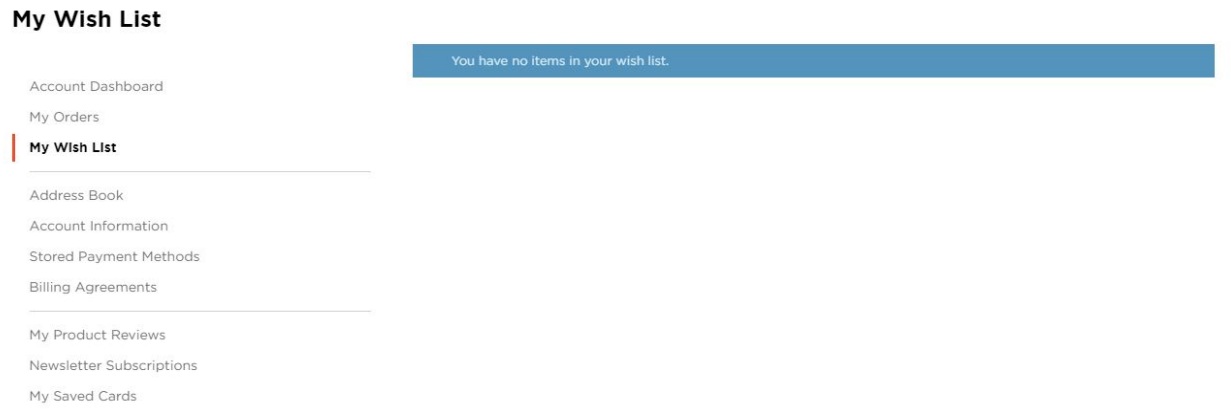
Move

Move is a perfect assistant for positioning elements at the page and migrating it from a container to a container.
Let us consider the example of title block migration that displays the current tab in the user account above the main container:

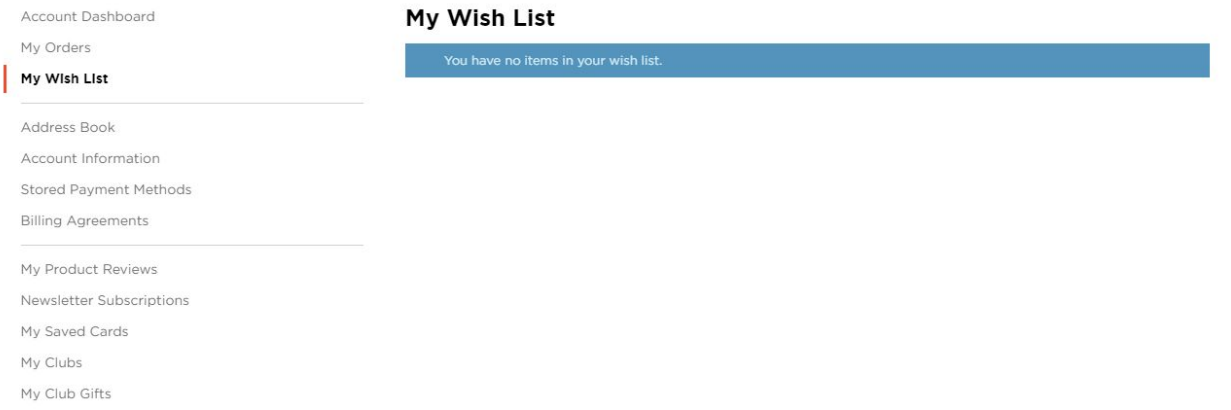
```
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
layout="2columns-left"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
```

```
page_configuration.xsd" >
    <body>
        <move element="page.main.title" destination="main"
before="-"/>
    </body>
</page>
```

Migration of title block from account-nav into main container



before



after

Using a single command in the layout, we can easily modify the whole pages structure without using styles and templates. The instruction includes the following attributes:

- element - the name of the target element for passing the instruction
- destination - the name of the parent element that will perform the moving

- as - an alias name for the element; set after the element is moved
- after/before - used for positioning inside the target parent element

Remove

It allows to remove static resource elements, like blocks with script files connection and styles in <head> container.

Below is the example of such modification:

```
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_layout.xsd">
    <head>
        <remove src="css/style-m.css"/>
    </head>
</layout>
```

Removal of the previously connected mapping CSS file

Update

The attribute is recommended to apply in cases when you need to duplicate one or several containers and blocks from the parent page template.

The instruction is specified in the beginning of XML template and applied with handle attribute, in which the path and parent template filename is set. This method has similar functionality to the super() method in the prototype-oriented programming and performs the same functions. The target template file will be updated recursively, or in a sequential order.

For example, you need to connect a slider with certain information to a number of pages in your project. In this case, we need to create a template

file with XML markup, which is necessary for connecting a slier. It may look the following way:

```
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_configuration.xsd">
   <body>
       <referenceContainer name="page.wrapper">
           <!--rev slider-->
           <container name="page.index.slider.container"
htmlTag="div" htmlClass="page-slider-container"
before="main.content">
               <block class="Nwdthemes\Revslider\Block\Revslider">
                   <arguments>
                       <argument name="alias"
xsi:type="string">Midleton</argument>
                   </arguments>
               </block>
           </container>
       </referenceContainer>
   </body>
</page>
```

Example of page template with slider connection

Further, we can apply this template as a prototype for the child files. To do this, set the path to the parent template file in the **update** directive of each target document.

```
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_configuration.xsd">
   <body>
       <update handle="{name_of_handle_to_include}"/>
```

```
        <!-- CONTENT -->
    </body>
</page>
```

This way, our slider will display at each page where the update directive with slider connection file as a prototype is specified.

Argument

This tag is used to pass arguments to a block or container. It contains a required name and type tags. Type contains the passed element's type:

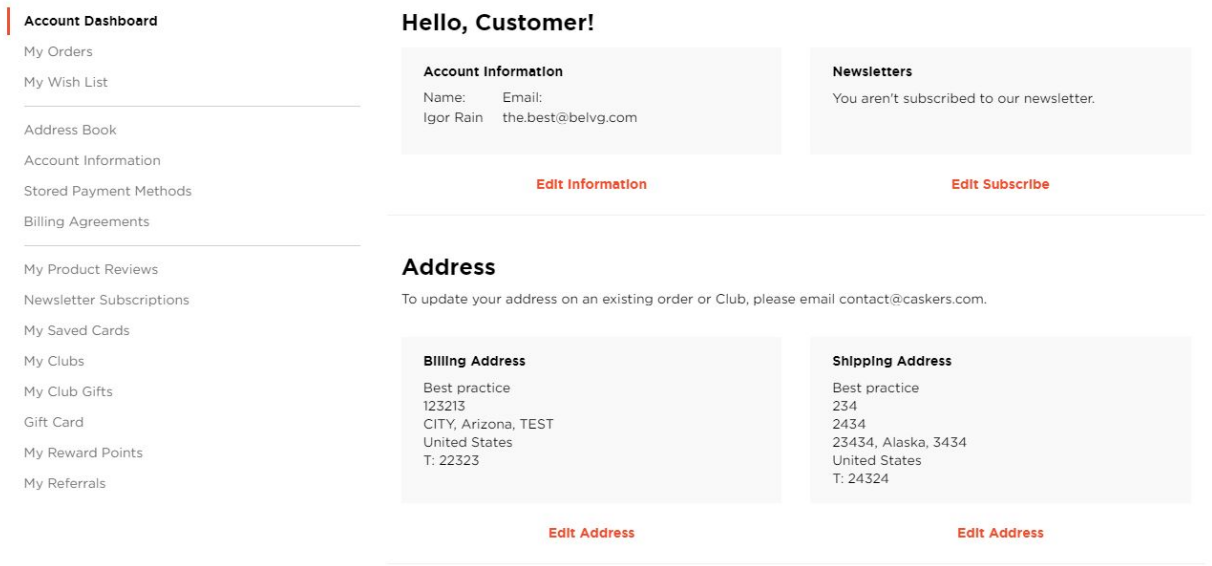
- string
- boolean
- object
- number
- null
- array

For example, earlier we moved the title from the header container into main in the user account, and now we need to set a custom string for the main tab myDashboard. For this, we open the template, responsible for myDashboard page and using **action** directive and **setPageTitle** method, we pass as an argument the string with the title we want our page to have:

```
<referenceBlock name="page.main.title">
    <action method="setPageTitle">
        <argument translate="true" name="title"
xsi:type="string">Hello, Customer!</argument>
    </action>
</referenceBlock>
```

Passing the parameter as a string for page-title

We set string for the type and the necessary text we passed as a string. Due to all this, we got the desired result.



Also, the passed values can be obtained from the current template file using **get()** method. This functionality perfectly corresponds to the modularity principle in Magento. Using it, you can create a universal but dynamic template and use it in several places in our store, just like with the example of a universal layout with connecting a slider on pages, using the update method inheritance.

As an example, consider the creation of a template with a button rendering and any functionality on the client. The dynamism will lie in the fact that on different pages we will use the same template, but depending on the requirements, in the page template file, we can pass a css class to it, to which some style from the stylesheet can be attached or even a functional load implemented on js. In the markup, it will look the following way:

```
<block class="Magento\Framework\View\Element\Template" name="button"
template="Magento_Theme::buttons/dynamic_button.phtml" >
   <arguments>
```

```
        <argument name="css_class" xsi:type="string">white</argument>
    </arguments>
</block>
```

Passing of a custom argument into the template

To pass the class, we declare our block and set our template in the template attribute. Then we simply pass any parameters to our template as arguments, and <arguments> container and <argument> blocks enclosed in it are used to pass several parameters . We passed our class with an argument and the name css_class, in order to get it in the template file, we need to resort to php knowledge and write the following construction:

```
$_className = $block->getCssClass();

<div class="actions-toolbar <?= $_className ?>">
    <button class="action primary">Button</button>
</div>
```

Example of the method for getting the argument, passed from the template

Then, depending on the logic we set, the button can take different modifications from page to page; in this example, it is white.

In fact, this is the simplest example of using this functionality, which is necessary to familiarize yourself with the principle of its operation. If we limit ourselves only to our imagination, then applying the method of passing arguments, you can do many things, up to passing an argument that will determine the controller for processing our template.
Arguments can be different and have different names, and you can pass not only a string, but, as previously stated, 6 basic data types. In order to get the attributes in the template, the function takes the following form:

```
getCssClass();
```

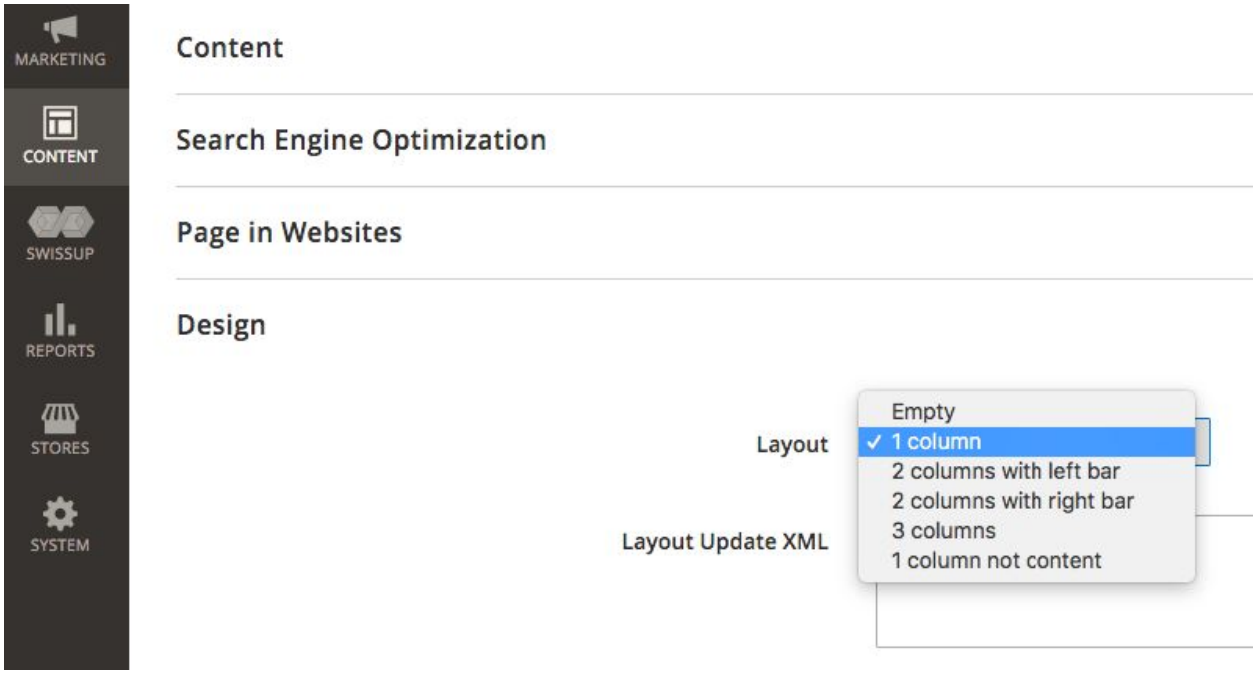
The **name** parameter passed in xml in the form of “css_class” takes the form of CamelCase and looks like “CssClass”. This method is similar to the method of passing arguments using data attributes, applied in JavaScript.

3.2 Describe page layouts and their inheritance

How can the page layout be specified?

To select a page layout for a certain page, log in to the admin panel and navigate at the following path for the settings:

```
Content ⇒  Pages ⇒  Target_Page_Edit ⇒ Design ⇒ Layout
```



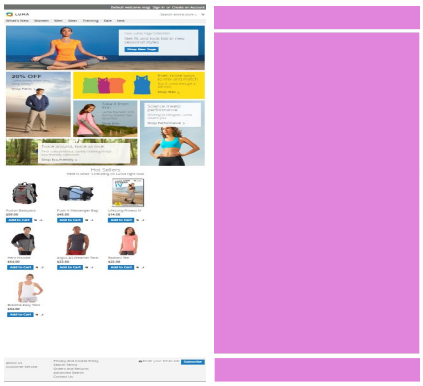
Another way to change the page layout is by applying an XML layout of the target page. For this, we can use a special layout attribute for <page> container, in which we will specify the layout we want to apply to the page:

```
<?xml version="1.0"?>
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
layout="1column"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_configuration.xsd">
   <body>
      // Page layout handles //
   </body>
</page>
```

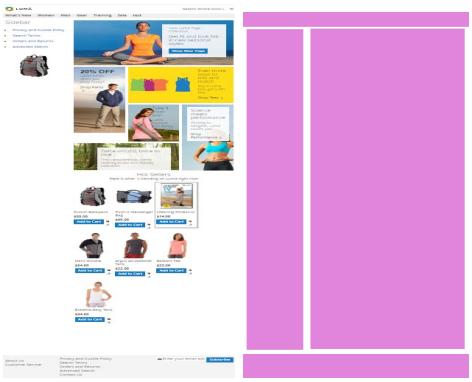
Specify the page layout from which you inherit in **<page>** tag of the **layout** attribute. As a result, the module page with this layout file will have a one column grid; the same as the default Magento 2 theme Luma uses for the main page, but with custom additions and modifications inside the **<body** tag.

What is the purpose of page layouts?

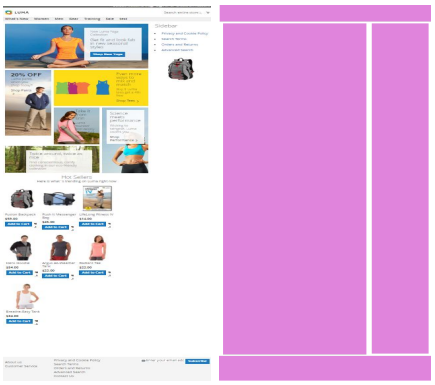
The main purpose of page layouts is to create a structured and united set of template layouts for the page display.
If we apply any of the standard layouts to the main page of the default Luma theme, we get the following results:



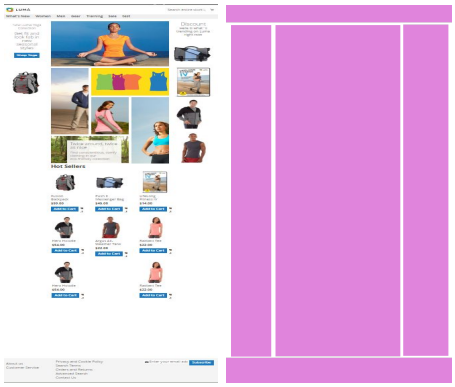
1 column



2 columns with left sidebar



2 columns with right sidebar



3 columns

It is worth mentioning that module-checkout uses a default **empty** template in the structure of the pages, but modified, and its markup will look the following way:

```
<?xml version="1.0"?>
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_layout.xsd">
    <update handle="empty"/>
    <referenceContainer name="page.wrapper">
        <container name="checkout.header.container"
as="checkout_header_container" label="Checkout Page Header Container"
htmlTag="header" htmlClass="page-header" before="main.content">
```

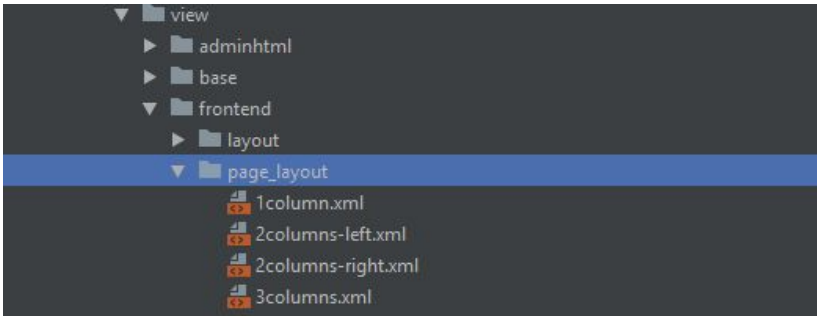
```
            <container name="checkout.header.wrapper" label="Checkout
Page Header" as="checkout_header_wrapper" htmlTag="div"
htmlClass="header content"/>
        </container>
    </referenceContainer>
    <move element="logo" destination="checkout.header.wrapper"/>
</layout>
```

How can a custom page layout be created?

By default, the standard layout files are located at the following path:

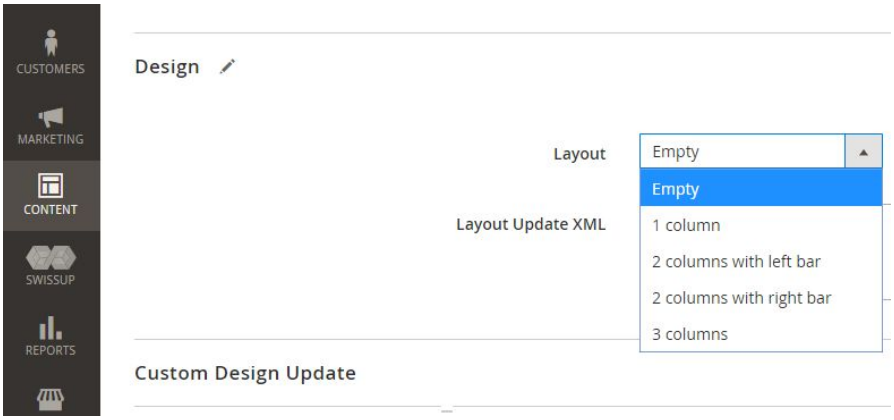
```
vendor ⇒ magento ⇒ module-theme ⇒ view ⇒ frontend ⇒
page_layout
```

Here stored the layout files that we examined above:



Default files in the file system

This is how they look in the admin panel:



Default grid alternatives in the admin panel

In order to apply the custom template file to the custom theme, apply template files at the following address:

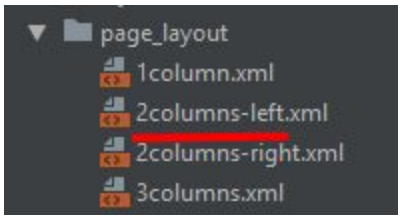
```
app ⇒ design ⇒ [vendor_name] ⇒ [module_theme] ⇒
Magento_Theme ⇒ page_layout
```

For example, many store pages will have two-column structure, without a side menu and breadcrumbs. Therefore, it would be reasonable to create a custom template (let's call it **2columns-not-breadcrumbs.xml** and place it at the path we specified above) with the following markup in it:

```
<?xml version="1.0"?>
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_layout.xsd">
    <update handle="2columns-left"/>
    <referenceContainer name="page.wrapper">
        <referenceBlock name="breadcrumbs" remove="true"/>
    </referenceContainer>
    <referenceContainer name="columns">
        <referenceContainer name="div.sidebar.main" remove="true"/>
    </referenceContainer>
</layout>
```

Example of the page with inheritance template markup

Then, all we need is to apply it to all the pages we need.
In the markup we described above, we applied the method of inheritance from the parent layout. The update directive passes the handle attribute with the layout template name value, which is the same as in the vendor folder of the standard theme, but without ".xml" extension:



How the pages layouts are displayed in the vendor folder

The newly created layout file will recursively adapt the parent markup during the inheritance process, yet it will change all the overridden blocks for the ones set in the child file.

To complete the creation of the custom page layout, declare it in the layout configuration file. In the standard Magento module, it is located at the following path:

```
vendor ⇒ magento ⇒ module-theme ⇒ view ⇒ frontend ⇒
layouts.xml
```

To save it for further work with the composer, place it in the theme file:

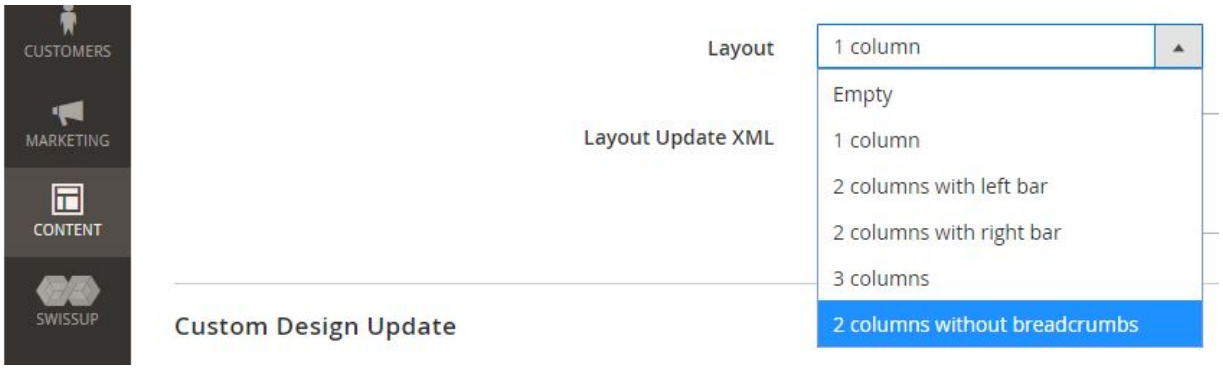
```
app ⇒ design ⇒ [vendor_name] ⇒ [module_theme] ⇒
Magento_Theme ⇒  layouts.xml
```

Then, in **layouts.xml** file template, using the <layout> tag, pass the displayed template name by wrapping it in the <label> tag and, using id attribute, set the value that will be applied at the client side:

```
<?xml version="1.0" encoding="UTF-8"?>
<page_layouts xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/PageLayout/
etc/layouts.xsd">
    <layout id="2columns-not-breadcrumbs">
        <label translate="true">2 columns without breadcrumbs</label>
    </layout>
</page_layouts>
```

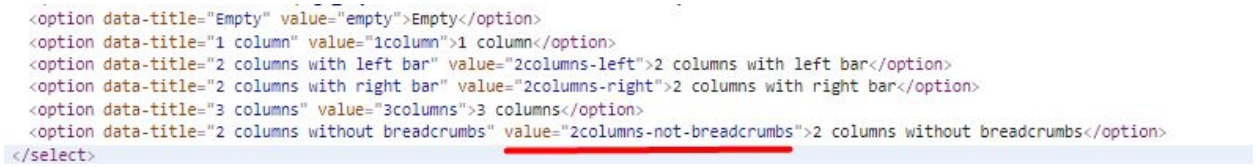
Enhancement of layout configuration file

In the end, we will see a similar result at the admin panel:



A custom grid in the admin panel

The passed id attribute for <layout> tag will be the value of a new layout.



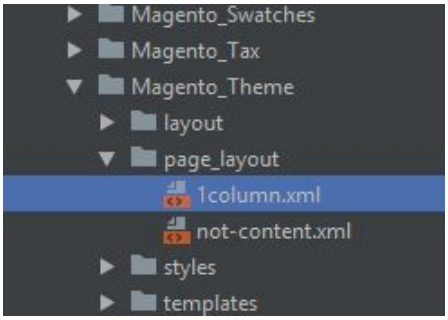
The displayed value at the client side

This is not the only way to override or overwrite the layout file. In the example above described the way to create a custom layout file for all the shop pages; you can also overwrite the existing current layout file the same way by applying the necessary modifications to it. For example, we can create a one-column layout, but without breadcrumbs:

```
<?xml version="1.0"?>
<layout xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_layout.xsd">
   <update handle="empty"/>
   <referenceContainer name="page.wrapper">
       <container name="header.container" as="header_container"
label="Page Header Container" htmlTag="header"
htmlClass="page-header" before="main.content"/>
       <container name="page.top" as="page_top" label="After Page
Header" after="header.container"/>
       <container name="footer-container" as="footer"
before="before.body.end" label="Page Footer Container"
htmlTag="footer" htmlClass="page-footer"/>
   </referenceContainer>
   <referenceContainer name="breadcrumbs" remove="true" />
</layout>
```

File template markup for overriding the standard one-column layout

Placing the file in the theme folder with the name of the target layout and current markup, we have it applied to all the store pages, where this type of grid is used. So, in this case, all the pages that use the one column layout file as a template will use the new overwritten copy of it. Bear in mind that, according to the described above inheritance rules and all the priorities, all the layout files that are located in the current theme and inherited from 1 column layout, will have the overwritten file as the parent.



Position of the template file for override of the 1 column template

Where are the existing page layouts used?

Magento 2 has 5 default page layout files; using the Luma theme as an example, let us examine what store pages they are applied on.

- **empty (page without containers)**
  checkout pages (update from empty)
- **1 column (one common column for the content)**
  home page,product-view, all-cms-pages, cart, login-page, success page
- **2 column with left bar (content container and a left aside bar)**
  what-is-new, category-view, subcategory-view, account-pages,
- **2 column with right bar (content container and a right aside bar)**
  no-route-page (404),
- **3 column (3 containers optionally)**

How can the root page layout be specified for all pages and for specific pages?

As we have already mentioned, we can apply one of the existing or created page layouts (from page_layout folder) using <page> layout tag directive. This can be done for certain pages as well as for all the website pages. For

example, we can apply the layout page file for all the catalog products (using catalog_product_view.xml file), while we also can apply it only for a certain product by using its ID (in catalog_product_view_id_xxx.xml file, where xxx stands for product id) or for a product of a certain type (for example, catalog_product_view_type_simple.xml file is used for simple product type)

3.3 Demonstrate understanding of layout handles and corresponding files

What is the purpose of layout handles?

Handle is the tag similar to the <block> and <container>, inserted into theme xml configuration with an aim to target the application of the nested instructions to a certain page or element category.

Handles can have the form of not only tags in xml configuration, but separate files placed in layout modules' folder as well. The example of such file:

```
catalog_product_view_id_112.xml
```

This method is more useful for mass modifications, when there are too many records in one descriptor to place it in a common document. Also, this way is better for data structuring. Instead of searching through one large document with a number of descriptors, looking for the one you need, you can create an orderly data structure in the file system, where looking for the needed descriptor will be a simple task.

How can the available layout handles for a given page be determined?

In order to determine how to set the descriptor name for the custom page or help find the tag of the existing, let us get into the details of its name creation. This is what we use as an example:

```
<cms_index_index>
```

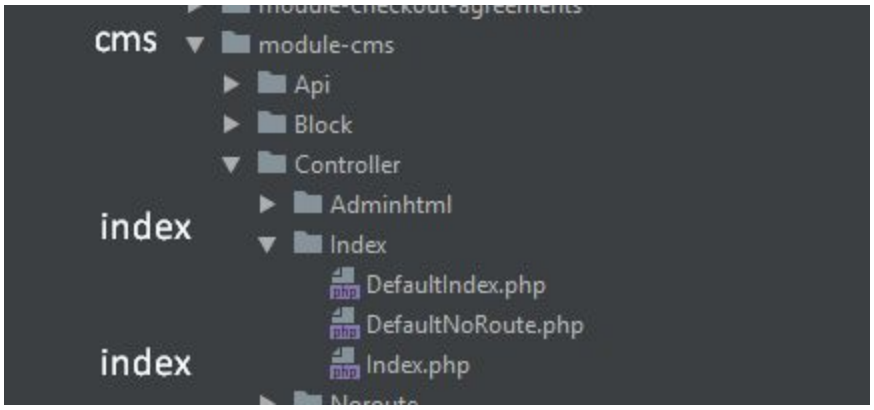
- **cms** - route identifier.

It is applied to compare the URL addresses with the controller from the module folder which is responsible for the given descriptor. In this case, this is a standard CMS-module.

Bear in mind that the final URL address can be re-assinged,while the current one is used solely for working with a controller.

- **index -** the folder on the physical path to the controller
- **index -** controller file

When we examine the directory structure in the module folder more closely, the principle of descriptor forming becomes clear:



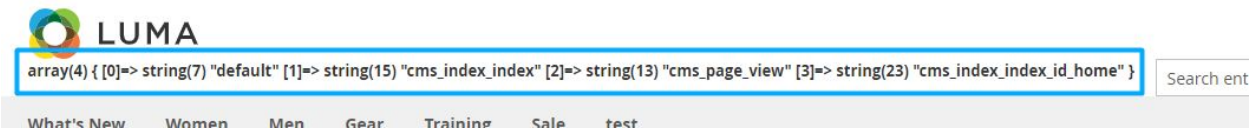
Module-cms controller structure

This principle is embedded in forming of any name, so knowing it will significantly simplify the descriptor search for the needed page. However, in case a developer needs help with searching for a new descriptor or in the situation when it is unclear what layout file has a priority over the given page, there is a more reliable way of displaying all the current descriptors directly at the screen. Add the following php script to any template file connected to the needed page:

```
<?php var_dump ($block -> getLayout() -> getUpdate() ->
getHandles())?>
```

Php script for displaying all the descriptors for the current page

Taking the default Luma theme homepage as an example, let us have a look on what we got:



Each template has a method that allows to get the name of the root file template:

```
getLayout()
```

On this page, it's default.xml, then using the method:

```
getUpdate()
```

We get the list of the layout files from which inheritance is performed (cms_index_index and cms_page_view) with the method:

```
getHandles()
```

As a result, we get cms_index_index_id_home descriptor.
It is worth mentioning that they are listed according to their priority.

How do you add a new layout handle?

To create a new custom layout handle, use in the new layout file the handle directive we are already acquainted with:

```
<update handle = "handle_name" />
```

When the Layouts cache is disabled, we can use a new descriptor name for working with layouts.

What are the most commonly used layout handles?

Below is a table with the most commonly used layout handled:

XML descriptor	Target category	Source
<default>	All pages	default.xml
<cms_index_index>	Main page of the website	cms.xml
<catalog_category_default>	Product categories	catalog.xml
<customer_account>	Customer account	customer.xml
<catalog_product_view>	Product view page	catalog.xml
<cms_page>	Custom pages created with WYSIWYG editor ir in a file system, or so called CMS page.	cms.xml

<1column> <2column> ...	Only the pages with the target grid type	1column.xml 2column.xml ...
catalog_product_view_type_simple(configurable)	Configured or simple product view pages	catalog.xml
catalog_product_view_id_(number_id) catalog_product_view_sku_(number_sku)	View pages of the product with a certain ID or SKU number	catalog.xml

Layout descriptor type

How can layout handles be used during theme customization?

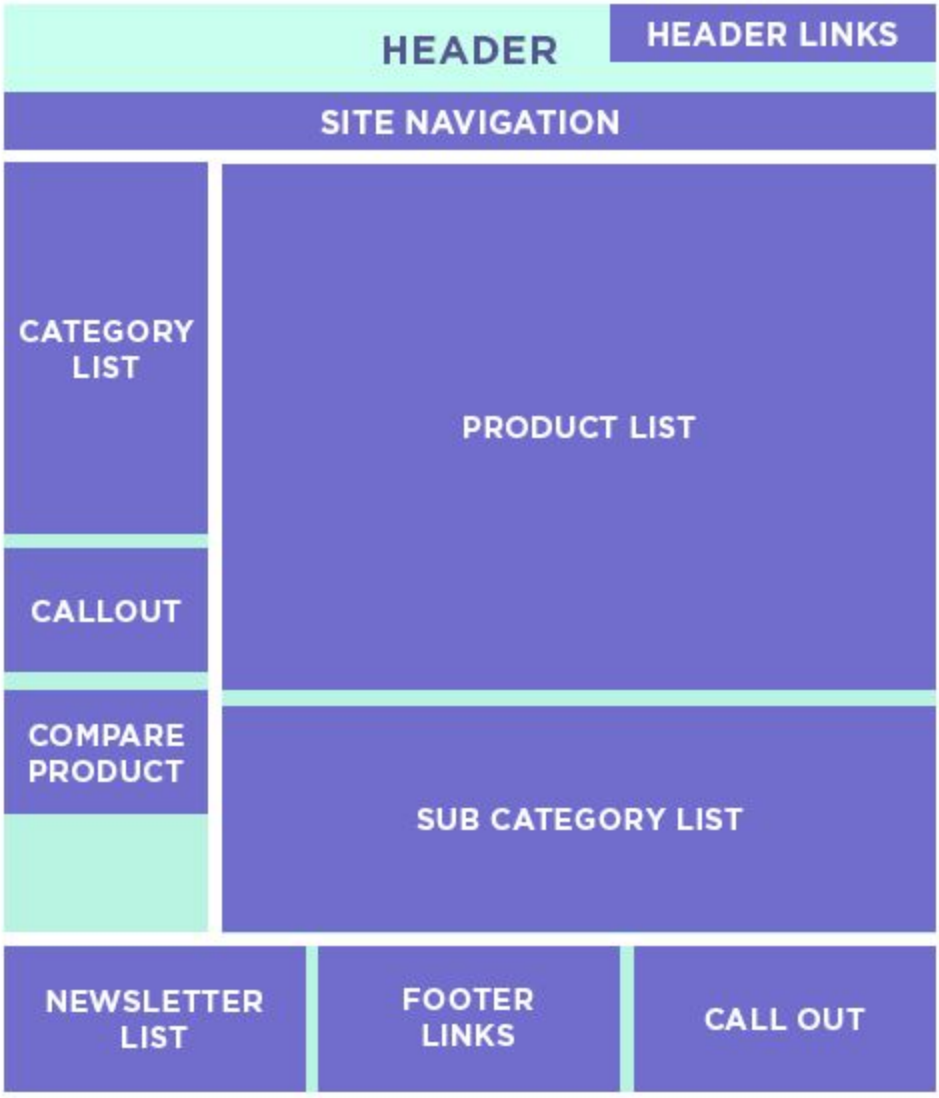
Layout handles are applied to group layout instructions for certain pages and elements. This is a very useful functionality, for it allows to quickly create and configure page and page elements structure. It can be applied for a number of similar product pages as well as for the selected pages.

For example, we use layout handle catalog_product_view.xml for all the products in the catalog. Let us imagine we want to add a CMS block for all simple product pages. For this, we use layout handle catalog_product_view_type_simple.xml. Or, let us imagine another situation, when we wish to modify the layout of a single page; for this, we will use layout handle catalog_product_view_id_112.xml.

3.4 Understand the differences between containers and blocks

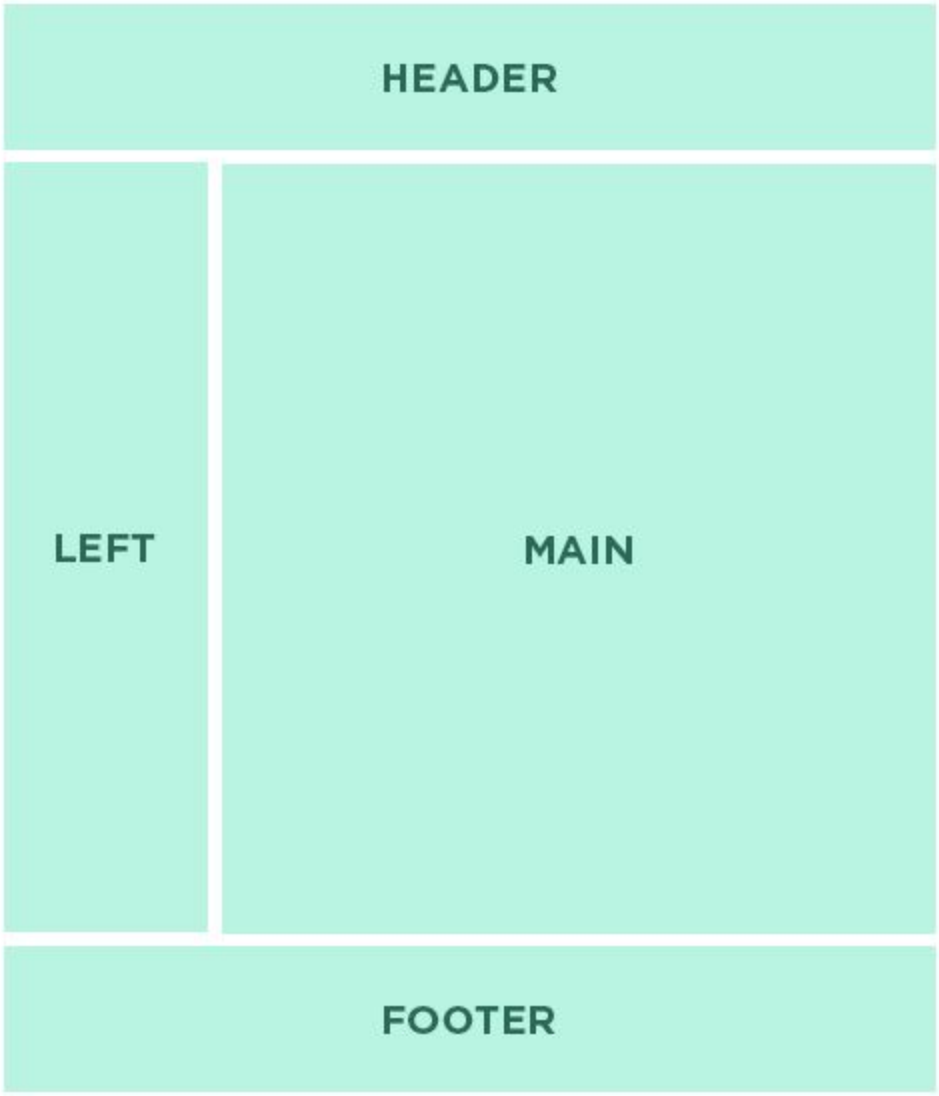
What is the purpose of blocks?

Content is added to containers by means of blocks, which are files with templates, in which html code is generated. Site navigation, product list, category list, footer links are some of the most frequently used page blocks.



What is the purpose of containers?

Containers are what page framework is built from. They are the basic elements of Magento page structure. Header, footer, left sidebar, main column are the main containers of a Magento webpage.



How can containers be used in theming?

Containers make up the page framework. They can contain child elements, such as blocks and other containers, but can stay empty as well. Here you can see an example of a container and block applied in layout:

```
<container name="category.view.container" htmlTag="div"
htmlClass="category-view" after="-">
```

```
<block class="Magento\Catalog\Block\Category\View"
name="category.image"
template="Magento_Catalog::category/image.phtml"/>
<block class="Magento\Catalog\Block\Category\View"
name="category.description"
template="Magento_Catalog::category/description.phtml"/>
<block class="Magento\Catalog\Block\Category\View"
name="category.cms" template="Magento_Catalog::category/cms.phtml"/>
</container>
```

Containers have their own attributes that allow you to manage it. Let us briefly examine each of them.

- Name. An obligatory attribute. This identifier can be used to address a certain container. Name attribute should be unique within a single page. This attribute could be compared with id for html tags.
- Label. The signature that will be displayed in the browser.
- Before/after. These attributes allow placing the current container before or after an element, which name is specified in this attribute. If the attribute has “-” value, then the container is placed according to the first or last one on its level of nesting.
- As. Assigns with an alias, on which the current container is identified in the range of its parent.
- Output. Defines whether to output the root element.
- Html Tag. Specifies which tag to use to wrap a container.
- HtmlId. Specifies id attribute value for a tag that wraps a container.
- HtmlClass. Specifies class attribute value for the tag that wraps a container.

How can blocks be used in theming?

Blocks are applied to output a content according to the template. In the code we placed above, you can see how are the blocks displayed in the layout file. Each block, as well as a container, has its own attributes, that allows to manage it. Let's consider these attributes.

- **Class**. Specifies a class on which rendering of this block is implemented.
- **Name**. This attribute is obligatory. This is an identifier which can be used to address to the certain block. Name attribute should be unique within a single page. This attribute could be compared with id for html tags.
- **Before/after**. These attributes allow placing the current block before or after an element, which name is specified in this attribute. If the attribute has "-" value, then the block is placed according to first or last on its level of nesting.
- **Template**. Specifies a path to a template of the current block.
- **As**. Assigns with an alias, on which the current block is identified in the range of its parent.
- **Cacheable**. Decides whether to cache the current block or not.

What is the default block type?

Block type is defined by "class" attribute.
For example, for <block class="Magento\Framework\View\Element\Template" name="store_account" template="Magento_Theme::html/account.phtml"/> block it will be class="Magento\Framework\View\Element\Template"
For instance, for

<block class="Magento\Framework\View\Element\Template" name="store_account" template="Magento_Theme::html/account.phtml"/> block it will be class="Magento\Framework\View\Element\Template".

In this attribute, we set the name of the class that realizes the block rendering. The object of this class is responsible for the block output visualization.
The default block class in Magento 2:
“Magento/Framework/View/Element/Template”

There are other classes in Magento as well; moreover, you can create a custom class. Bear in mind that all custom classes should extend from the default class.
To view the file content for the class, follow the path “/vendor/magento/framework/View/Element/Template.php”

Or go to github - https://github.com/magento/magento2/blob/2.2-develop/lib/internal/Magento/Framework/View/Element/Template.php

How can the order of rendered child blocks be influenced both in containers and in blocks?

There is a **Before/after** attribute both for container and blocks. This attribute sets the order in which blocks and containers are displayed on a page.

3.5 Describe layout XML override technique

How can layout overriding be used in theming?

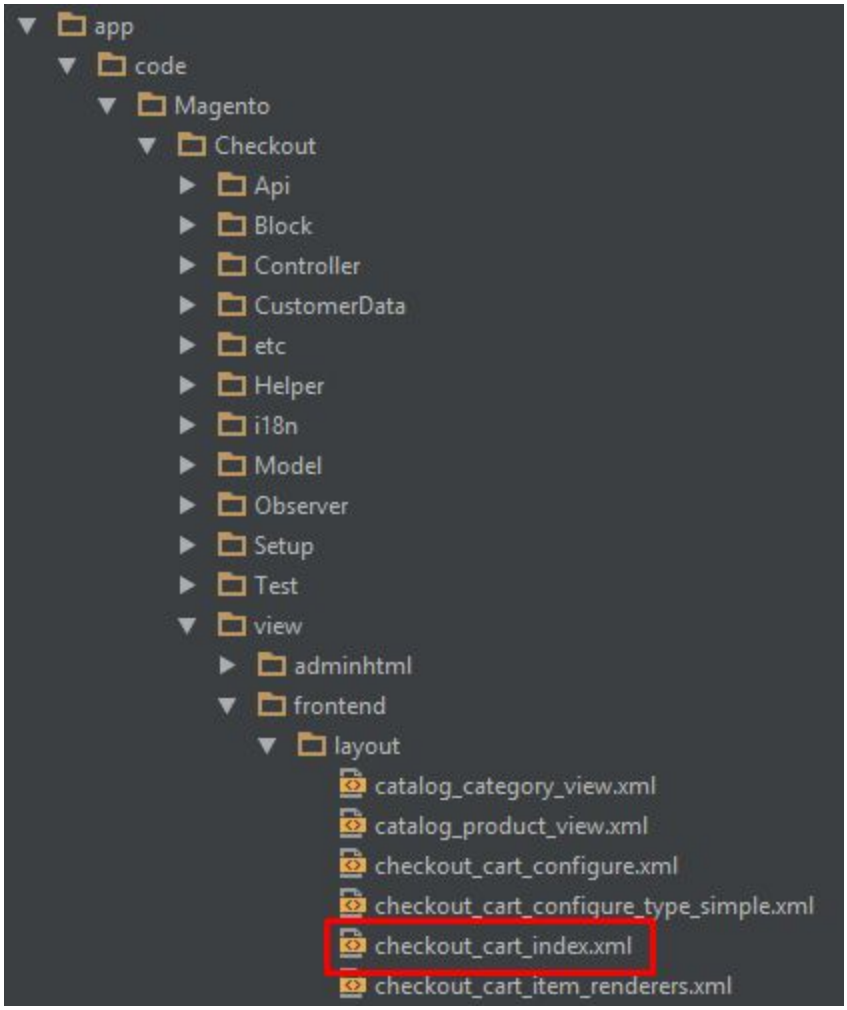
Typically, the extension of the file layout is good enough to solve most of the development tasks. However, if you need to make a lot of changes or the layout file contains an instruction that can not be changed in the file extension, then the only option left will be to override such file.

How can layout XML be overridden?

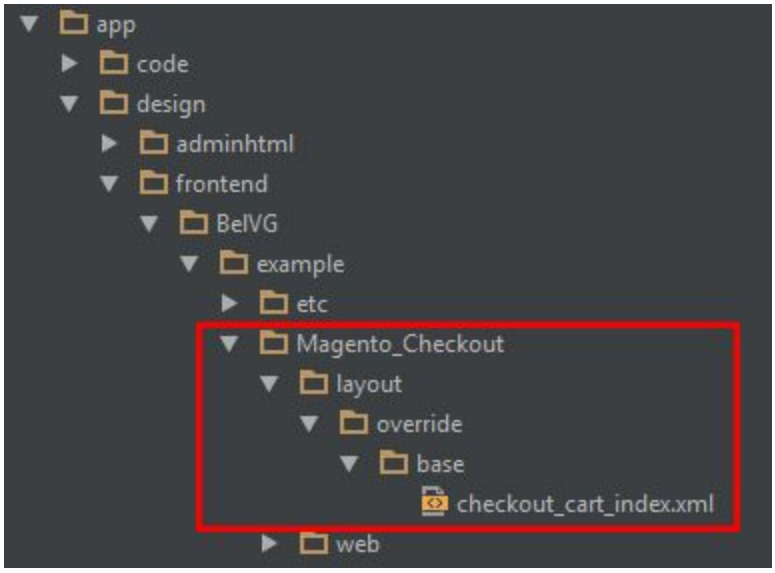
To perform overriding, create an overriding layout file in the directory of your theme. You can override base layouts and theme layouts as well. The mechanism is pretty similar, but there are still some differences. Let us consider the examples to understand the processes better.

Creating overriding base layout file

For instance, we need to override base file *checkout_cart_index.xml* in Checkout_Module (see the picture below)



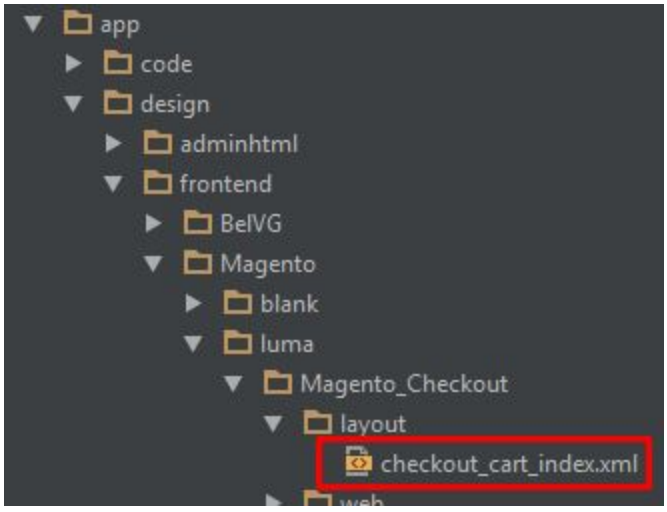
For that, we must create a file with the same name in our theme directory:



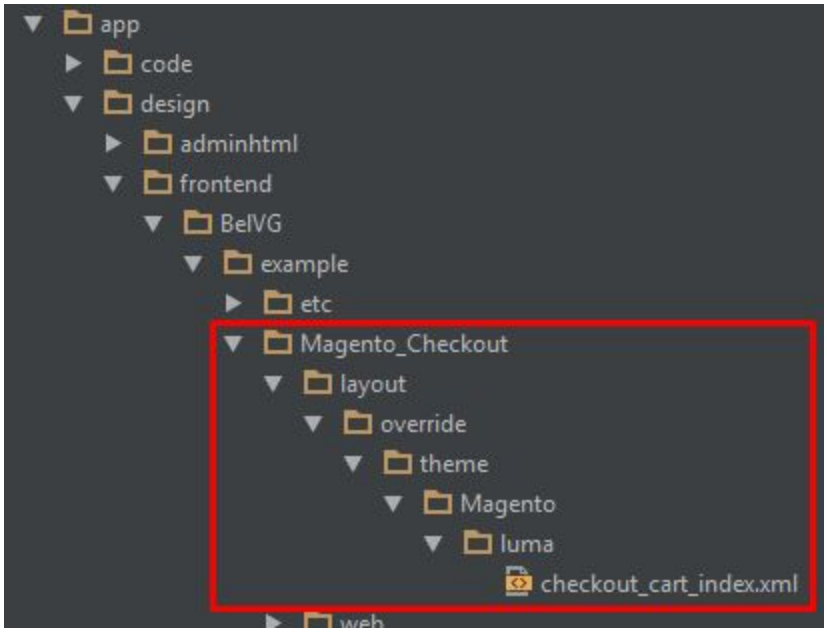
Similarly, it is possible to do so for other modules.

Creating overriding theme layout file

For example, we need to override checkout_cart_index.xml file in Magento_Checkout module of parent theme.



To achieve that, we must create a file with the same name in our theme directory:



Similarly, it is possible to do the same for the other modules.

What are consequences of layout overrides during upgrades?

When Magento is updated, the layout modules' root files can be updated as well. New blocks and containers may appear in them and can be referenced by other modules. If you have overridden a file where such blocks or containers appeared during the update, the modules referring to these elements can not access them, which will lead to errors in the operation of the modules and eventually hinder their functionality. In order to avoid this, you will be forced to add these new elements to your overriding layout file. You need to remember this when you override some layout file in your theme and are going to update Magento after that.

What is the effect of layout overrides on compatibility?

Overriding layout files provides ample opportunities for flexible theme customization. However, you need to remember that the overridden layout file is not affected by any changes to the main files, which in turn can lead to future problems with upgrade if you do not keep a close watch on the updated files.

Therefore, overriding should be used very carefully and only when it can not be done without.

Here are some guidelines that we want to share with you:

- Use overriding in case you can not afford the desirable effect through extending;
- Do not change names and aliases of blocks;
- Do not change handle inheritance.

3.6 Understand layout merging

What is layout merging?

To explain the notion of layout merging, let us consider its process. First, all layout XML files are united into one XML document with “_loadFileLayoutUpdatesXml”. Then, all layout instructions for certain handles are found with “_fetchPackageLayoutUpdates” method and merged. As a result, we get a final XML document. Magento reviews this document and creates PHP classes for each block. Afterward, the result for

each certain page is rendered, forming the html-tree of the page and adding css-styles and JavaScript files.

Bear in mind that during the process of layout merging, the layouts are merged in a certain order. The order is determined by the modules loading order (to learn more about it, follow the link https://devdocs.magento.com/guides/v2.2/extension-dev-guide/build/module-load-order.html). In the "app/etc/config.php" file, you can also see the order in which the modules are currently loaded. In case when in layout files two or more conflicting instructions from different modules meet, then the upper hand has the instruction whose module is loaded the latest.

What is also worth mentioning in the context of layout merging is layout files extending and overriding. When we create extending layout file, it will complement the file that it extended. In case the extending file will contain the instructions that contradict the initial file instructions, the contradicting ones will be overridden during the layout merging. For example, when the initial layout file had the following instruction

```
<move element="form.subscribe" destination="footer_row"
after="footer_column_4"/>
```

And the file from our theme, that extended the initial file, will have the following instruction:

```
<move element="form.subscribe" destination="footer_row"
after="footer_column_5"/>
```

As a result, the "form.subscribe" element will be placed into a "footer_row" after "footer_column_5" element, not the "footer_column_4" one.
In case we use overriding layout files, then the initial file that we have overridden is not included into the layout merging at all. Still, one must use overriding carefully; otherwise, certain issues may occur during Magento update (to learn more, read the article https://belvg.com/blog/override-a-layout-in-magento-2.html).

On the surface of it, it may be unclear how the final merged XML should look like. To see the layout merging merging, do the following:

- Enter the “magento\framework\View\Result\Page.php” file;
- Find $output = $this->renderPage() line in render() method;
- Add the $output .= $this->getLayout()->getXmlString() command after this line.

Afterward, go to your store’s front end and find the final XML for a certain page in the “Source” folder in the browser development instruments (after <html> tag).

How do design areas influence merging?

After the request to the current page (either front end or admin panel) is received, Magento first merges layout files into the base area and then into the area applied to the current request (frontend or adminhtml correspondingly).
For the sake of clarity, let us overview the general order of layout merging:

1) Basic module files are loaded
2) Request area modules are loaded (frontend or adminhtml)
3) Theme (basic, parent or current) files are loaded. The more recent the theme is, the higher priority its instructions have during extending and overriding files. The current theme has the highest priority.

How can merging remove elements added earlier?

The “remove” attribute for “referenceBlock” and “referenceContainer” tags correspondingly is applied to remove the elements added earlier.

It looks the following way:

```
<referenceBlock name="breadcrumbs" remove="true"/>
```

and

```
<referenceContainer name="header.container" remove="true"/>
```

What are additive changes and what are overriding changes during layout merging?

Additive changes is adding new elements to the existing XML elements (for example, adding new arguments to the existing element). Overriding changes, in its turn, is when you change the existing element for a new one (for example, the current argument value for a custom one).

Let us demonstrate additive changes with the following example.

For instance, we have "register-link" block with "label" attribute that has “Create an Account” value:

```
<block class="Magento\Customer\Block\Account\RegisterLink"
name="register-link">
      <arguments>
            <argument name="label" xsi:type="string" translate="true">
Create an Account
            </argument>
       </arguments>
</block>
```

To add a new "test-argunemt" argument for "register-link" block, add the following code to your existing layout file:

```
<referenceBlock name="register-link">
   <arguments>
      <argument name="test-argunemt" xsi:type="string"
translate="true">
Value Test Argument
      </argument>
   </arguments>
</referenceBlock>
```

Below is the example of overriding changes.
To modify the existing “label” argument for the "register-link" block, write the following code in existing layout file:

```
<referenceBlock name="register-link">
   <arguments>
      <argument name="label" xsi:type="string" translate="true">
New label value
      </argument>
   </arguments>
</referenceBlock>
```

3.7 Understand processing order of layout handles and other directives

In what order are layout handles processed?

First, default.xml handles of all modules are loaded, regardless of that website page being loaded at the moment (the order of default.xml handle merger for the corresponding modules will be outlined in the next

paragraph). Then, the rest of the handles are loaded in the order they are called for a certain page.

The order of handle calls corresponds to the order in which the handles are added to Merge class (you can see the class file here - https://github.com/magento/magento2/blob/2.2-develop/lib/internal/Magento/Framework/View/Model/Layout/Merge.php).

In this file, load() method reviews and merges all the connected layout handles. Also, bear in mind that handles can be added to other handles via the layout instruction **update**. In this case, the content of the added handle will be added to the layout handle.

In what order is layout XML merged within the same handle?

In short, Layout XML is merged in the order in which the corresponding modules are loaded. In case two conflicting layout instructions from different modules meed, the one from the most recently loaded module will have the priority. If the same-name layout files meet in several inherited themes, they will be merged, and in case the layout instructions conflict occurs, the theme closest in terms of inheriting to the current theme will have priority over all (the maximum priority has the current theme).

To learn more about layout merging, explore the article: https://belvg.com/blog/magento-2-certification-layout-merging-overview.html

How can the processing order be influenced?

As it was mentioned before, the order of layout instruction execution is determined by the order of modules loading. Therefore, when you change the order of modules loading, you influence the layout instructions execution. The order of modules loading is determined by the <sequence> parameter in <Module_folder>/etc/module.xml module file.

To learn more, follow the link:
https://devdocs.magento.com/guides/v2.2/extension-dev-guide/build/module-load-order.html

Using plugins, you can also embed your layout handles into any place of the layout handles loading process. For example, you can add a custom layout handle before the handle of the catalog category page is loaded - catalog_category_view. For this, create a corresponding before plugin.

To learn more about creating such plugins, follow the link:
https://devdocs.magento.com/guides/v2.3/frontend-dev-guide/layouts/xml-manage.html#layout_markup_modify_with_plugins

What are common problems arising from the merge order of layout declarations?

If you follow all given recommendations for working with layout files, then no problem will occur with merging. However, one can make an error due to the lack of attention during the development. Let us consider the most common mistakes that occur this way.

1. Blocks and containers naming. When you make a mistake in the name of a block or container you want to override or create, you do not get the result you intended for and created additional errors instead. Here are a couple of typical examples of this error:

- You mistakenly name the new block with the name of the existing block. As a result, the existing block is deleted.
- You want to override the existing block and create a new block with the same name as the existing, but you make a mistake in spelling and simply create a new block.
- You want to add a block into the existing container, but you make a mistake and the block is not added anywhere (if there is no container with such a name), or added into the container where you did not want it to be.

2. When you remove or rename certain blocks during the layout files overriding, errors can occur due to the fact that the blocks will be used in other layout files and will not be accessible to you. Keep this in mind when you decide to override a layout file.

3.8 Set values on block instances using layout XML arguments

How can arguments be set on blocks?

The arguments can be set on blocks with the help of the following code in layout:

```
<arguments>
   <argument name="..." xsi:type="...">
   ...
   </argument>
</arguments>
```

For examples, to set an argument named “example_argument” with the value “Example argument value” on a block named “example-block”, you should use the following code in layout:

```
<referenceBlock name="example-block">
   <arguments>
      <argument name="example_argument" xsi:type="string">
      Example argument value
      </argument>
   </arguments>
</referenceBlock>
```

Get more details in this article in the section “**Argument**”: https://belvg.com/blog/layout-xml-directives-and-their-arguments-in-magento-2.html

Which data types are available?

The following data types are available for an argument:

- string
- boolean
- object
- number
- null
- array

Get more details in this article in the section “**Argument**”: https://belvg.com/blog/layout-xml-directives-and-their-arguments-in-magento-2.html

What are common arguments for blocks?

Let’s take a look at the most common arguments for blocks and discuss where you can use them:

css_class - it is used to set a “class” attribute on element.

Example:

Let’s say you want to add a class “product-test-example” to the title of the product page.

In order to do it, you need to add the following code in your theme (file *app/design/frontend/Example_Vendor/Example_Theme/Magento_Catalog/layout/catalog_product_view.xml*) in product page layout:

```
<referenceBlock name="page.main.title">
   <arguments>
      <argument name="css_class"
xsi:type="string">product-test-example
      </argument>
   </arguments>
</referenceBlock>
```

As a result, the title of the product page acquires a class “product”:

```
<div class="page-title-wrapper product-test-example">
   <h1 class="page-title">
      <span class="base" data-ui-id="page-title-wrapper"
itemprop="name"> Product example name
      </span>
   </h1>
</div>
```

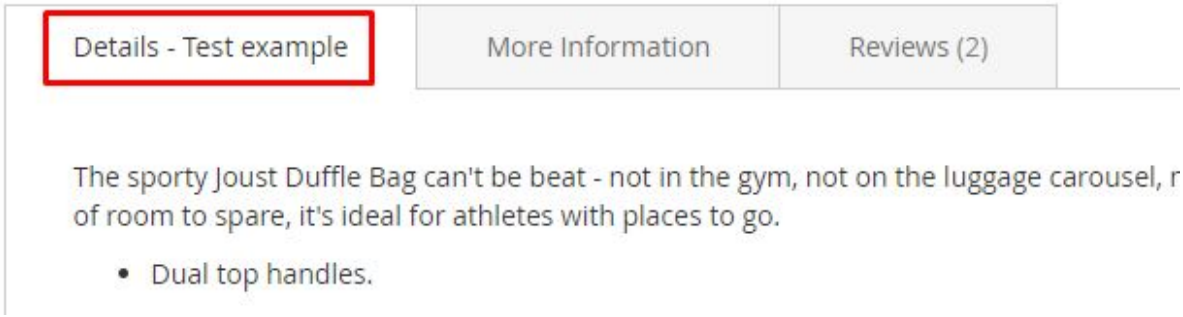
title - it is used to set titles on different blocks.

Example:

Let’s say you need to change the title of the section with the full description on the product page. In order to do it, you need to add the following code in your theme (file *app/design/frontend/Example_Vendor/Example_Theme/Magento_Catalog/layout/catalog_product_view.xml*) in product page layout:

```
<referenceBlock name="product.info.description">
   <arguments>
      <argument name="title" translate="true"
xsi:type="string">Details - Test example
      </argument>
   </arguments>
</referenceBlock>
```

As result, the section with the full product description will be titled “Details - Test example”:



label - it is used to label different blocks.

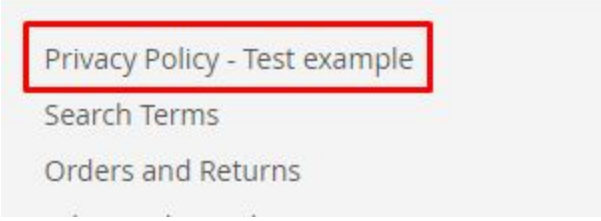
Example:

Let’s say you need to change the name of the link to Privacy policy in the footer. In order to do it, you need to add the following code to layout

default.xml in your theme (CMS *app/design/frontend/BelVG/belvgtheme/Magento_Cms/layout/default.xml*):

```
<referenceBlock name="privacy-policy-link">
   <arguments>
      <argument name="label" xsi:type="string"
translate="true">Privacy Policy - Test example
      </argument>
   </arguments>
</referenceBlock>
```

As a result, the name of the link in footer will change to “Privacy Policy - Test example”:



path - it is used to set the path for different links.

Example:

Let’s say, you want to change the path to the Privacy policy page from the previous example. In order to do it, you need to add the following code to layout default.xml in your theme (for example, module file Theme *app/design/frontend/BelVG/belvgtheme/Magento_Theme/layout/default.xml*):

```
<referenceBlock name="privacy-policy-link">
   <arguments>
      <argument name="path"
xsi:type="string">privacy-policy-test-example
      </argument>
```

```
    </arguments>
</referenceBlock>
```

As a result, the site address in footer changes:

```
<ul class="footer links">
    <li class="nav item">
        <a
href="your-site-address/privacy-policy-test-example/">Privacy Policy
- Test example
        </a>
    </li>
...
</ul>
```

type - it is used to set different block types. Depending on which block type is set, certain block template will be uploaded.

Example:

Depending on which “type” argument is set on the block, the templates for product display will differ:

```
<referenceContainer name="content.aside">
    <block class="Magento\Catalog\Block\Product\ProductList\Related"
name="catalog.product.related"
template="Magento_Catalog::product/list/items.phtml">
    <arguments>
        <argument name="type" xsi:type="string">related</argument>
    </arguments>
    ...
    </block>
</referenceContainer>
```

In the example described above, related products will be displayed according to the template which corresponds to the “related” argument value in the template file:

“vendor/magento/module-catalog/view/frontend/templates/product/list/items.phtml”.

Besides the block arguments, there are plenty of other arguments. Moreover, you can create your own argument and use it in templates .phtml with the help of get{ArgumentName}() and has{ArgumentName}() methods, as it was described above.

3.9 Customize a theme's appearance with etc/view.xml

What is the etc/view.xml file used for?

The **etc/view.xml** file is located in theme folder:

```
<theme_folder>/etc/view.xml
```

Properties of the product catalog’s images, settings of the product catalog’s image gallery and settings of JavaScript bundling are located in this file. We are going to take a closer look at these settings and properties in the next section of our tutorial.

Based on the above, this file is used to set themes, to create and to use custom properties when working with a theme.

How can it be used to customize a theme?

In order to use this file, you need to create it in your theme. For example, in theme Example_Theme of the vendor Example_Vendor, it can be found at:

```
app/design/frontend/Example_Vendor/Example_Theme/etc/view.xml
```

After you have created a file in your theme, you can change the properties value and this way customize the theme as you want.

Let's see what properties are available for us in this file (the example of such a file you can see in the default Blank theme https://github.com/magento/magento2/blob/2.0/app/design/frontend/Magento/blank/etc/view.xml).

In order to change the image properties, you need to use ***<images module="Magento_Catalog">...</images>*** element content. Elements act as content for this element ***<image id="image_id" type="image_type">...</image>***.

Here is an example of how it looks in the etc/view.xml file:

```
<images module="Magento_Catalog">
    ...
        <image id="category_page_list" type="small_image">
            <width>240</width>
            <height>300</height>
        </image>
    ...
</images>
```

Here **id** and **type** are attributes of the <image> element.

id is a unique identifier for image properties (it should not be repeated). With the help of this identifier, you can get the image and its properties in the .phtml template.

type is the type of the image. There are the following types in Magento: *image, small_image, swatch_image, swatch_thumb, thumbnail*. The role of the image in the website page templates depends on the image type (as, for example, the image with the thumbnail type will be used as a thumbnail image - shrinked view of a picture).

In the example above, **width** and **height** are the properties of the <image> element and there can be more of these properties. Let's take a look at all the available properties of the <image> element:

width is the image width in pixels.

height is the image height in pixels.

constraint property is responsible for how the image acts when its size is smaller than the one set by "width" and "height" properties. Possible values ***true*** and ***false***. If the value ***true*** is selected, the image will not get bigger up to the required size when it starts losing quality. If the ***false*** value is selected, it will do so. By default, the value ***true*** is selected.

aspect_ratio property is responsible for whether the aspect ratio of an image will change to match the values set by the "width" and "height" properties. Possible values ***true*** and ***false***. If the value ***true*** is selected, the image's aspect ratio will not change. If the ***false*** value is selected, it will do so. By default, the value ***true*** is selected.

frame property is responsible for whether the image will be cropped to match the values set by the "width" and "height" properties. Possible values

true and ***false***. If the value ***true*** is selected, the image will not be cropped. If the ***false*** value is selected, it will happen. By default, the value ***true*** is selected. This property is applicable only if the ***aspect_ratio*** property has the the value ***true***.

transparency property is responsible for keeping transparency by the downloaded image. Possible values ***true*** and ***false***. If the value ***true*** is selected, the transparency remains. If ***false*** is selected, the background will be as it is set in ***background*** property. By default, the value ***true*** is selected.

background is the background color for downloaded images. It is set in rgb format [xxx, xxx, xxx]. By default, the value is [255, 255, 255] which means the white color in rgb format. This property is applicable only if the **transparency** property has a ***false*** value.

In variables in the ***<vars module="Magento_Catalog">...</vars>*** element, you can set the properties for the image gallery on the product page. It looks like this:

```
<vars module="Magento_Catalog">
   <var name="gallery">
      ...
         <var name="nav">thumbs</var>
      ...
   </var>
</var>
```

In this example, **name="nav"** property is responsible for the style of the gallery navigation (there can be thumbnails, dots or just nothing). Here we see thumbnails.

There is a variety of such properties for the gallery. You can find them in the file of one of a standard theme (the property comments will help you find out what they are needed for) or you can read about them here - https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_gallery.html

Also, there are settings for **JavaScript bundling** in the **etc/view.xml** file. In a nutshell, it is a feature that gathers multiple JavaScript files in one. Eventually, it splits into several set sizes and loads when the website loads. It is necessary to reduce the number of file requests and as a result, to increase page loading speed.

There are the following settings in this file:

- The list of files which are excluded from JavaScript bundling (They are loaded by a separate request and are not included into output files). This list is in the **<exclude>...</exclude>** element and the path to every JS file from this list is in the **<item type="file">...</item>** element:

```
<exclude>
   <item type="file">path_to_JS_file</item>
</exclude>
```

- The size of the file which the output JavaScript file splits into while loading. (It increases the load speed of the output file). The following property is responsible for it:

```
<vars module="Js_Bundle">
   <var name="bundle_size">1MB</var>
</vars>
```

Here the **bundle_size** is the size of this file.

Get more details about **JavaScript bundling** here - https://devdocs.magento.com/guides/v2.3/frontend-dev-guide/themes/js-bundling.html

How can values from etc/view.xml be used during theming?

You can use the value of the properties from **etc/view.xml** in the .phtml files of module templates.

You can get the value of any property from the **etc/view.xml** file with the help of **getVar()** method**.**

Here is an example. Let's say you need to get the value type of the gallery navigation. In the **etc/view.xml file**, this property is set in the following way:

```
<vars module="Magento_Catalog">
  <var name="gallery">
      ...
         <var name="nav">thumbs</var>
      ...
   </var>
</var>
```

In the *gallery.phtml* file, we can get this value in the following way:

```
$block->getVar("gallery/nav")
```

You can also create your own variables in **etc/view.xml** and use them in .phtml template files.

How does theme inheritance influence values from etc/view.xml?

The way inheritance works in Magento for the **etc/view.xml** files for current and parent themes is similar to the inheritance of extending layout files for current and parent themes. In other words, if there are no variables or properties in the **etc/view.xml** file of your theme, then Magento will search for these variables or properties in its parent theme and then in the parent theme of the parent theme and so on until it reaches the theme without the parent one. It means that there is no point in copying the **etc/view.xml** file fully, it should be enough to copy the part which will differ in comparison with your theme's parent themes. The rest Magento will get from the parent themes.

If the inherited themes will include the properties and variables, the values of which are different, the value which is the closest to the current theme in the inheritance chain will be set (the current theme has the highest priority)

Section 4: Create and Customize Template Files

4.1 Assign a customized template file using layout XML

How can a customized template file be assigned to a block using layout XML?

Let’s say you want to assign a customized wishlist template.

You need to copy the standard file of the wishlist module.

```
<Magento_folder>\vendor\magento\module-wishlist\view\frontend\templates\view.phtml
```

Paste it to following folder:

```
<Vendor>_<Module>/view/frontend/templates\view.phtml
```

Then, you need to create a file:

```
<Vendor>_<Module>/view/frontend/layout/wishlist_index_index.xml
```

```
 <?xml version="1.0"?>
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_configuration.xsd">
<body>
 <referenceBlock name="customer.wishlist">
   <arguments>
     <argument name="template"
xsi:type="string">Vendor_Module::view.phtml</argument>
   </arguments>
 </referenceBlock>
</body>
</page>
```

Another way of changing the template:

```
 <?xml version="1.0"?>
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_configuration.xsd">
<body>
<referenceBlock name="customer.wishlist"
template="Namespace_Module::new_template.phtml" />
</body>
</page>
```

And there is also an old method:

```
 <?xml version="1.0"?>
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
```

```
page_configuration.xsd">
<body>
 <referenceBlock name="customer.wishlist">
   <action method="setTemplate">
     <argument name="template"
xsi:type="string">Vendor_Module::view.phtml</argument>
   </action>
 </referenceBlock>
</body>
</page>
```

There is one more step to make your overriding come into force. You should add a sequence in your module.xml file for the module that contains the changing template file. In this example, your etc/module.xml file will look like this:

```
 <?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:Module/etc/modul
e.xsd">
<module name="Vendor_Module" setup_version="100.0.0">
 <sequence>
   <module name="Magento_Wishlist"/>
 </sequence>
</module>
</config>
```

It ensures that the Magento_Wishlist module will be downloaded and added to the common template file before your module. If you need to assign a customized template file which displays necessary content, you need to create a new .phtml file in the folder with the theme in the magento_theme module:

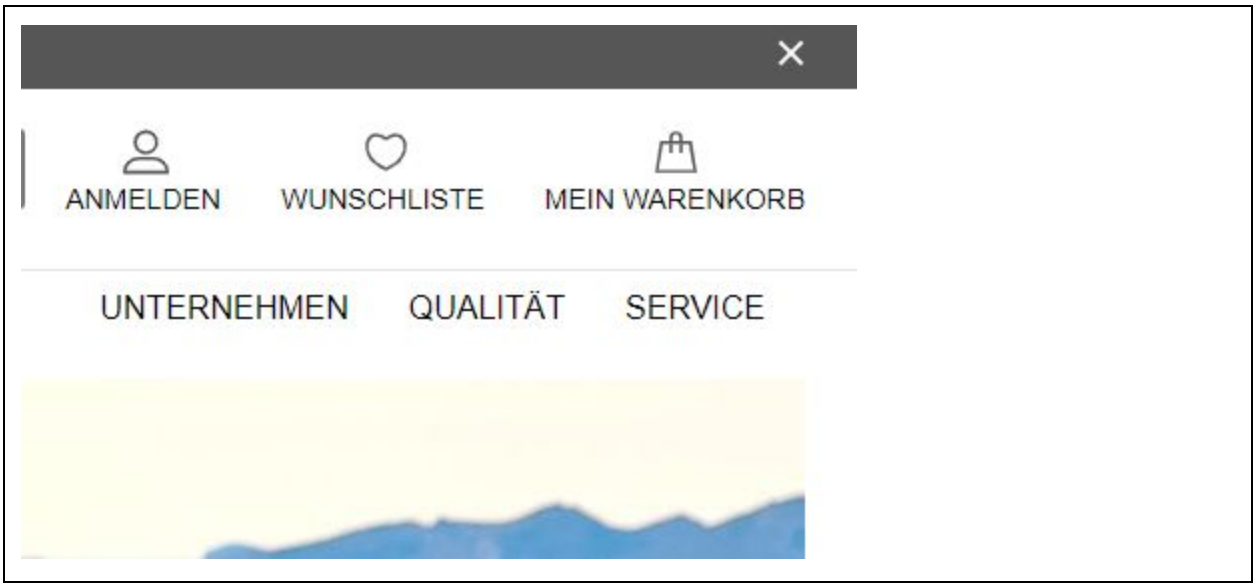
```
<Magento_folder>\app\design\frontend\<vendor_name>\<theme_name>\Magen
```

```
to_Theme\templates\header\clickme.phtml
```

With the default content (link with the icon and text):

```
<div class="clickme-wrap" >
 <a href="">
   <svg version="1.1" xmlns="http://www.w3.org/2000/svg" viewBox="0 0
129 129" xmlns:xlink="http://www.w3.org/1999/xlink"
enable-background="new 0 0 129 129">
     <g>
       <path
d="m121.6,40.1c-3.3-16.6-15.1-27.3-30.3-27.3-8.5,0-17.7,3.5-26.7,10.1
-9.1-6.8-18.3-10.3-26.9-10.3-15.2,0-27.1,10.8-30.3,27.6-4.8,24.9
10.6,58 55.7,76 0.5,0.2 1,0.3 1.5,0.3 0.5,0 1-0.1 1.5-0.3 45-18.4
60.3-51.4 55.5-76.1zm-57,67.9c-39.6-16.4-53.3-45-49.2-66.3 2.4-12.7
11.2-21 22.3-21 7.5,0 15.9,3.6 24.3,10.5 1.5,1.2 3.6,1.2 5.1,0
8.4-6.7 16.7-10.2 24.2-10.2 11.1,0 19.8,8.1 22.3,20.7
4.1,21.1-9.5,49.6-49,66.3z"/>
     </g>
   </svg>
   <span>click me</span>
 </a>
</div>
```

Then you need to assign the created block in the XML file where you want (for example, you want to add the link with the icon to the header close to the other links):



At the same time, we add the XML file.

```
<Magento_folder>\app\design\frontend\BelVG\<theme_name>\Magento_Theme
\layout\default.xml
```

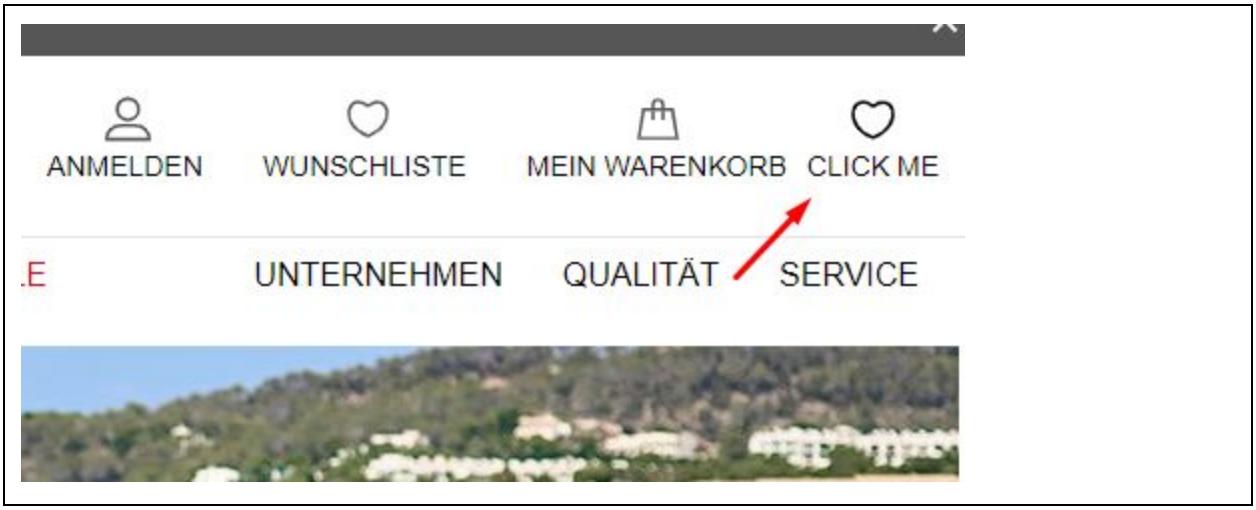
We add the following code to the file:

```
<referenceContainer name="header-wrapper">
   <block class="Magento\Framework\View\Element\Template"
name="clickme-header" before="-"
template="Magento_Theme::header/clickme.phtml"/>
</referenceContainer>
```

Clear the cache:

```
php bin/magento cache:flush
```

On the front, we get the following:



If you need to customize the standard template of the module and display it in the necessary place on the website, you should do the following:

Copy the necessary template file to the folder with the theme by relative path.
For example, the addtocart.phtml template is on the path:

```
<Magento_folder>/vendor/magento/module-catalog/view/frontend/templates/product/view/addtocart.phtml
```

Copy on the path:

```
<Magento_folder>\app\design\frontend\BelVG\<theme_name>\Magento_Catalog\templates\product\view\addtocart.phtml
```

Then, in the XML file, we set the template display in the necessary place :

```
<referenceContainer name="product.info.main">
 <block class="Magento\Catalog\Block\Product\View"
name="product.info.addtocart.second" as="addtocart-second"
```

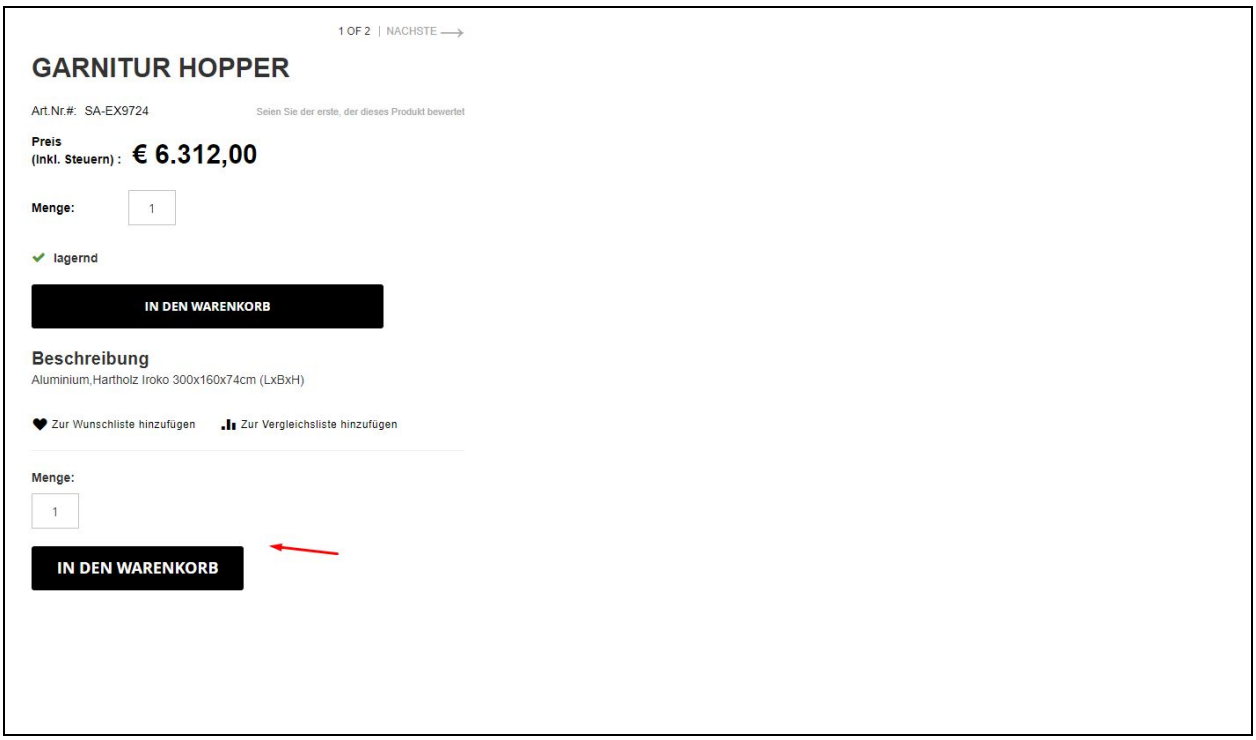
```
template="product/view/addtocart.phtml"/>
</referenceContainer>
```

You need to change **name** and **as** to the custom ones to make sure they are not the same as **name** and **as** of the original block.

Then, we clear the cache:

```
php bin/magento cache:flush
```

As a result, you see two identical blocks one of which is located in the place that we need. (For example, standard block output can be removed with the help of `<referenceBlock name=" " remove="true"/>`):



This way, the block can be displayed anywhere.

How does overriding a template affect upgradability?

When you override the template file with the help of layout XML, you actually violate the fallback path for the template. It does not allow the themes to modify the templates. While upgrading a module, it can happen that new functionality which appears in templates after the upgrade, is not available on the website as the template has been overridden.

What precautions can be taken to ease future upgrades when customizing templates?

If you are a backend developer: use plugin for the block (in this case, it will not allow to use view models), which will turn on the set template only when needed. The best way is to avoid overriding when possible.

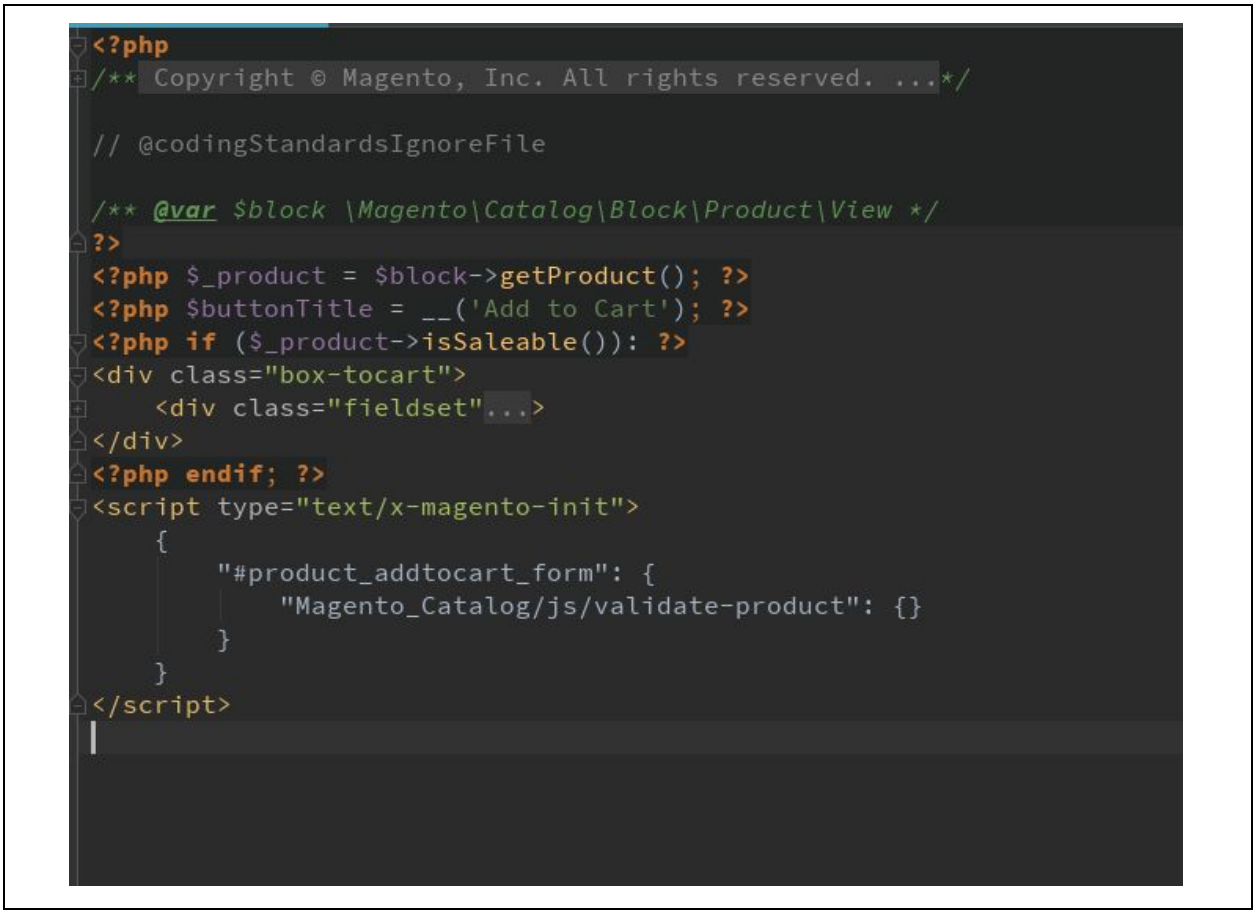
4.2 Override a native template file with a customized template file, using the design fallback

How can the design fallback be used to render customized templates?

If you need to use a customized template instead of the standard one, all you need is to copy the necessary phtml file to your theme with the right path. This way you override the standard theme. For example, in order to use customized template of the “Add to the Cart” button for the product:

Copy the file:

```
<Magento
folder>\vendor\magento\module-catalog\view\frontend\templates\product
\view\addtocart.phtml
```



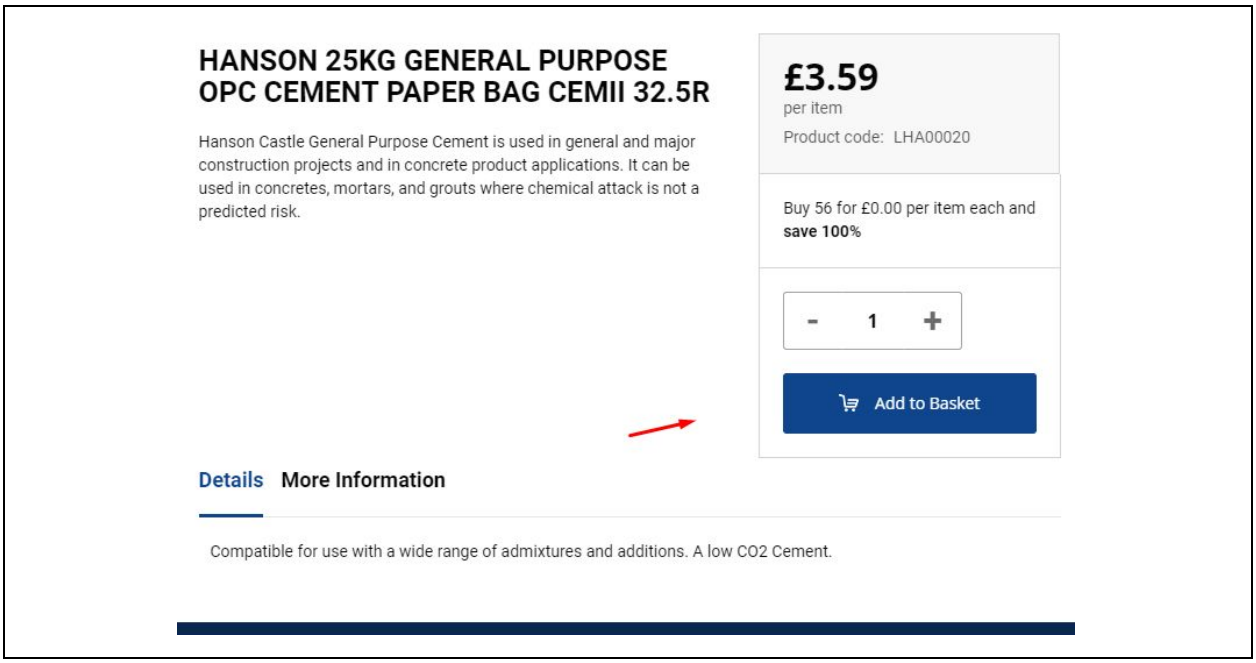
```php
<?php
/** Copyright © Magento, Inc. All rights reserved. ...*/

// @codingStandardsIgnoreFile

/** @var $block \Magento\Catalog\Block\Product\View */
?>
<?php $_product = $block->getProduct(); ?>
<?php $buttonTitle = __('Add to Cart'); ?>
<?php if ($_product->isSaleable()): ?>
<div class="box-tocart">
    <div class="fieldset"...>
</div>
<?php endif; ?>
<script type="text/x-magento-init">
    {
        "#product_addtocart_form": {
            "Magento_Catalog/js/validate-product": {}
        }
    }
</script>
```

Then, move it to the path:

```
<Magento
folder>\app\design\frontend\BelVG\<theme_name>\Magento_Catalog\templa
tes\product\view\addtocart.phtml
```

```
<?php
/** Copyright © 2013-2017 Magento, Inc. All rights reserved. ...*/

// @codingStandardsIgnoreFile

/** @var $block \Magento\Catalog\Block\Product\View */
?>
<?php $_product = $block->getProduct(); ?>
<?php $buttonTitle = __('Add to Basket'); ?>
<?php if ($_product->isSaleable()): ?>
<div class="box-tocart">
    <div class="fieldset"...>
</div>
<?php endif; ?>
<script type="text/x-magento-init">
    {
        "#product_addtocart_form": {
            "Magento_Catalog/product/view/validation": {
                "radioCheckboxClosest": ".nested"
            }
        }
    }
</script>
<?php if (!$block->isRedirectToCartEnabled()) : ?>
<script type="text/x-magento-init">
    {
        "#product_addtocart_form": {
            "catalogAddToCart": {}
        }
    }
</script>
<?php endif; ?>
```

On the front:



How does that influence upgradability?

When upgrading, the template functionality can be interrupted. The reason is that the template which has been overridden, could be upgraded together with the other system aspects which depend on the template. If the overridden template does not upgrade, everything can break down.

How can you determine which template a block renders?

In Magento 2, there are a few ways to find out which template renders.

You can search in the Magento files. Usually, the template files are in the XML files of the layout. In the XML file (for example, if you look for the **.box-tocart** block of the **addtocart.phtml** template, it is likely to be set in the <Magento_folder>\app\design\frontend\BelVG\<theme_name>\Magento_

Catalog\layout\catalog_product_view.xml file). You need to find the block which renders by the template:

```
<container name="product.info.form.content" as="product_info_form_content">
   <block class="Magento\Catalog\Block\Product\View"
name="product.info.addtocart" as="addtocart"
template="product/view/addtocart.phtml"/>
</container>
```

In the block, we see `template="product/view/addtocart.phtml"` path to the template. In some places, the template file is defined in the PHP class. Then, you need to search for the word $_template in the block's class.

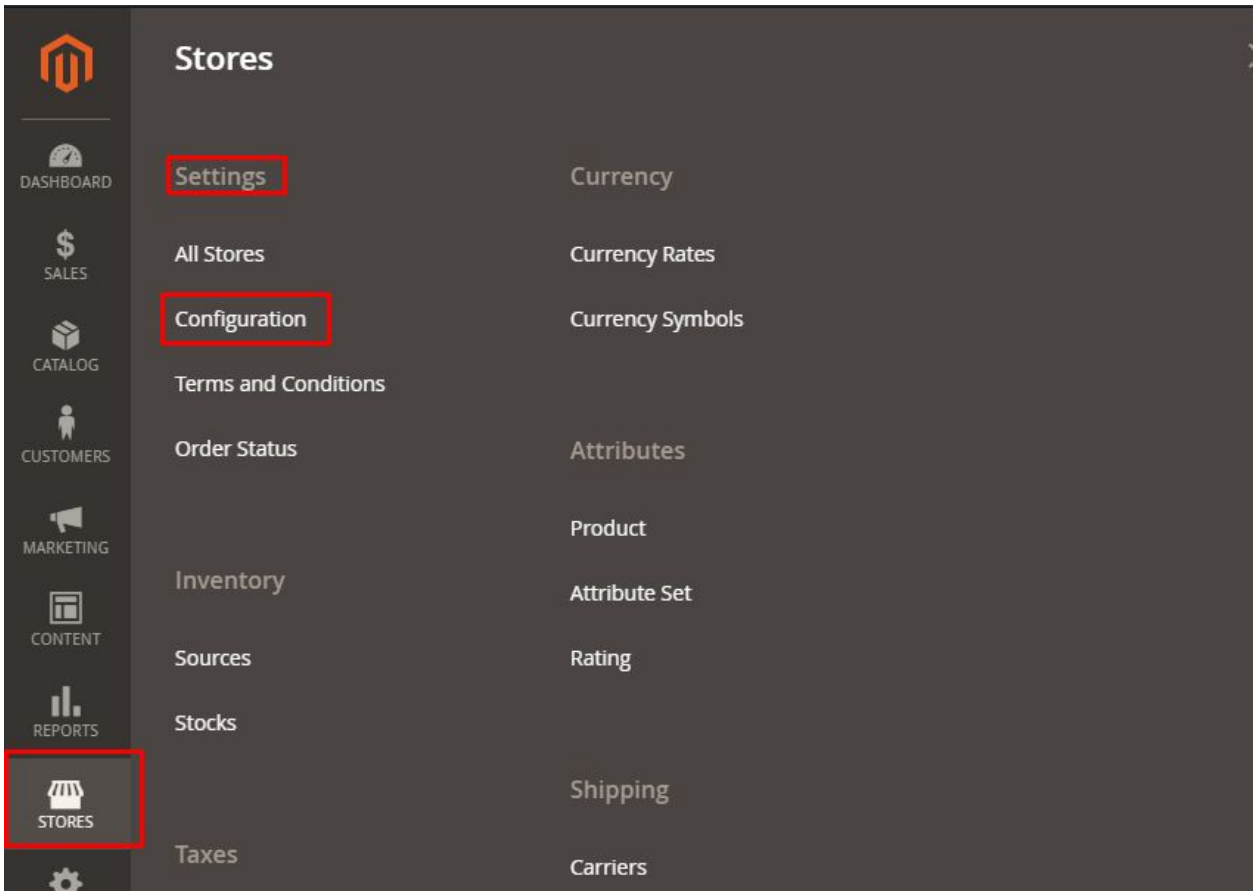
Defining the template

```
<Magento_folder>\app\code\Belvg\PaymentTrade\view\frontend\templates\
form\paymenttrade.phtml
```

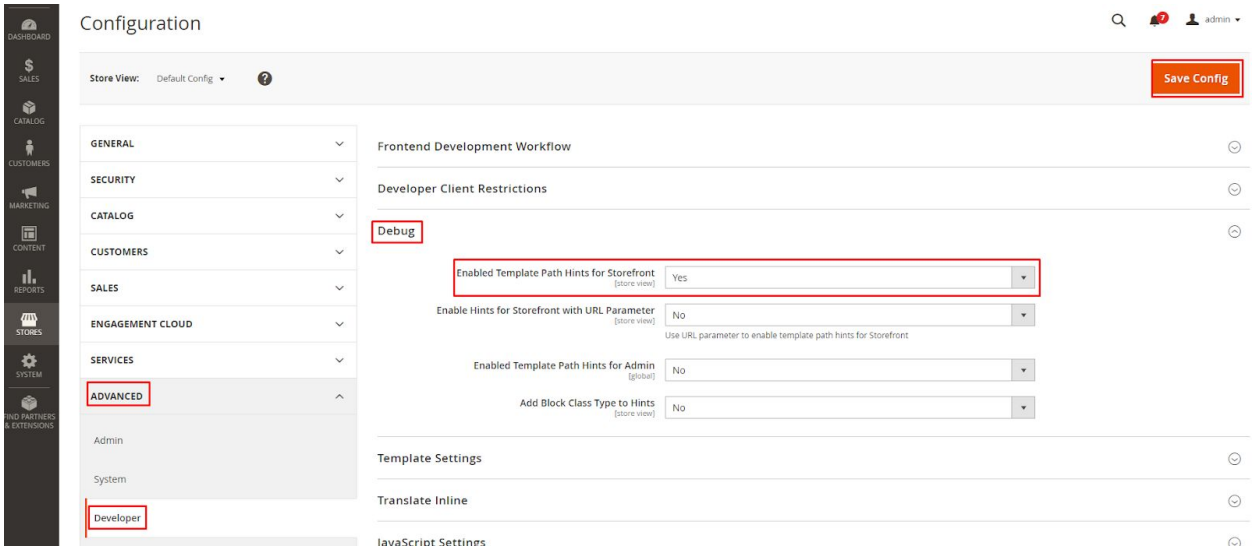
```
<?php

namespace Belvg\PaymentTrade\Block\Form;

/**
* Class PaymentTrade
* @package Belvg\PaymentTrade\Block\Form
*/
class PaymentTrade extends \Magento\Payment\Block\Form
{
   /**
    * Purchase order template
    *
    * @var string
    */
   protected $_template = 'Belvg_PaymentTrade::form/paymenttrade.phtml';
}
```

The easiest way to look at the path to the template is to enable the template path hints. In order to do this, proceed to:

STORES -> Settings -> Configuration



Here, on the tab ADVANCED -> Developer in the “Debug” section, you give the priority “Enabled Template Path Hints for Storefront” value “Yes”.



As a result, on the website, you can see the paths to the templates in red:

Another way to enable the path to the template is via the console command:

```
bin/magento dev:template-hints:enable
```

4.3 Describe conventions used in template files

What conventions are used in PHP templates?

- $this is no longer applicable to the block rendering. No matter which object you use in the template ($ block or $ this), it will always call the method from $block. If you take a look at `\Magento\Framework\View\TemplateEngine\Php::__call()` you will see:

```
public function __call($method, $args) {
 return call_user_func_array([$this->_currentBlock, $method], $args);
}
```

  Here you can see that `\Magento\Framework\View\TemplateEngine\Php` is just a proxy between the template and $block, so the use of $block is preferable because of the direct call.

- echo command in PHP should be written using a short tag in the Magento templates.

```
<?= $block->getModuleName() ?>
Istead of
<?php echo $block->getModuleName() ?>
```

- To make sure that your theme is displayed correctly with any language applicable to store view, check that unique lines of your theme are added in i18n. To get your new line added to the

vocabulary and translated, use the `__('<your_string>')` method when displaying the line in the .phtml template.

```
<?php echo __('Hello world') ?>
```

If the line includes a variable, then you should use the following syntax:

```
<?php echo __('Hello %1', $yourVariable) ?>
```

- If you need to use loop, use the foreach > endforeach constructions

```
<?php foreach ($answerData as $answerPosted):?>
      <h4>Hello</h4>
      <p>Answer given by: <?php echo $answerPosted['username'];
?></p>
      <p>Answer: <?php echo $answerPosted['answer']; ?></p>
<?php endforeach; ?>
```

Why aren't the common PHP loop and block constructs used?

Magento templates do not use constructions with { } because curly brackets are more difficult to identify in HTML.

```
<?php if($block->value): ?>
Hello
<?php elseif($block->asd): ?>
Your name is: <?= $block->name ?>
<?php else: ?>
You don't have a name.
<?php endif; ?>
```

Which common methods are available on the $block variable?

All the common methods from the block context are available in template.

Block context is a block class which is assigned to the template in layout XML file. It is equal to the **block type** in Magento 1. By default it is `\Magento\Framework\View\Element\Template` which is equal to `Mage_Core_Block_Template` in Magento 1.

This block context is assigned to the template as the $block variable during rendering.

Name the common methods available on the $block variable:

getRootDirectory() - Instantiates filesystem directory
getMediaDirectory() - Get media directory
getUrl($route = '', $params = []) - Generate url by route and parameters
getBaseUrl() - Get base url of the application
getChildBlock($alias) - Retrieve child block by name
getChildHtml($alias = '', $useCache = true) - Retrieve child block HTML
getChildChildHtml($alias, $childChildAlias = '', $useCache = true) - Render output of child child element
getChildData($alias, $key = '') - Get a value from child block by specified key
getBlockHtml($name) - Retrieve block html
formatDate($date = null, $format = \IntlDateFormatter::SHORT, $showTime = false, $timezone = null) - Retrieve formatting date

formatTime($time = null, $format = \IntlDateFormatter::SHORT, $showDate = false) - Retrieve formatting time
getModuleName() - Retrieve module name of block
escapeHtml($data, $allowedTags = null) - Escape HTML entities
escapeJs($string) - Escape string for the JavaScript context
escapeHtmlAttr($string, $escapingSingleQuote = true) - Escape a string for the HTML attribute context
escapeCss($string) - Escape string for the CSS context
stripTags($data, $allowableTags = null, $allowHtmlEntries = false) - Wrapper for standard strip_tags() function with extra functionality for html entities
escapeUrl($string) - Escape URL
getVar($name, $module = null) - Get variable value from view configuration

How can a child block be rendered?

Use the \Magento\Framework\View\Element\AbstractBlock::getChildHtml() method:

```
public function getChildHtml($alias = '', $useCache = true)
{
   $layout = $block->getLayout();
   if (!$layout) {
       return '';
   }
   $name = $this->getNameInLayout();
   $out = '';
   if ($alias) {
       $childName = $layout->getChildName($name, $alias);
       if ($childName) {
           $out = $layout->renderElement($childName, $useCache);
       }
```

```
    } else {
        foreach ($layout->getChildNames($name) as $child) {
            $out .= $layout->renderElement($child, $useCache);
        }
    }
    return $out;
}
```

Here is the example of use:

```
public function getSaveButtonHtml()
{
    return $block->getChildHtml('save_button');
}
```

Or in the template:

```
$block->getChildHtml('save_button');
```

How can all child blocks be rendered?

The same way as you would call a specific child block but without using the **name** variable:

```
$block->getChildHtml();
```

How can a group of child blocks be rendered?

The block group methods are located in vendor/magento/framework/View/LayoutInterface.php

The main group example in Magento core is located on the product detail page's tabs (https://github.com/magento/magento2/blob/2.2-develop/app/code/Magento/Catalog/view/frontend/templates/product/view/details.phtml)

```
<?php
/**
 * Copyright (c) Magento, Inc. All rights reserved.
 * See COPYING.txt for license details.
 */
// @codingStandardsIgnoreFile
?>
<?php if ($detailedInfoGroup =
$block->getGroupChildNames('detailed_info', 'getChildHtml')):?>
<div class="product info detailed">
    <?php $layout = $block->getLayout(); ?>
    <div class="product data items"
data-mage-init='{"tabs":{"openedState":"active"}}'>
        <?php foreach ($detailedInfoGroup as $name):?>
        <?php
                    $html = $layout->renderElement($name);
                    if (!trim($html)) {
                        continue;
                    }
                    $alias = $layout->getElementAlias($name);
                    $label = $block->getChildData($alias, 'title');
                ?>
        <div class="data item title" data-role="collapsible"
id="tab-label-<?= /* @escapeNotVerified */ $alias ?>">
            <a class="data switch" tabindex="-1"
data-toggle="trigger" href="#<?= /* @escapeNotVerified */ $alias ?>"
id="tab-label-<?= /* @escapeNotVerified */ $alias ?>-title">
                <?= /* @escapeNotVerified */ $label ?>
            </a>
        </div>
        <div class="data item content" aria-labelledby="tab-label-<?=
/* @escapeNotVerified */ $alias ?>-title" id="<?= /*
```

```
@escapeNotVerified */ $alias ?>" data-role="content">
            <?= /* @escapeNotVerified */ $html ?>
        </div>
        <?php endforeach;?>
    </div>
</div>
<?php endif; ?>
```

Their rendering includes getting all their names with the use of getGroupChildNames and then rendering of every block by name in the loop.

4.4 Render values of arguments set via layout XML

How can values set on a block in layout XML be accessed and rendered in a template?

Let's take a look at the example. For instance, you want to add a class to the product form block using XML.

In order to do it, in the module file (for example, /app/design/frontend/{vendor_name}/{theme_name}/Magento_Catalog/layout/catalog_product_view.xml), you need do add the following:

```
<referenceBlock name="product.info"
template="product/view/form.phtml" after="alert.urls">
 <arguments>
   <argument name="custom_class"
xsi:type="string">custom-class</argument>
```

```
 </arguments>
</referenceBlock>
```

In product.info block, we set an argument with the name "custom_class" (or any other), assign a type (type="string"), then the value - “custom-class”

In order to use this value in the template file you should use one of the following methods:

```
app/design/frontend/BelVG/{theme_name}/Magento_Catalog/templates/product/view/form.phtml
```

```
$cssClass = $block->hasCustomClass() ? ' ' . $block->getCustomClass() : '';
```

or

```
$cssClass = $block- >getData('ustom_class');
```

Let’s add the variables with different methods of obtaining the value and render them in the class=”” attribute.

```
<?php $cssClass = $block->hasCustomClass() ? ' ' . $block->getCustomClass() : ''; ?>

<?php $cssClass2 =  $block->getData('custom_class'); ?>

<div class="product-add-form <?php echo $cssClass; ?> <?php echo $cssClass2; ?>">
```

As a result, it looks like that:

4.5 Demonstrate ability to escape content rendered and template files

How can dynamic values be rendered securely in HTML, HTML attributes, JavaScript, and in URLs?

To render the values securely, you need to use the block's built-in methods:

For JavaScript: `$block->escapeJs('value');`

For HTML: `$block->escapeHtml('value', $allowedTags);`

HTML attributes: `$block->escapeHtmlAttr('value', $escapeSingleQuote);`

URLs: `$block->escapeUrl($url);`

All these methods are described in **AbstractBlock** (Magento\Framework\Escaper)

escapeJs()

```
/**
* Escape string for the JavaScript context *
* @param string $string
* @return string
 */
 public function escapeJs($string)
```

The unicode symbols are encrypted, for example,⌛ turns into U+231B. This method is used for JS string literal. For inline JavaScript,
(for example, onclick) you should use escapeHtmlAttr()

escapeHtml()

```
/**
* Escape string for HTML context. allowedTags will not be escaped,
except the following: script, img, embed,
* iframe, video, source, object, audio
*
* @param string|array $data
* @param array|null $allowedTags
* @return string|array
*/
 public function escapeHtml($data, $allowedTags = null)
```

It should be your method of escaping data by default for any output. The main point is that the output of all the methods that do not include "Html" should be escaped.

escapeHtmlAttr()

```
/**
```

```
* Escape a string for the HTML attribute context
*
* @param string $string
* @param boolean $escapeSingleQuote
* @return string
*/
public function escapeHtmlAttr($string, $escapeSingleQuote = true)
```

Use it for escaping the output in the HTML attribute, for example:

```
title="<?php echo $block->escapeHtmlAttr($title) ?>"
```

By default, it also escapes single quotes:

```
onclick="console.log('<?php echo $block->escapeHtmlAttr($message)
?>')"
```

Set the value **false** on the second variable if you do not need it.

escapeUrl()

```
/**
* Escape URL
*
* @param string $string
* @return string
*/
public function escapeUrl($string)
```

This method is used for URL output. It applies HTML literal by default and additionally deletes `javascript:`, `vbscript:` and `data:`. If you want to deny URLs similar to these among the links provided by a user, you can use the method.

In the releases before Magento 2.1, this function was not enabled and you had to use escapeXssInUrl().

Section 5: Static Asset Deployment

5.1 Describe the static asset deployment process for different file types

What commands must be executed to deploy static file types?

Static files are the ones that can be cached on the website (caching increases page load speed). CSS, fonts, images and javaScript used by the theme are static files. These cached files are stored in "*/pub/static*" and "*/var/view_preprocessed/pub/static*" folders. They get in there after the deployment.

In order to execute the deployment process manually, you need to execute the following command (make sure that your user has enough rights for it and you are in the site's core).

php bin/magento setup:static-content:deploy

It was an example of a standard command without additional variables. As a result, all the static files of all the themes, areas and so on will be deployed. There are many different variables for this command that correct the deployment process.The line with variables will look like this:

php bin/magento setup:static-content:deploy [<languages>] [--theme["<theme>"]] [--exclude-theme["<theme>"]] [--language["<language>"]] [--exclude-language["<language>"]] [--area["<area>"]] [--exclude-area["<area>"]] [--jobs["<number>"]]

[--no-javascript] [--no-css] [--no-less] [--no-images] [--no-fonts] [--no-html] [--no-misc] [--no-html-minify] [--dry-run] [--force] [--verbose]

Where

<languages> is a space-separated list of language codes. Static content deploys only for the specified languages. By default, it deploys only for en_US (if variable is not specified). For example, *en_US fr_FR*.

--theme ["<theme>"] is a list of themes for which static content deploys (if variable is not specified, static content deploys for all themes). For example, *--theme Magento/blank --theme Magento/luma*.

--exclude-theme ["<theme>"] is a list of themes for which static content does not deploy. For example, *--exclude-theme Magento/blank --exclude-theme Magento/luma*.

--language["<language>"] is a list of languages for which static content deploys (if variable is not specified, the static content will deploy for all the languages). For example, *--language en_US --language es_ES*. If you specify this variable and the <languages> variable (described above), then the <languages> variable will have the priority. For shortening, instead of --language, you can use **-l** (for example: *-l es_ES*)

--exclude-language["<language>"] is a list of languages for which static content does not deploy. For example, *--exclude-language en_US --exclude-language es_ES*.

--area["<area>"] is a list of areas for which static content deploys, (if variable is not specified, the static content will deploy for all the areas). For example, *--area adminhtml*. For shortening, instead of --area, you can use **-a** (for example: *-a adminhtml*)

--exclude-area[="<area>"] is a list of areas for which static content does not deploy. For example, *--exclude-area adminhtml*.

--jobs[="<number>"] is a variable that allows you to use a certain number of jobs during parallel processing (by default, it is 4). For example, *--jobs 1*. For shortening, instead of --jobs, you can use **-j** (for example: *-j 1*).

--no-javascript is a variable that prevents JavaScript files deployment.

--no-css is a variable that prevents CSS files deployment.

--no-less is a variable that prevents LESS files deployment.

--no-images is a variable that prevents images deployment.

--no-fonts is a variable that prevents font files deployment.

--no-html is a variable that prevents HTML files deployment.
--no-misc is a variable that does not allow other file types (for example, .md, .jbf, .csv, .json, .txt, .htc, .swf) to deploy.

--no-html-minify is a variable that does not allow the HTML files to minify.

--dry-run is a variable that allows you to look at the output files without any changes.

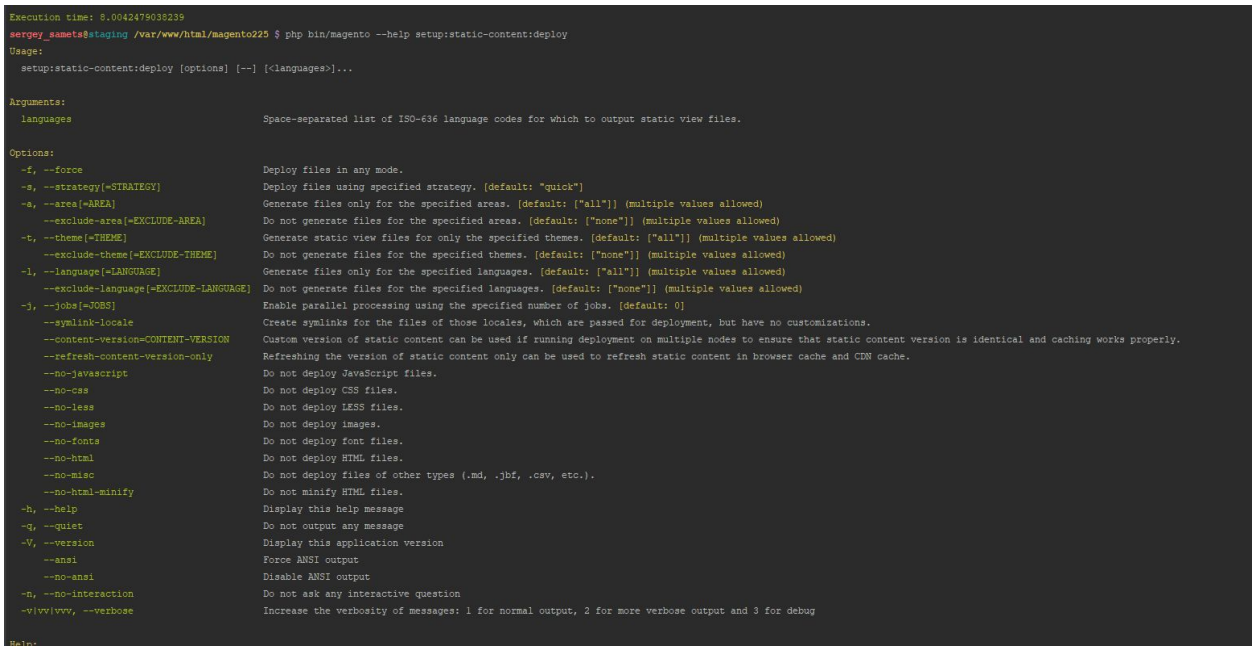
--force is a variable that deploys files at any mode. For shortening, instead of --force, you can use **-f**

--verbose is a variable that changes the amount of information that outputs while deploying. Usually, the short form of this variable is used

where **-v** is for minimum information about deployment (default), **-vv** is for more information about deployment, **-vvv** is used for debugging.

You can see the full list of variables if you execute this command:

php bin/magento --help setup:static-content:deploy



All these variables are not required but using them, you will be able to configure the deployment process according to the tasks that you need to manage.

We also would like to remind you of clearing the static cache files before deployment. There are two ways how this can be done:

1. In the admin panel, you should proceed to *System > Tools > Cache Management* and choose **Flush Static Files Cache**.
2. Manually, we delete what is inside the folders "*/pub/static*" and "*/var/view_preprocessed/pub/static*" (do not delete the folders themselves and the .htaccess files).

3. On the command line, using the **rm -rf pub/static/frontend/*** command

You can get more details about it here: https://devdocs.magento.com/guides/v2.1/frontend-dev-guide/cache_for_frontdevs.html#clean_static_cache)

What are common mistakes during the process?

The following mistakes are common during file deployment:

- Sometimes developers forget about deployment for some of the languages. Default deployment command deploys only the files for the current language version (by default, en_US) and if there are several language versions, they will not be deployed. In order to deploy the content of other language versions, you should use the <languages> variable or --language[="<language>"] described above.

- Sometimes developers try to carry out deployment not having enough rights to record files. Before the deployment process, it is recommended to execute the following commands:

  cd <your Magento install dir>
  find . -type f -exec chmod 644 {} \;
  find . -type d -exec chmod 755 {} \;
  find ./var -type d -exec chmod 777 {} \;
  find ./pub/media -type d -exec chmod 777 {} \;
  find ./pub/static -type d -exec chmod 777 {} \;
  chmod 777 ./app/etc
  chmod 644 ./app/etc/*.xml

chown -R :<web server group> .
chmod u+x bin/magento

- Sometimes developers forget to clear the static cache files. It can lead to the issue when the old unnecessary static cache files remain after the deployment and interfere with the correct website view.
- Sometimes developers try to manage deployment not from the website core.

- Finally, there may also be ordinary mistakes in a team's syntax.

5.2 Describe the effect of deploy modes on frontend development

What are the differences between development and production mode in regard to frontend development?

There are 3 website operating modes: Default, Developer and Production.

Default mode is a default Magento mode when no other mode is selected. It allows you to deploy Magento on one server without changing any settings. At the same time, this mode does not provide such a level of website optimization as production mode. That is why later developers usually switch the default mode to developer or production mode depending on a task.

These are the special features of the default mode:

- The link is generated for every requested static view file in pub/static catalog;
- The static files are generated dynamically and because of it, website loading will take much more time than in the production mode;
- The user cannot see the mistakes (on frontend) but these mistakes are recorded to server reports.

To make website configuration more convenient, you should turn on **developer mode**, and to maximize production optimization, use **production mode**. Below you can find the information on how to change developer mode.

Developer mode is the mode that should be used when you develop new functionality or customize the existing one as it allows you to get feedback from the system much faster than in other modes.

These are the special features of the developer mode:

- Static view files are not caching. They are recorded to the pub/static folder every time when they are requested
- Static files are generated dynamically and because of it, website loading will take much more time than in the production mode;
- Uncaught exceptions and errors are visible in browser;
- Exceptions and mistakes are recorded in var/log and /var/reports _;
- Exceptions are generated in the error handler rather than being logged
- Exceptions are generated when an event subscriber cannot be invoked.

Production mode is used when the website has already been deployed to a production server. This website will be available to all the users and in this mode, the website has maximum performance and load speed. The

production mode should be switched on only when website development is over and you are ready to go. Before you switch it on, you should manage all the necessary configurations on the production server where the website is hosted (optimize server environment).

After the website has been deployed, you should run the static view files deployment tool to generate static files which will be recorded to pub/static folder (it increases the performance as all the necessary static files are immediately available instead of generating them when needed as in other modes).

These are the special features of the production mode:

- Static view files are served from cache. They do not have formed links in the pub/static catalog, the links are created along the way. New or updated files are not recorded to the file system.
- The errors are not shown to a user (on frontend), but they are recorded to the report files on the server.

It is clear from the description of modes that the main difference between development and production modes is that the first one is used while developing and configuring the website, and the second one is when the development is over and the highest website performance is required.

There is also **Maintenance mode** which is used to temporar close the website from users. When users visit the website, they will be redirected to the “Service Temporarily Unavailable” page (you can find out how to create a custom maintenance page here - https://devdocs.magento.com/guides/v2.2/comp-mgr/trouble/cman/maint-mode.htm). This mode is useful when you configure some website features which can affect website’s functionality or design. Then, it is better

to make sure that the user will not come to the website and encounter broken design or incorrect functionality.

To find out which mode is used at the moment, execute this command:

bin/magento deploy:mode:show

In order to enable the **production mode**, execute this command:

bin/magento deploy:mode:set production

In order to enable the **developer mode**, execute this command:

bin/magento deploy:mode:set developer

In order to enable the **maintenance mode**, execute this command:

bin/magento maintenance:enable

In order to disable the **maintenance mod**, execute this command:

bin/magento maintenance:disable

You get more information about changing the mode here: https://belvg.com/blog/how-to-switch-between-deploy-modes-in-magento-2.html

5.3 Demonstrate your understanding of LESS > CSS deployment and its restrictions in development

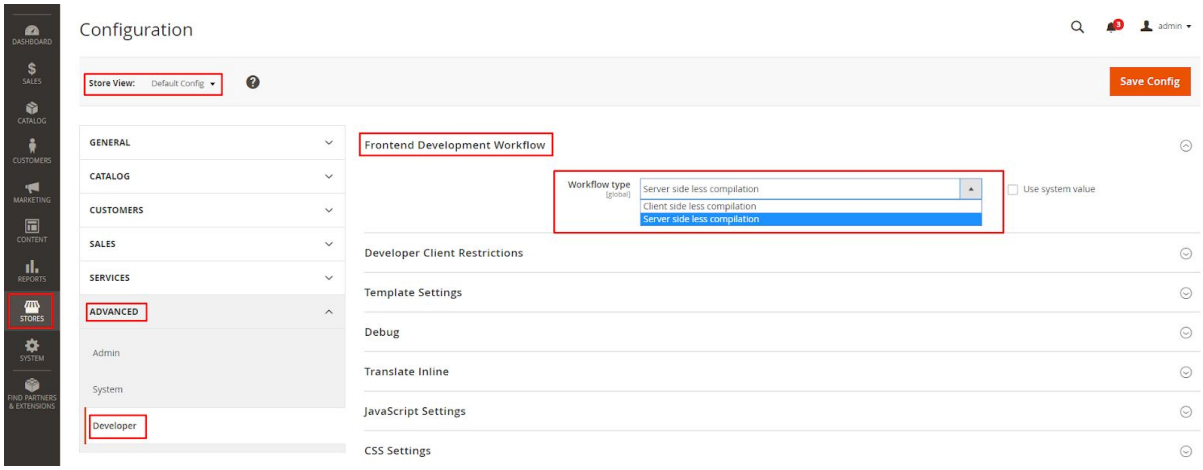
Which LESS compilation options are available in Magento?

In Magento, there are two LESS compilation options in CSS:

- **server-side LESS compilation** (default option)
- **client-side LESS compilation**

You can change the compilation option in the following way (It can be done only if the website is not in the Production mode. You can find how to change website modes in the 5.2 section):

- Click on "*Stores > Configuration > ADVANCED > Developer"* in the admin panel.
- In the dropdown menu "**Store View**" choose "**Default Config**".
- In the "*Frontend Development Workflow*" tab in the "**Workflow type**" field choose the needed mode.
- Save the changes pushing on the "**Save Config**" button.



How are they different?

To understand the differences between compilation options, we need to consider both of them in detail.

Server-side LESS compilation
This compilation is done on the server side with **Less PHP library** (more details about the library are here https://github.com/oyejorge/less.php). As it was mentioned before, only this compilation option is available for the Production mode.

During **server-side Less compilation**, there is a compilation of the files which the system can't find in cache and static files.
As a result, changing something in the .less file of your theme, we need to clear the folders "*pub/static/frontend/<our_package>/<our_theme>/<local>*", "*var/cache*" and "*var/view_preprocessed*" (more information about static files you can find here https://devdocs.magento.com/guides/v2.0/frontend-dev-guide/cache_for_frontdevs.html#clean_static_cache). After that, we need either to reload the page or make a deploy (about which it was written above). If we reload the

page, only the files related to this page are recompiled. Making a deploy, we recompile all the sites' styles (it's easier to track errors during a deploy, if they occur). The changes in the .less file are applied to the site page only after that.

Use Grunt to simplify and speed up the process. In short, Grunt compiles only those files which have changed (there is no need to clean the folders yourself and make a deploy). More information about Less compilation with Grunt you can find here (https://devdocs.magento.com/guides/v2.2/frontend-dev-guide/css-topics/css_debug.html).

Client-side LESS compilation

This compilation is done on the client's side - in browser - using **native less.js library** (more information about the library you can find here http://lesscss.org/usage/#using-less-in-the-browser). This compilation option isn't available in the Production mode.

Each time the page is refreshed, there starts a file compilation process in a browser whether you've changed .less theme files or not.
Sometimes it's necessary to clear the folder "*pub/static/frontend/<Vendor>/<theme>/<locale>*" (more information about this and a detailed analysis of the both compilation types processes you can find here https://devdocs.magento.com/guides/v2.2/frontend-dev-guide/css-topics/css-preprocess.html#css_exception). However, it's unnecessary in most cases. You only need to reload the page to start a client-side compilation.

Thus, the main differences between **server-side LESS compilation** and **client-side LESS compilation** are as follows:

- **server-side LESS compilation** is done on the server side, while **client-side LESS compilation** on a client side (in a browser).
- **client-side LESS compilation** is run every time the page is reloaded, while **server-side LESS compilation** only after cleaning the "*pub/static/frontend/<our_package>/<our_theme>/<local>*", "*var/cache*", "*var/view_preprocessed*" folders, the page refreshment or deploy.
- **client-side Less compilation** is not available in the Product mode, while **server-side LESS compilation** is available.

How do they influence the developer workflow during theming?

In theory, **client-side LESS compilation** should speed up a theme development process since files will be compiled every time we refresh the page. If we make changes in the .less styles files of our theme, they will be immediately seen on the client side when the page is refreshed without static files pre-cleaning and so on. In practice, each page refreshment takes a lot of time. The more complex the site is, the more styles' files it has, the more weighty JavaScript files are loaded on the page, it takes more time to load the page and, as a result, each page might be loading several minutes. You can't deny that it's a doubtful way to speed up a theme development.

On the other hand, **server-side LESS compilation** requires cleaning the folders of static files when changing .less files before loading each page (it's better to make a deploy after cleaning folders with static files). As a result, we make additional actions before refreshing each page which doesn't simplify the theme development process.

Here it’s better to use Grunt to track the source .less files and recompile the output when these files are changing (more information about how to use Grunt you can find here https://devdocs.magento.com/guides/v2.2/frontend-dev-guide/css-topics/css_debug.html)

Grunt will keep tracking changes in the source files and run compilation, if necessary. So, we don’t need to do unnecessary actions before each page reload. In the end, this helps to simplify and speed up the development process.

Section 6: Customize and Create JavaScript

6.1 Include custom JavaScript on pages

What options exist to include custom JavaScript on a page?

Magento 2 allows connecting custom scripts for the entire site, certain pages, or even certain blocks or parts of the page.
It’s possible through connecting scripts in the page head, a special engine framework or RequireJS (the most preferred one).

Consider the first option - connecting scripts in the page head. This option is considered the least desirable, since it is impossible to redefine classes in the script and any interaction with a user is minimized. The option is applicable if you connect well known libraries via cdn or if you use a third-party aggregator of your scripts.
The connection process is as follows:
<script src = ”absolute or relative path to the script” (also use src_type = "url" if the path is absolute)> </script> in your xml
or
<script type = "text / javascript"> script content </script> inline in phtml.
To partially smooth out the cons of this option, try to use the attribute deferred = "true" or at least async = "true" in all cases when it is possible (for scripts by links, it doesn’t work with inline scripts).
Instead of <script src = "js / script.js" /> you can use <link src = "js / script.js" />

The second option - Magento framework.

The script can be connected in two ways:
Using the following attribute: data-mage-init = '{"VendorName_Module / js / script": {"configuration-value": true}}'. This attribute requests a script with 2 parameters - the element that will work out, and its parameters (animation speed, number of elements, etc.)
The final script:

```
app/code/VendorName/Module/view/frontend/web/js/script.js
define([], function() {
return function(element, config) {
/* config: {
configuration-value: true
} */
};
});
```

Or using script tag

```
<script type="script/x-magento-init">
{".element-selector":
{"VendorName_Module/js/script":
{"configuration-value": true}
}
}
</script>
```

where .element-selector is a selector for which the module will be applied. If several elements fall under the selector, it will be applied to each of them.

There is not the most correct, but definitely an easy way to connect a script that depends on a third-party library:

```
<script type="text/javascript">
require([
"VendorName_Module/js/script"    //indicate dependence from the
library
], function(loader) {
/* your script */
});
</script>
```

Through RequireJS:
This option allows connecting the script at any page, it works faster than the first two, because it requires less code to initialize.

```
var config = {
deps: ['VendorName_Module/js/script']
};
```

What are the advantages and disadvantages of inline JavaScript?

Pros and cons of inline js are the following:

+ Easy to connect
+ Doesn’t require additional request per page
- The code is not delayed. It is executed immediately, so it is not recommended to place inline-bulky and time-consuming code.
- Such code is difficult to reuse - on other pages it will have to be reinserted.
- Such code is difficult to find in templates, which makes it difficult to maintain.
- It is not cached by the browser as it is a part of the page

- Such code doesn't take to Content-Security-Policy, which makes it dangerous in terms of cross-site scripting.

How can JavaScript be loaded asynchronously on a page?

By the define method.
Require is executed immediately, unlike define, which will expect a call from other modules. For instance:

```
require([
'VendorName_Module/js/script'
], function(FuntionName) {
/* ... */
});
```

Instead, we can delay execution and with the help of modules, for example, Defer for Magento by BelVG will do fine with your scripts. You can download it here:

https://store.belvg.com/defer-js-for-magento-2-0.html

How can JavaScript on a page be configured using block arguments in layout XML? How can it be done directly in a .phtml template?

There is a getJsLayout option in AbstractBlock that returns an array from jsLayout parameters.
In frontend <?= $block->getJsLayout();?> it returns a json format string, built from jsLayout.

So, using <arguments/>, you can create your json object:

```
<referenceBlock name="BlockName">
      <arguments>
            <argument name="jsLayout" xsi:type="array">
                  <item name="components" xsi:type="array">
                  <item name="script" xsi:type="array">
                  <item name="component"
xsi:type="string">VendorName_Module/js/script</item>
                  </item></item>
            </argument>
      </arguments>
</referenceBlock>
```

6.2 Demonstrate understanding of using jQuery

Demonstrate understanding of jQuery and jQuery UI widgets.

To simplify the development of elements like dropdown lists, accordions, buttons, date pickers, etc. Magento uses the jquery and jquery ui libraries.

Comparing jquery with clean js, jquery code is shorter and more simple. But it influence a browser performance.
jquery ui is a library extension that provides a set of ready-to-use widgets in projects. The full list can be found here.

Demonstrate understanding of how to use Magento core jQuery widgets.

Magento uses jquery ui widgets to save development time - the most frequently used elements in projects are already available out of the box.

In almost any project, it's necessary to create a FAQ page or tabs with product descriptions. Date picking, slider are some of the most commonly used widgets that you will prefer in your theme.

How can jQuery UI Widget methods be instantiated?

Usually the requirement process looks as follows (for example, an accordion):

```
$("#element").accordion();
```

You can make a requirement in many different ways.

- In template using script

```
<script>
    require([
        'jquery',
        'accordion'], function ($) {
        $("#element").accordion();
    });
</script>
```

- In template using data-mage-init

```
<div id="element" data-mage-init='{"collapsible":{"openedState":
"active", "collapsible": true, "active": true, "collateral": {
"openedState": "filter-active", "element": "body" } }}'>
    <div data-role="collapsible">
        <div data-role="trigger">
            <span>Title 1</span>
        </div>
    </div>
    <div data-role="content">Content 1</div>
....
```

- In the script file using selector or requiring a function

```
require([
        'jquery',
        'accordion'], function ($) {
        $("#element").accordion();
    });
```

- Using <script type="text/x-magento-init" />

```
<script type="text/x-magento-init">
{
     "#element": { // your selector that uses widget
          "accordion": // parameters are displayed here, for example
<?php echo $block->getNavigationAccordionConfig(); ?>

}
}
<script>
```

How can you call jQuery UI Widget methods?

It’s done as follows:

```
$('#element').accordion("someAction");
```

We require the widget and the function we want through the method.

How can you add new methods to a jQuery UI Widget?

We either finish writing the widget (using mixins) or create our own custom component. The first option is preferable when there is already the code that we would like to expand and supplement. Otherwise, working from scratch, it is better to use the second one.
Let’s consider the first way:

app/code/Vendor/Module/view/frontend/requirejs-config.js

```
var config = {
   "config": {
      "mixins": {
         "mage/tabs": {
            'Vendor_Module/js/accordion-mixin': true
             }
          }
       }
};
```

app/code/Vendor/Module/view/frontend/web/js/accordion-mixin.js

```
define([
   'jquery'
], function($) {
```

```
      return function (original) {
         $.widget('mage.accordion', original, {
            activate: function() {
// you code is here
           return this._super();
           }
         }
      );
   return $['mage']['accordion'];
   }
});
```

How can a jQuery UI Widget method be wrapped with custom logic?

The second way - a custom component introduction:

app/code/Vendor/Module/view/frontend/web/js/custom-accordion.js

```
define([
'jquery',
'jquery/ui',
'mage/accordion'
], function($) {
      $.widget('vendor.customAccordion', $.mage.accordion,{
         someAction: function(element) {
            // your code is here
         }
      });
});
```

And add to the template:

app/code/Vendor/Module/view/frontend/templates/test.phtml

```
<div id="element"></div>
<script>
require([
'jquery',
'Vendor_Module/js/custom-accordion'], function ($) {
   $("#element").customAccordion();
   $("#element").customAccordion("someAction");
   // Require a widget function as described above
});
</script>
```

6.3 Demonstrate understanding of requireJS

How do you load a file with require.js?

Magento 2 introduced a number of innovations in JavaScript usage. One of them is requireJS. It is used to load script files, helps add dependencies, and generally simplifies working with script files.

To add Js file to your Magento 2 custom module, you need to add requirejs-config.js file to your module along the path app/code/<Company>/<modulename>/view/frontend

```
var config = {
    map: {
        '*': {
            belvgmodule: 'Company_modulename/js/belvgrequirejs',
        }
```

```
    }
};
```

The **config** variable contains properties with the **map** and **deps** keys. For example, in this case the map property contains an object with the key that is alias to file and value that is path to file.

“ * ” is used to determine that you are using a new module in require js. If you want to add a file to the existing module, you should indicate the name of the module you need instead of **“*”**.

No file extension is required; the system accepts js by default.

The combined configuration will be loaded onto the page immediately after require.js and will be used by the require () and define () functions.

How do you define a require.js module?

For example, there is helper/belvg module. To use it, you need to add the following code to the js file (main.js as an example):

```
requirejs(['helper/belvg'], function(helper_belvg) {
    var message = helper_belvg.getMessage();
    alert(message);
});
```

We specify the modules with the array we want to load, and pass this array as the first argument to the requirejs function call. RequireJS then passes the object that exports the helper/belvg module to our main function as the first helper_belvg parameter.

Then we need to define a helper / belvg module. To define a module, turn the module name into a file path, and add the following contents (scripts / helper / belvg.js):

```
define([], function(){
    var x = {};
    x.getMessage = function()
    {
        return 'Some message';
    }
    return x;
});
```

How are require.js module dependencies specified?

To build dependency on a third-party plugin, specify [shim] in the following configuration files:

requirejs-config.js

```
var config = {
     "shim": {
     "3-rd-party-plugin": ["jquery"]
     }
 };
```

<third-party-plugin>.js

```
!(function($){
     // plugin code
     // where $ == jQuery
 })(jQuery);
```

How are module aliases configured in requirejs-config.js?

In our example, the RequireJS module name was tied to the source location of this module on disk. In other words, the helper/belvg module will always be on the helper / belvg.js path.

RequireJS allows changing this through configuration. For example, if you want your helper/belvg module to be named hello, you would run the following configuration code somewhere before running your program:

```
require.config({
    paths: {
        "hello": "helper/belvg"
    },
});
```

The paths configuration key the place where we can rename modules.The key of the path object is the name that we want to give (hello), and the meaning is the actual mode name (helper/belvg).
Therefore, we can work with the module as follows:

```
requirejs(['hello'], function(hello) {
    alert("Some message");
});
```

How do you regenerate the compiled requirejs-config.js file after changes?

Delete static content and make:

```
php bin/magento cache:clean
```

To be totally sure.

How do you debug which file a requireJS alias refers to?

To track which file corresponds to the renamed module, you need to open the requirejs-config.js file:

```
var config = {
    map: {
        '*': {
            addToCart: 'Magento_Module/js/module'
        }
    }
};
```

In the file, you can find the key map with aliases module parameters and its paths.

Demonstrate that you understand how to create and configure Magento JavaScript mixins.

Magento 2 has lots of off-the-shelf functions. When it comes to overriding the javascript component or extending the javascript component in Magento 2, you need to use mixins.

Mixins are JavaScript files located in the web/js directory. A mixin file can be in several directories if these directories are in web/js.

An example of creating a mixin for custom product samples:

1: Create app/code/VendorName/ModuleName/view/frontend/requirejs-config.js

```
var config = {
    config: {
        mixins: {
            'Magento_Swatches/js/swatch-renderer': {
                'VendorName_ModuleName/js/swatch-renderer-mixin': true
            }
        }
    }
};
```

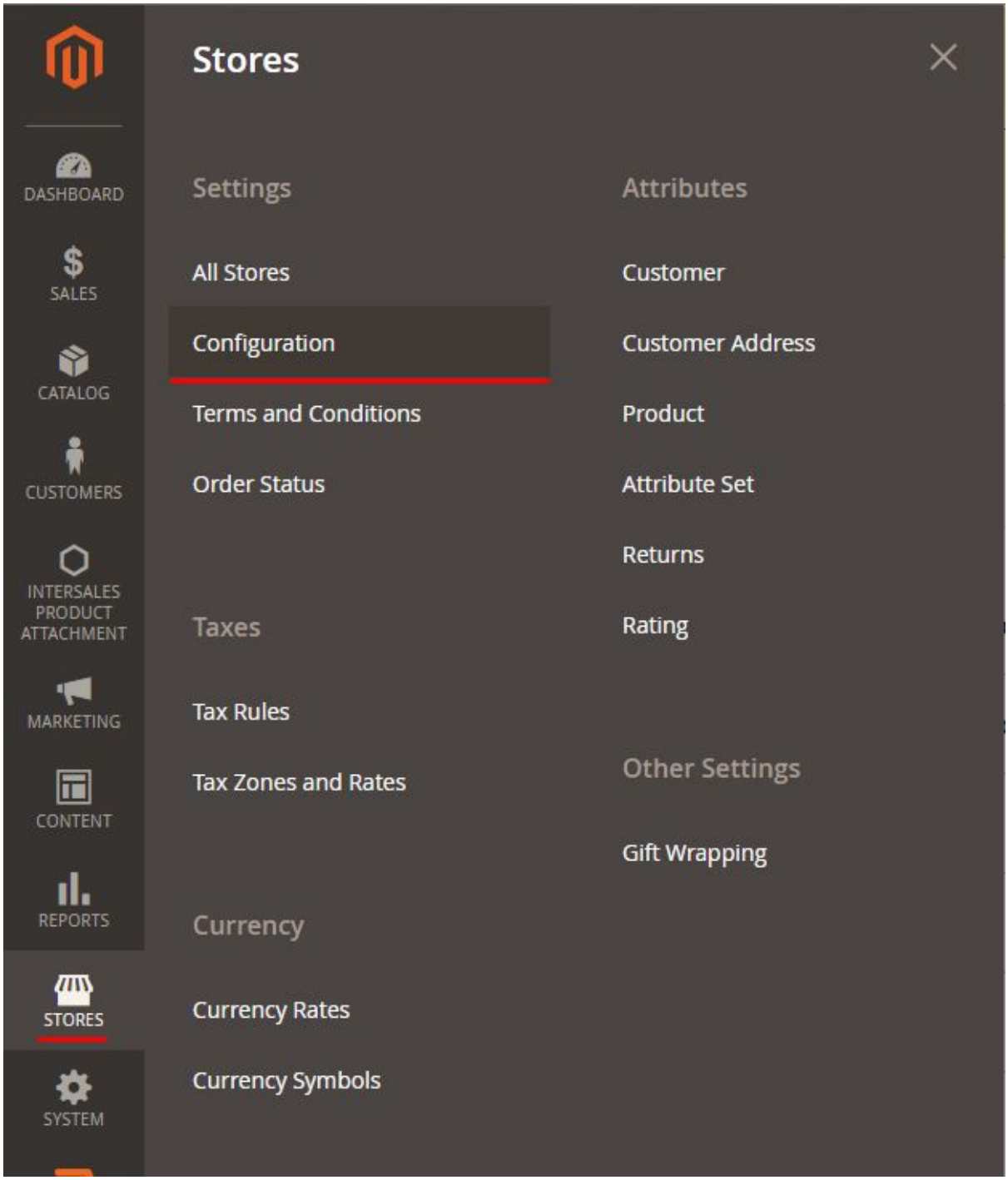
Step 2: Create app/code/VendorName/ModuleName/view/frontend/web/js/swatch-renderer-mixin.js

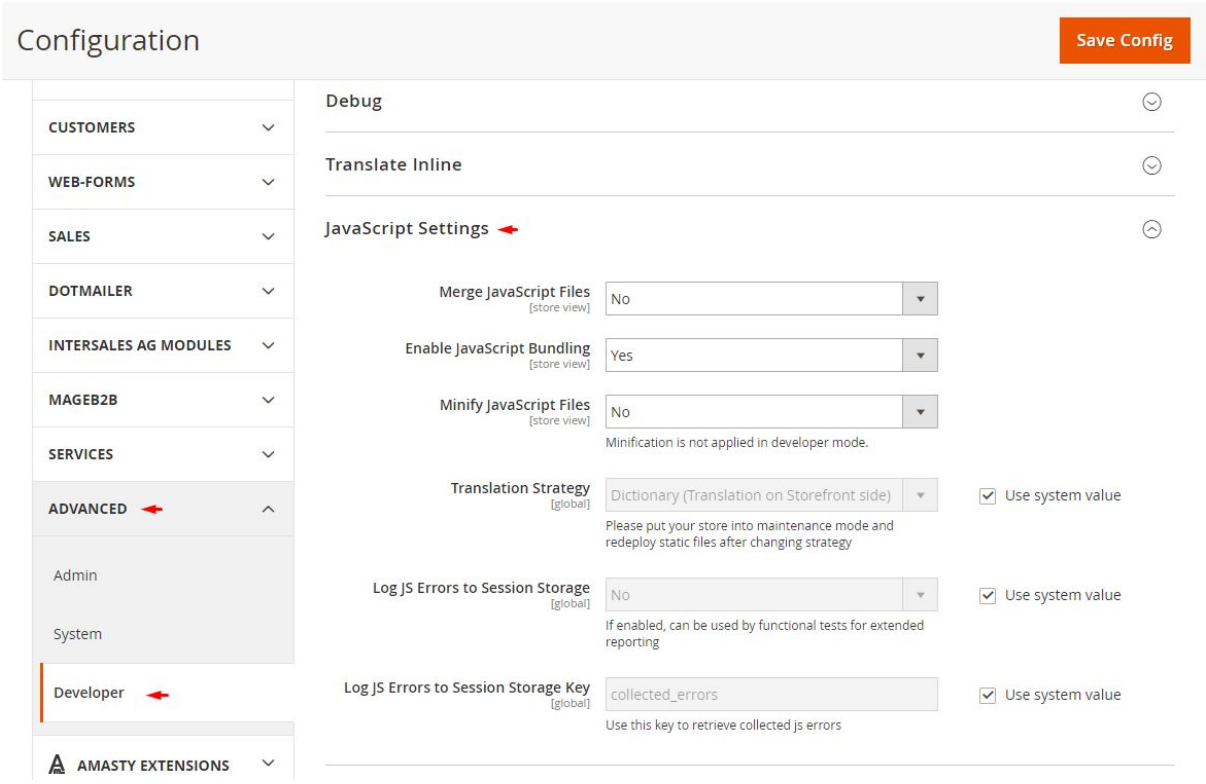
```
define(['jquery'], function ($) {
    'use strict';

    return function (SwatchRenderer) {
        $.widget('VendorName.SwatchRenderer',
$['mage']['SwatchRenderer'], {
            _init: function () {
                console.log('getProductSwatchRenderer');
                this._super();
            }
        });
        return $['mage']['SwatchRenderer'];
    };
});
```

6.4 Configure JavaScript merging and minify in the Admin UI

What options are available to configure JavaScript minification and bundling?

Options responsible for merging and minify JS files configuration are on the path:
Admin UI > Stores > Configuration > Advanced tab > Developer > JavaScript Settings


Stores
DASHBOARD
SALES
CATALOG
CUSTOMERS
INTERSALES PRODUCT ATTACHMENT
MARKETING
CONTENT
REPORTS
STORES
SYSTEM
Settings
All Stores
Configuration
Terms and Conditions
Order Status
Taxes
Tax Rules
Tax Zones and Rates
Currency
Currency Rates
Currency Symbols
Attributes
Customer
Customer Address
Product
Attribute Set
Returns
Rating
Other Settings
Gift Wrapping



The “Developer” tab is displayed if the site is in Developer’s mode. To switch the site to this mode, you must enter the magento server in console:

```
bin/magento deploy:mode:set developer
```

If everything is entered correctly, the console will display the following:

```
Switched to developer mode.
```

How does Magento minify JavaScript?

The point of js files minification is that everything useless (spaces, lines breaks, etc.) is removed from the file code, thereby reducing the size (weight), which positively affects the loading of the site as a whole.

What is the purpose of JavaScript bundling and minification?

Before turning on building options, all js files are loaded alternately. Many files - many server requests during the site loading.

A lot of files - a lot of server requests when loading a site. JavaScript bundling combines all the files into one, therefore there will be only one request to the server, and with the files minification (their weight reduction) the merged file will be loaded faster.

All this optimizes the site and reduces page loading time.

There’re also some disadvantages. You can exclude some scripts from the package, but you cannot select specific scripts for a specific page.

With Merging, you can only merge Javascript files that are not loaded via RequireJS. If you want to “merge” JS modules, you have to use packaging.

In Vendor/Theme/etc/view.xml you can resize the package and exclude some scripts from the package.

```
182     <vars module="Magento_Catalog"...>
246     <vars module="Magento_Bundle"...>
249     <vars module="Magento_ConfigurableProduct"...>
252     <vars module="Js_Bundle">
253         <var name="bundle_size">1MB</var>
254     </vars>
255     <exclude>
256         <item type="file">Lib::jquery/jquery.min.js</item>
257         <item type="file">Lib::jquery/jquery-ui-1.9.2.js</item>
258         <item type="file">Lib::jquery/jquery.ba-hashchange.min.js</item>
259         <item type="file">Lib::jquery/jquery.details.js</item>
260         <item type="file">Lib::jquery/jquery.details.min.js</item>
261         <item type="file">Lib::jquery/jquery.hoverIntent.js</item>
262         <item type="file">Lib::jquery/colorpicker/js/colorpicker.js</item>
263         <item type="file">Lib::requirejs/require.js</item>
264         <item type="file">Lib::requirejs/text.js</item>
265         <item type="file">Lib::date-format-normalizer.js</item>
266         <item type="file">Lib::legacy-build.min.js</item>
267         <item type="file">Lib::mage/captcha.js</item>
268         <item type="file">Lib::mage/dropdown_old.js</item>
269         <item type="file">Lib::mage/list.js</item>
270         <item type="file">Lib::mage/loader_old.js</item>
271         <item type="file">Lib::mage/webapi.js</item>
272         <item type="file">Lib::mage/zoom.js</item>
273         <item type="file">Lib::mage/translate-inline-vde.js</item>
274         <item type="file">Lib::mage/requirejs/mixins.js</item>
275         <item type="file">Lib::mage/requirejs/static.js</item>
276         <item type="file">Magento_Customer::js/zxcvbn.js</item>
277         <item type="file">Magento_Catalog::js/zoom.js</item>
278         ...
304     </exclude>
305 </view>
306
```

The default size for the package is 1 MB. The packet size determines the number of packets that will be created.

For example, if you have 3 MB of script files and the package size is 1 MB, there will be created 3 packages. If the number is too small, you are likely to have 10 or more small bundles that will block each other during rendering, so be careful with this. Remember that packages aren't loaded asynchronously.

We can also exclude some scenarios from the kits. They will be downloaded from RequireJS if necessary.

Example script requests without bundling and merging JavaScript files:

website-rule.js	200	script	require.js:1895
messageList.js	200	script	require.js:1895
confirm.js	200	script	require.js:1895
sidebar.js	200	script	require.js:1895
jquery.highlight.js	200	script	require.js:1895
price-utils.js	200	script	require.js:1895
alert.js	200	script	require.js:1895
authentication-popup.js	200	script	require.js:1895
login.js	200	script	require.js:1895
form.js	200	script	require.js:1895
jquery.metadata.js	200	script	require.js:1895
minicart.js	200	script	require.js:1895
minicart.js	200	script	require.js:1895
local.js	200	script	require.js:1895
links.js	200	script	require.js:1895
messages.js	200	script	require.js:1895
popular.js	200	script	require.js:1895
navigation.js	200	script	require.js:1895
provider.js	200	script	require.js:1895
autocomplete.js	200	script	require.js:1895
injection.js	200	script	require.js:1895
quote_grid.js	200	script	require.js:1895
messages.js	200	script	require.js:1895
authentication-popup.js	200	script	require.js:1895
jquery.validate.js	200	script	require.js:1895
image.js	200	script	require.js:1895
totals.js	200	script	require.js:1895
totals.js	200	script	require.js:1895
collection.js	200	script	require.js:1895
element.js	200	script	require.js:1895
validation.js	200	script	require.js:1895
layout.js	200	script	require.js:1895
types.js	200	script	require.js:1895
invalidation-processor.js	200	script	require.js:1895
block-loader.js	200	script	require.js:1895
validation.js	200	script	require.js:1895
catalog-add-to-cart.js	200	script	require.js:1895
menu.js	200	script	require.js:1895
form-mini.js	200	script	require.js:1895

resizable.js	200	script	require.js:1895
renderer.js	200	script	require.js:1895
observable_source.js	200	script	require.js:1895
es6-collections.js	200	script	require.js:1895
events.js	200	script	require.js:1895
wrapper.js	200	script	require.js:1895
knockout-fast-foreach.js	200	script	require.js:1895
knockout-repeat.js	200	script	require.js:1895
url.js	200	script	require.js:1895
bound-nodes.js	200	script	require.js:1895
observable_array.js	200	script	require.js:1895
bootstrap.js	200	script	require.js:1895
engine.js	200	script	require.js:1895
scripts.js	200	script	require.js:1895
jquery-ui.js	200	script	require.js:1895
jquery.storageapi.min.js	200	script	require.js:1895
common.js	200	script	require.js:1895
jquery-migrate.js	200	script	require.js:1895
storage.js	200	script	require.js:1895
section-config.js	200	script	require.js:1895
wrs_env.js	200	script	content.js:74
collapsible.js	200	script	require.js:1895
knockout-es5.js	200	script	require.js:1895
knockout.js	200	script	require.js:1895
customer-data.js	200	script	require.js:1895
jquery-ui.js	200	script	require.js:1895
tabs.js	200	script	require.js:1895
tabs.js	200	script	require.js:1895
matchMedia.js	200	script	require.js:1895
key-codes.js	200	script	require.js:1895
text.js	200	script	require.js:1895
underscore.js	200	script	require.js:1895
amShopbyTopFilters.js	200	script	require.js:1895
bootstrap.js	200	script	require.js:1895
main.js	200	script	require.js:1895
ie-class-fixer.js	200	script	require.js:1895
smart-keyboard-handler.js	200	script	require.js:1895

A large number of scripts adversely affects page loading speed.

Here is an example of loading with the bundling function of JavaScript files:

logo.svg	200	svg+xml	(index)
wrs_env.js	200	script	content.js:74
add_product_to.js	200	script	:8080/:45
phones.js	200	script	:8080/:44
scripts.js	200	script	:8080/:43
owl.carousel.min.js	200	script	:8080/:42
requirejs-config.js	200	script	:8080/:41
mixins.js	200	script	:8080/:40
static.js	200	script	:8080/:39
bundle5.js	200	script	:8080/:38
bundle4.js	200	script	:8080/:37
bundle3.js	200	script	:8080/:36
bundle2.js	200	script	:8080/:35
bundle1.js	200	script	:8080/:34
bundle0.js	200	script	:8080/:33
require.js	200	script	:8080/:32
less.min.js	200	script	:8080/:31
config.less.js	200	script	:8080/:30
lords-logo.png	200	png	Other

We see that some scripts were loaded separately, through RequireJS and six scripts packaged in bundle.

Here is the example of loading scripts with bundle and merging JavaScript files functions:

logo.svg	200	svg+xml	:8080/:284
wrs_env.js	200	script	content.js:74
ce055950c6ab40249143b2f14bd9e6b3.js	200	script	(index)
less.min.js	200	script	:8080/:31
config.less.js	200	script	:8080/:30

Instead of some bundling JavaScript files, we get one merged file and several separate scripts loaded using RequireJS

6.5. UI component configuration in Magento 2

Each UI Component consists of the following elements combination:

- **XML declaration** (defines the internal structure of the component and its configuration parameters)
- **JavaScript class** (inherits from one of the base classes of the Magento JavaScript framework UI components, such as UIElement, UIClass, or UICollection)
- **Related templates** (using KnockoutJS bindings, UI Components can be bound to HTML templates).

UI Components can be used either for the admin panel or site's frontend.

How can specify configuration options on a UI Component widget in JSON and in Layout XML?

Before clarifying configuration options on a UI Component, it's necessary to explain the general rendering principles of the component in the Magento system. Initially, the server receives a page request. During response generation, the system receives the UI Component configuration as .xml declaration files. Then this configuration is changed by .php modifiers. After that, the combined configuration is packaged in JSON format and added to the HTTP response code from the server

As you can see, initially, configuration on UI Components is specified in .xml declaration. The example of such configuration you can see in the file *“app/code/Magento/Catalog/view/frontend/layout/default.xml”*:

```
<!-- -->
<arguments>
      <argument name="jsLayout" xsi:type="array">
            <item name="components" xsi:type="array">
                  <item name="compareProducts" xsi:type="array">
                        <item name="component"
xsi:type="string">Magento_Catalog/js/view/compare-products</item>
                  </item>
            </item>
      </argument>
</arguments>
<!-- →
```

Inside argument name=”jsLayout” we specify the data that needs to be displayed in this place and set the configuration on the components (change properties values, etc.). In our example it is a component with “compare products” in the left sidebar on the category page. The specified for item name = "component" the path to the file “Magento_Catalog/js/view/compare-products”. Rendering the page, the system will look for the component file “app/code/Magento/Catalog/view/frontend/web/js/view/compare-products.js”.

After rendering the page configuration of our component in the JSON format will look as follows:

```
<script type="text/x-magento-init">
{"[data-role=compare-products-link]": {"Magento_Ui/js/core/app": {
"components":{"compareProducts":{
```

```
"component":"Magento_Catalog\/js\/view\/compare-products"
}}
}}}
</script>
```

What configuration options are available on UI Components?

There are 2 UI Component types in Magento 2 - basic and secondary. There are 2 basic ones - Listing component and Form component. Other components are secondary and inherited from basic components.

Each UI Component has **default** properties. Parental components of this one (basic UI Components) also have **defaults**. Value of all these properties can be redefined in your configuration (the information on how to do it can be found above).

After page rendering, properties' values from the resulting JSON overwrite the values for the **defaults** properties. After that, the resulting properties become the first-level properties of the newly created UI Component instance, and the original **defaults** property values are deleted.

The full list of properties for each UI Components can be found on the official site

https://devdocs.magento.com/guides/v2.2/ui_comp_guide/bk-ui_comps.html

How do you specify the ko template for a UI Component?

Almost every UI Component contains a “template” property that sets the template for it. The value of this property is formed as {Module_Name}/{Path}/{Template_Name}, where
{Module_Name} is the name of the module where the template is located
{Path} - path to the template file, regarding the *view/<area>/web/template* folder in the module folder (if the template lies directly in this folder, the Path will be absent in the template address)
{Template_Name} - the name of the template without the suffix .html

For example, the file *app/code/Magento/Checkout/view/frontend/layout/checkout_index_index.xml* has a fragment of the UI Component configuration code:

```
<!-- -->
<arguments>
      <argument name="jsLayout" xsi:type="array">
            <item name="components" xsi:type="array">
                  <item name="checkout" xsi:type="array">
                        <item name="component"
xsi:type="string">uiComponent</item>
                                          <item name="config"
xsi:type="array">
                                                <item
name="template" xsi:type="string">Magento_Checkout/onepage</item>
                                          </item>
</item>
</item>
</argument>
</arguments>
<!-- -->
```

Here is the template located at
app/code/Magento/Checkout/view/frontend/web/template/onepage.html
which is connected as follows
<item name="template"
xsi:type="string">**Magento_Checkout/onepage**</item>

Here
Magento_Checkout is the module name with ko template
onepage - the ko template name without .html syffix
and since the template is at the root of the *view/<area>web/template* module folder, **{Path}** is absent.

Demonstrate an understanding of default.tracks

UI Component JS files can contain tracks special object. In this object you put those UI Component properties that can be changed dynamically and those changes we need to track. This object converts the properties placed into it in knockout-es5 observable (more about knockout-es5 observable can be found here https://knockoutjs.com/documentation/observables.html).

It has the following structure:

```
<!-- -->
defaults: {
      property1: 'value 1',
       property2: 'value 2',
       property3: 'value 3',
            tracks: {
            property2: true,
            property3: true
      }
},
<!-- -->
```

Now “property2” and “property3” will be tracked and when they are changed, the script from the JS file will run again.

We tried to outline the UI Component configuration in Magento 2 in general terms. For more details, we recommend to read the official documentation:
https://devdocs.magento.com/guides/v2.2/ui_comp_guide/bk-ui_comps.html
https://devdocs.magento.com/guides/v2.2/ui_comp_guide/concepts/ui_comp_config_flow_concept.html

6.6 Understanding knockout framework

Knockout.js is a library that allows creating dynamic interfaces using various bindings to DOM elements. It implements the Model-View-ViewModel pattern. In **Magento 2**, it is used in dynamic modules like Checkout, Customer and others. In this article, we refer to the module files that use knockout.js and explain the usage of the framework to perform various functions in **Magento 2**. In this article, we will answer the following questions:

1. How do you use knockout.js bindings?
2. How do you bind a ko view model to a section of the DOM with the scope binding?
3. How do you render a ko template of a UiComponent?
4. Demonstrate an understanding of the different types of knockout observables.
5. What common ko bindings are used?
6. Demonstrate an understanding of ko virtual elements.

Describe the architecture of the Knockout library: MVVC concept, observables, bindings.
MVVC (Model-View-ViewModel) has 3 components:

Model contains data
View shows data on the screen
ViewModel is the intermediary between Model and View, that allows getting data for View and impact Model.

How do you use knockout.js bindings?

Bind is the ability to bind various components to HTML.
In knockout.js we use the html **data-bind** attribute for this as follows:
On frontend **Magento 2** as an example of a user greeting component, a link to a file on github:
https://github.com/magento/magento2/blob/2.2/app/code/Magento/Customer/view/frontend/templates/account/customer.phtml

```
<span data-bind="text: customer().fullname"></span>
```

If we assume that the **fullname** value is observable, then as soon as the fullname value changes, this value changes immediately on the site. There are many different binds and all of them are described in documentation (https://knockoutjs.com/documentation/introduction.html) knockout.js, and here are some of them: visible, text, html, if, foreach, click.
Magento 2 also has its custom binds which can be specified in **data-bind** attribute. Some of them can be specified as an attribute, **knockout.js** virtual element and a custom element. There are some of them:
1. `<!-- ko template: getTemplate() --><!-- /ko -->` is responsible for the output of the template specified in the component, here is the function responsible for the output:
https://github.com/magento/magento2/blob/2.2/app/code/Magento/Ui/view/base/web/js/lib/core/element/element.js#L266

```
getTemplate: function () {
            return this.template;
        }
```

2. `<!-- ko i18n: 'International text' --><!-- /ko -->` is a virtual element used for translation in **Magento 2**. You may say that you can just write it in `<!-- ko text: -->` but no. It is specially designed to integrate everything with translations in **Magento 2**. It can be used as an attribute and **data-bind**:

```
<span translate="international text" />
<span data-bind="i18n: 'international text'"></span>
```

3. `<span data-bind="afterRender: setModalElement"></span>` — used to add a function after an element rendering.
4. `<div data-bind="fadeVisble: isVisble"></div>` — used for the smooth appearance or disappearance of an element depending on the function result
5. `<div data-bind="mageInit: 'mywidget':{'configuration-value':true}"></div>` — used to initialize the jqueryUi widget.

Knockout bindings:

1. Appearance:
    a. visible - show/hide the item depending on the condition
    b. text - displays text, html tags are displayed as text
    c. html - displays html, html tags will be rendered
    d. css - add css class depending on the condition
    e. style - add css style depending on the condition
    f. attr - set attribute values
2. Flow:
    a. foreach - duplicates the contents of the element for each item in the specified array
    b. if - adds the contents of a condition block to the DOM, if it's true

c. ifnot - adds the contents of a condition block to the DOM if it is false
d. with - creates a new bundling property context in which you can directly access the subproperties of this property.
e. component - embeds an external component in the DOM

3. Work with form fields:
    a. click - function call on click
    b. event - function call upon event occurrence
    c. submit - function call on submit
    d. enable - turns on the element when the condition is true
    e. disable - turns off the element when the condition is true
    f. value - changes the value of a form field
    g. textInput - similar to value, it works only with text fields, provides instant updates from the DOM for all types of user input, including autocomplete, drag-and-drop, and clipboard events
    h. hasFocus - changes the value of a property upon receipt (if it's true) and loss of focus (if it's false)
    i. checked - changes the value of a property when selecting checkbox, radio elements
    j. options - provides options for dropdown
    k. selectedOptions - provides selected options of dropdown
    l. uniqueName - sets a unique name for empty form fields

4. Template rendering:
    a. template - renders another template

The other custom bindings can be found in the documentation. https://devdocs.magento.com/guides/v2.1/ui_comp_guide/concepts/knockout-bindings.html and https://devdocs.magento.com/guides/v2.3/ui_comp_guide/concepts/magento-bindings.html

How do you bind a ko view model to a section of the DOM with the scope binding?

It binds an element with a UI Component that is already declared in xml in **jsLayout** or in **<script type="text/x-magento-init">** and registered in **uiRegistry**.

On the checkout example:

xml:

```
<referenceContainer name="content">
            <block class="Magento\Checkout\Block\Onepage"
name="checkout.root" template="Magento_Checkout::onepage.phtml"
cacheable="false">
                <arguments>
                    <argument name="jsLayout" xsi:type="array">
                        <item name="components" xsi:type="array">
                            <item name="checkout" xsi:type="array">
                                <item name="component"
xsi:type="string">uiComponent</item>
                                <item name="config" xsi:type="array">
                                    <item name="template"
xsi:type="string">Magento_Checkout/onepage</item>
                                </item>
                            </item>
                        </argument>
                </arguments>
            </block>
</referenceContainer>
```

In the line `<item name="checkout" xsi:type="array">` in the name attribute, we write the component name in our case **checkout** and call it in phtml:

```
<div id="checkout" data-bind="scope:'checkout'"
```

```
class="checkout-container">
<!-- ko template: getTemplate() --><!-- /ko -->
</div>
```

In the **checkout** example we, firstly, created it in html, then it became attached to **UiRegistry** by parsing our specified xml settings and getting them using json using the php function **$block->getJsLayout().**

```
<script type="text/x-magento-init">
        {
            "#checkout": {
                "Magento_Ui/js/core/app": <?= /* @escapeNotVerified
*/ $block->getJsLayout() ?>
            }
        }
</script>
```

Also, if we bind a component to a DOM element through **data-mage-init**, the scope is bound to the element to which we bind the component or widget.
Example:

```
<div id="checkout-loader" data-role="checkout-loader"
class="loading-mask" data-mage-init='{"checkoutLoader": {}}'>
```

#checkout-loader element becomes a scope of **checkoutLoader** component.

How do you render a ko template of a UiComponent?

Knockout templates can be specified in different ways. Here we will try to cover possible ways to specify templates for UiComponent.

The first way is to indicate a specific component in the viewmodel file, for example, as it's done in the authorization pop-up form: https://github.com/m1kash/magento2/blob/2.2-develop/app/code/Magento/Customer/view/frontend/web/js/view/authentication-popup.js#L28
Code:

```
define([...], function (...) {
    'use strict';

    return Component.extend({
        defaults: {
            template: 'Magento_Customer/authentication-popup'
        },
    });
});
```

In the above mentioned code we see that the standard `Magento_Customer/authentication-popup` template will be used, when creating our custom UI Component, we can specify our template through mixin, layout, text/x-magento-init, data-mage-init.
The second way. We can specify our template in **text/x-magento-init**

```
<script type="text/x-magento-init">
    {
        "*": {
            "Magento_Ui/js/core/app": {
                "components": {
                        "customComponent": {
                            "template": "Magento_Theme/template",
                            "component":
"Magento_Theme/js/view/custom-component"
                        }
                    }
                }
```

```
            }
    }
</script>
```

The third way. We can specify our template in **jsLayout** like in checkout**:**

```
<item name="summary" xsi:type="array">
  <item name="component"
xsi:type="string">Magento_Checkout/js/view/summary</item>
  <item name="displayArea" xsi:type="string">summary</item>
  <item name="config" xsi:type="array">
  <item name="template"
xsi:type="string">Magento_Checkout/summary</item></item>
  <item name="children" xsi:type="array"></item>
</item>
```

For the Magento_Checkout/js/view/summary component apply the template that we specified in xml: **Magento_Checkout/summary**.
The fourth way to specify a template is directly in the template where our component is called, for example:

```
<!-- ko template: "BelVG_CustomModule/custom-template"
--><!-- /ko →
```

For the first 3 ways, we need to specify a template call in the template, for example, as it was done in checkout:
https://github.com/magento/magento2/blob/2.2/app/code/Magento/Checkout/view/frontend/templates/onepage.phtml#L17

Code:

```
<div id="checkout" data-bind="scope:'checkout'"
class="checkout-container">
    <!-- ko template: getTemplate() --><!-- /ko -->
```

```
    <script type="text/x-magento-init">
        {
            "#checkout": {
                "Magento_Ui/js/core/app": <?= /* @escapeNotVerified
*/ $block->getJsLayout() ?>
            }
        }
    </script>
</div>
```

As you can see from the code, we declared the scope for the #checkout element: and called the **getTemplate ()** function through the virtual element, which makes the function visible in github (https://github.com/magento/magento2/blob/2.2/app/code/Magento/Ui/view/base/web/js/lib/core/element/element.js#L266)

it just gets a link to the static address of our template. And after all this, it renders our template.
On the example of checkout component template github(https://github.com/magento/magento2/blob/2.2/app/code/Magento/Checkout/view/frontend/web/template/onepage.html):

```
<!--
/**
 * Copyright (c) Magento, Inc. All rights reserved.
 * See COPYING.txt for license details.
 */
-->
<!-- ko foreach: getRegion('authentication') -->
<!-- ko template: getTemplate() --><!-- /ko -->
<!--/ko-->

<!-- ko foreach: getRegion('progressBar') -->
<!-- ko template: getTemplate() --><!-- /ko -->
<!--/ko-->
```

```
<!-- ko foreach: getRegion('estimation') -->
    <!-- ko template: getTemplate() --><!-- /ko -->
<!--/ko-->

<!-- ko foreach: getRegion('messages') -->
    <!-- ko template: getTemplate() --><!-- /ko -->
<!--/ko-->
<div class="opc-wrapper">
    <ol class="opc" id="checkoutSteps">
    <!-- ko foreach: getRegion('steps') -->
        <!-- ko template: getTemplate() --><!-- /ko -->
    <!--/ko-->
    </ol>
</div>

<!-- ko foreach: getRegion('sidebar') -->
    <!-- ko template: getTemplate() --><!-- /ko -->
<!--/ko-->
```

On the template you can see that we use **foreach**, alternately displaying data from an array or $ date object. For example, in the last **foreach**, **getRegion ()** is used to get the sidebar template and so on. The function code is in github (https://github.com/m1kash/magento2/blob/2.2-develop/app/code/Magento/Ui/view/base/web/js/lib/core/collection.js#L205).

It's all about template rendering on Magento 2. Now, we come to the next question.

Demonstrate an understanding of the different types of knockout observables.

Observables are observed objects and arrays that are updated asynchronously when the value changes. In **knockout.js**, this is implemented by following functions:

1. `ko.observable()` — the function is used for observable objects to asynchronously change values, for example, when retrieving data from json.
2. `ko.observableArray([])` — the function is used for observable arrays, which can also be updated asynchronously. They change the values and use methods for arrays such as: **.map**, **.filter**, **.each**, etc.
If we want to assign a value to the observed object or array, we give our value to the function argument:

```
var Name = ko.observable(null);
Name('New value!');
```

```
> var Name = ko.observable(null);
< undefined
> Name()
< null
> Name('New value!');
< ▶Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, parent: Window, …}
>
```

If we want to get the value of the observed object or array, we call our function without arguments:
`Name();`

```
> Name();
< "New value!"
>
```

You can subscribe to observable objects or arrays to change the value using the **subscribe** method:
1. To get the value before assignment you need to write the following construction:

```
Name.subscribe(function(oldValue) {
    alert("Old value: " + oldValue);
}, null, "beforeChange");
```

2. To get the value after assignment you need to write the following construction:

```
Name.subscribe(function(newValue) {
    alert("New value: " + newValue);
});
```

There are also calculated functions:

1. ko.computed(**callback**) is used to update the value when changing an observable object used in this function.

2. ko.pureComputed(**callback**) is used to return observable objects without updating. Due to this, we get more performance and less memory consumption to compare with the **ko.computed** function.

Observables examples:

html:

```
The name is <span data-bind="text: name"></span>
<button data-bind="click: change">Change</button>
```

script:

```
var myViewModel = {
    name: ko.observable('Bob'),
    change: function() {
      this.name('Alice');
    }
};
ko.applyBindings(myViewModel);
```

output before pressing the Change button:
The name is Bob

Output after pressing the Change button:
The name is Alice

What common ko bindings are used?

As it was mentioned before, you can find more about bindings in the **knockout.js** documentation (https://knockoutjs.com/documentation/introduction.html). Let's consider some of the main ones used in Magento:
1. `<div data-bind="visible: Observer()">` is responsible for the visibility of the element and takes the value **boolean**.
2. `<div data-bind="text: ObserverText()">` is used to display text
3. `<p data-bind="html: ObserverElements()">` is used to output html.
4. `<form data-bind="attr: {'data-hasrequired': $t('* Required Fields')}">` — **attr** is user to uotput attributes
5. `<button data-bind="click: function">` is used to bind a function to an event when it clicks on an element, the function is callback, then we indicate only the name without calling.

Other bindings you can find on the official website - **knockout.js** - in documentation.

Demonstrate an understanding of ko virtual elements.

Virtual elements are child element bindings with no need to embed html to anchor the element. Virtual elements are used to render an html document. For example, how this is done in the payment methods template: (https://github.com/m1kash/magento2/blob/2.2-develop/app/code/Magento/Checkout/view/frontend/web/template/payment.html)

```
<!-- ko foreach: getRegion('payment-methods-list') -->
<!-- ko template: getTemplate() --><!-- /ko -->
<!-- /ko -->
```

We will also analyze the main virtual elements that are used in Magento 2:
1. `<!-- ko foreach: $data --><!-- /ko -->` is used to sort arrays to display date from the array.
2. `<!-- ko template: getTemplate() --><!-- /ko -->` as it was mentioned before is used to output data from the array according to the pattern that we set in template.
3. `<!-- ko if: function()--><!-- /ko -->` is used to check the conditions for displaying data from the condition.There is also `<!-- ko ifnot: conditionFunc() --><!-- /ko -->` for the opposite conditions.
5. `<!-- ko i18n: 'of' --><!-- /ko -->` is used for translations in Magento 2, what was written in the first section.
6. `<!-- ko text: getText() --><!-- /ko -->` is used to display text from functions, observable objects, etc.
So, we considered all the basic functions and features of knockout.js in Magento 2. Below we will consider how to create a component and to use child ones.

What are the pros and cons of knockout templates?

Pros:

1. Support for older browsers up to IE 6
2. Simple library
3. Separation of HTML and JavaScript
4. Dynamic data binding

Cons:

1. Poor performance if there are a lot of objects
2. Magento is switching to PWA, using React, and Knockout may sooner or later become deprecated.

Compare knockout templates with underscore JavaScript templates.

1. Underscore template is rendered only when the _.template (template) (params) method is called, the Knockout template is automatically rendered if the data changes.
2. Underscore uses the blocks <% = ...%>, <% ...%>, <% - ...%> to run scripts. Knockout uses the data-bind attribute in html elements and html comments.

Demonstrate understanding of the knockout-es5 library

Knockout observables get and change as functions:

```
var value = this.knockoutProperty(); // get
this.knockoutProperty(value); // set
```

Knockout-es5 library allows simplifying the knockout observable getting/changing. The library uses ES5 getter/setter.

```
var value = this.knockoutProperty; // get
this.knockoutProperty = value; // set
```

There also can be used aperators += , -= , *= , /=

```
this.knockoutProperty += value; // set
```

What is identical to the expression:

```
this.knockoutProperty(this.knockoutProperty() + value)
```

In order to apply the plugin to the model, you need to call ko.track (someModel);
Additionally, you can specify an array of model fields as the second parameter to limit the fields to which this plugin is applied: ko.track `(someModel, ['firstName', 'lastName', 'email']);`

Now to get the original observable, we need to call

```
ko.getObservable(someModel, 'email')
```

The plugin also makes it easier to get the value of the calculated functions. For example, instead of

```
<span data-bind="text: getSubtotal()"></span>
```

We can write

```
<span data-bind="text: subtotal"></span>
```

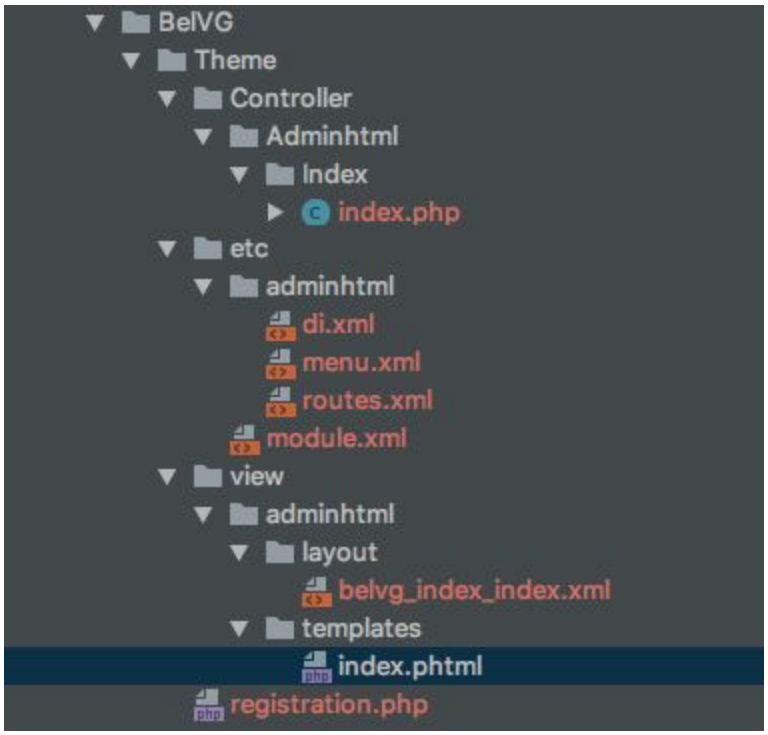
We need to call the following code:

```
ko.defineProperty(someModel, 'subtotal', function() {return
this.price * this.quantity; });
```

6.7 Understanding dependency between components

Demonstrate an understanding of the links, imports, exports, and listens UiComponent configuration directives.

Links, imports, exports and listens are used for binding components to demonstrate how binding works. We need two UI Components and a module with the following contents:



The module is created in the article on **areas in Magento** (Relink). We will add the missing files to display our module in the admin panel menu and as a separate page:

app/code/BelVG/Theme/etc/adminhtml/menu.xml —

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Backend:etc
/menu.xsd">
    <menu>
        <add id="BelVG_Theme::belvg"
             title="BelVG"
             module="BelVG_Theme"
             sortOrder="55"
```

```
            resource="BelVG_Theme::belvg"
            action="belvg"
        />
    </menu>
</config>
```

app/code/BelVG/Theme/etc/adminhtml/routes.xml —

```
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:App/etc/routes.x
sd">
    <router id="admin">
        <route id="belvg" frontName="belvg">
            <module name="BelVG_Theme" />
        </route>
    </router>
</config>
```

app/code/BelVG/Theme/Controller/Adminhtml/Index/index.php —

```
<?php
namespace BelVG\Theme\Controller\Adminhtml\Index;

class Index extends \Magento\Backend\App\Action
{
    /**
     * @var \Magento\Framework\View\Result\PageFactory
     */
    protected $resultPageFactory;

    /**
     * Constructor
     *
     * @param \Magento\Backend\App\Action\Context $context
     * @param \Magento\Framework\View\Result\PageFactory
$resultPageFactory
     */
    public function __construct(
        \Magento\Backend\App\Action\Context $context,
        \Magento\Framework\View\Result\PageFactory $resultPageFactory
```

```
    ) {
        parent::__construct($context);
        $this->resultPageFactory = $resultPageFactory;
    }

    /**
     * Load the page defined in
view/adminhtml/layout/belvg_index_index.xml
     *
     * @return \Magento\Framework\View\Result\Page
     */
    public function execute()
    {
        $resultPage = $this->resultPageFactory->create();
        $resultPage->getConfig()->getTitle()->prepend(__("Custom
Component"));
        return $resultPage;
    }
}
?>
```

app/code/BelVG/Theme/view/adminhtml/layout/belvg_index_index.xml

```
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_configuration.xsd">
    <body>
        <head>
            <title>Custom Component</title>
        </head>
        <referenceContainer name="content">
            <block class="Magento\Backend\Block\Template"
template="BelVG_Theme::index.phtml"/>
        </referenceContainer>
    </body>
</page>
```

This is the module we will need to see the UI Component binding, and now we will create our components ourselves:

app/code/BelVG/Theme/view/adminhtml/web/js/component-one.js —

```
define([
    'jquery',
    'uiComponent'
],function ($, Component) {
    'use strict';
    return Component.extend({
        defaults: {
            title: 'Component One',
            value: 'Value text'
        }
    });

});
```

app/code/BelVG/Theme/view/adminhtml/web/js/component-two.js —

```
define([
    'jquery',
    'uiComponent'
], function ($, Component) {

    return Component.extend({
        defaults : {
            title: 'Component One',
            value: 'Value text'
        }
    });

});
```

Declare them in the template

app/code/BelVG/Theme/view/adminhtml/templates/index.phtml —

```
<div class="row">
    <div class="col-m-6">
        <div class="dashboard-item" data-bind="scope:
'componentOne'">
            <div class="dashboard-item-title" data-bind="text:
```

```
title"></div>
            <div class="dashboard-item-content">

            </div>
        </div>
    </div>
    <div class="col-m-6">
        <div class="dashboard-item" data-bind="scope:
'componentTwo'">
            <div class="dashboard-item-title" data-bind="text:
title"></div>
            <div class="dashboard-item-content">

            </div>
        </div>
    </div>
</div>
<script type="text/x-magento-init">
    {
        "*": {
            "Magento_Ui/js/core/app": {
                "components": {
                    "componentOne": {
                        "component": "BelVG_Theme/js/component-one"
                    },
                    "componentTwo": {
                        "component": "BelVG_Theme/js/component-two"
                    }
                }
            }
        }
    }
</script>
```

Consider the imports directive. The imports directive is used to pass values, let's add imports and Observable to observe our value property. add our imports directive to the defaults object in the second component:

```
imports: {
                value: 'componentOne:value'
         }
```

where
componentOne is the name of the UiRegistry component from which the value will be taken.
value is the name of the property to be taken.

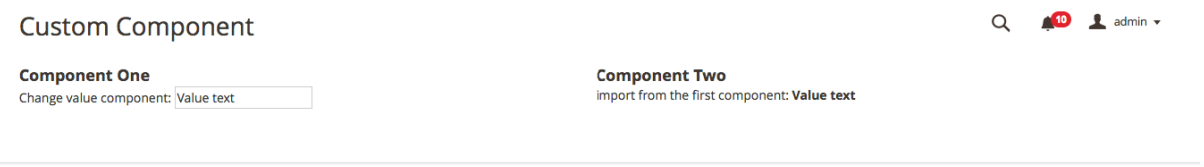
and into our two observer components:

```
       initObservable: function () {
            this._super().observe(['value']);

            return this;
        }
```

And also add to **BelVG/Theme/view/adminhtml/templates/index.phtml** in the block **.dashboard-item-content[data-bind="scope: 'componentOne'"]**

```
<p>Change value component: <input  data-bind="value: value" ></p>
And the second component .dashboard-item-content[data-bind="scope:
'componentTwo'"]
<p>import from the first component: <strong data-bind="text: value"
></strong></p>
```

We have the following page:



http://prntscr.com/ma50wt

When we change the value of input (defaults.value), our second component will import this value from the first. It will immediately be displayed in the second component.
Now let's move on to the **exports** directive:

To see how we export our values, we need to remove the imports directive in our second component, and add the exports directive to the first component:

```
exports: {
     value: 'componentTwo:value'
}
```

Now it can be seen how we imported the values from the second to the first component. We export our values from the first to the second one. When you follow the instructions you will see it.

Now move to the **links** directory:
The **links** directive is used to bind properties. You will have the opportunity to change which properties are connected in any component. To implement this function, we need to redo our layout of the component to demonstrate the connection:
Change in **BelVG/Theme/view/adminhtml/templates/index.phtml** in the block **.dashboard-item-content[data-bind="scope: 'componentTwo'"]**

```
<strong data-bind="text: value" ></strong>
```

to

```
<input  data-bind="value: value" >
```

And we change the directive in the first components from **exports** to **links:**

```
 links: {
                value: 'componentTwo:value'
            }
```

And also replace our initObservable function with tracks:
Delete initObservable to get the following file (refers to two components):
app/code/BelVG/Theme/view/adminhtml/web/js/component-one.js

```
define([
    'jquery',
    'uiComponent'
```

```
],function ($, Component) {
    'use strict';
    return Component.extend({
        defaults: {
            title: 'Component One',
            value: 'Value text',
            links: {
                value: 'componentTwo:value'
            },
            tracks: {
                value: true
            }
        }
    });
});
```

app/code/BelVG/Theme/view/adminhtml/web/js/component-two.js

Now when we go into our module we will change the value in any component they should be synchronized:

http://prntscr.com/ma6frs

Tracks object is used to convert our property to an observable object. If we use the methods in question, we use this object.

Consider the **listens** directory**,** delete the **links** directory and add:

```
            listens: {
                'value': 'valueChanged'
            },
```

where

value is the name of the property to be tracked.

valueChange is a function that will be called when the value changes.

As a result, we get this component

```
define([
    'jquery',
    'uiComponent',
    'Magento_Ui/js/modal/alert'
],function ($, Component, alert) {
    'use strict';
    return Component.extend({
        defaults: {
            title: 'Component One',
            value: 'Value text',
            listens: {
                value: 'valueChanged'
            },
            tracks: {
                value: true
            }
        },
        valueChanged: function (val) {
            alert({
                content: 'value changed to <strong>' + val +
'</strong>'
            });
        }
    });

});
```

The essence of this listens property is that when the value that we are listening to changes, an alert pops up. So, even if you export from the second component to the first, the property will also change, and due to this our alert will pop up.

Module sources:
https://github.com/m1kash/belvg-theme-module

6.8 Understanding string templates

Demonstrate an understanding of ES5 string literal templates like ${$.provider}.

Template literals in **Magento 2** are written **$ {$.variable}** in single quotes ‘, and in ES6 using` template literals is most often used, where $ is our component or this(https://github.com/magento/magento2/blob/2.3-develop/lib/web/mage/utils/template.js#L104)in UI Components if you look at the Listing (grid) component, for example (https://github.com/magento/magento2/blob/2.3-develop/app/code/Magento/Ui/view/base/web/js/grid/listing.js#L63)

```
defaults: {
            template: 'ui/grid/listing',
            listTemplate: 'ui/list/listing',
            stickyTmpl: 'ui/grid/sticky/listing',
            viewSwitcherTmpl: 'ui/grid/view-switcher',
            positions: false,
            displayMode: 'grid',
            displayModes: {
                grid: {},
                list: {}
            },
            dndConfig: {},
            editorConfig: {
                name: '${ $.name }_editor',
                component: 'Magento_Ui/js/grid/editing/editor',
                columnsProvider: '${ $.name }',
                dataProvider: '${ $.provider }',
                enabled: false
```

```
        },
        resizeConfig: {
            name: '${ $.name }_resize',
            columnsProvider: '${ $.name }',
            component: 'Magento_Ui/js/grid/resize',
            enabled: false
        },
        imports: {
            rows: '${ $.provider }:data.items'
        },
        listens: {
            elems: 'updatePositions updateVisible',
            '${ $.provider }:reload': 'onBeforeReload',
            '${ $.provider }:reloaded': 'onDataReloaded'
        },
        modules: {
            dnd: '${ $.dndConfig.name }',
            resize: '${ $.resizeConfig.name }'
        },
        tracks: {
            displayMode: true
        },
        statefull: {},
```

As you can see in most configuration parameters, template literals are used. We can also use the listens which we mentioned before (https://docs.google.com/document/d/1sG7NK-mYTEZbiKluhJlO3WruraQljyNcu7sdxmT35Ks/edit):

```
imports: {
            rows: '${ $.provider }:data.items'
        },
        listens: {
            elems: 'updatePositions updateVisible',
            '${ $.provider }:reload': 'onBeforeReload',
```

```
            '${ $.provider }:reloaded': 'onDataReloaded'
        }
```

Here we see when our component is restarted, the onDataReloaded method will be launched dynamically.

Separator :

(https://github.com/magento/magento2/blob/2.3-develop/app/code/Magento/Ui/view/base/web/js/lib/core/element/links.js)

As it can be seen from the **${ $.provider }:reloaded** value, we separate the values of **${ $.provider }(target) and reloaded(property),** if we work in a browser that supports (template strings ES6), then it processes them like template string ES 6 (https://github.com/magento/magento2/blob/2.3-develop/lib/web/mage/utils/template.js#L46) if it does not support, it is processed through the underscore template (http://underscorejs.ru/#template).

If we check the back of uiComponents, we will see that all these values are dynamic and are configured in the files of the ui components. For example, product_listing

product_listing

(https://github.com/magento/magento2/blob/2.3-develop/app/code/Magento/Catalog/view/adminhtml/ui_component/product_listing.xml)

Code excerpt:

```
<argument name="data" xsi:type="array">
        <item name="js_config" xsi:type="array">
            <item name="provider"
xsi:type="string">product_listing.product_listing_data_source</item>
        </item>
    </argument>
```

As you can see from the component Ui code, we pass the provider variable to js_config. And later these parameters are generated in json in a tag. We can also write parameters through the underscore which is then generated in camelCase

```
<script type="text/x-magento-init">
       "product_listing": {
          "deps": [
            "product_listing.product_listing_data_source"
          ],
          "provider": "product_listing.product_listing_data_source"
         },
</script>
```

then it is all processed app.js(https://github.com/magento/magento2/blob/2.3-develop/app/code/Magento/Ui/view/base/web/js/core/app.js).

What does $. Inside of ${ } resolve to?

$ {} Is the ES6 template literal implemented by Magento 2 to substitute properties as in ES6 templates. Implementation code is (https://github.com/magento/magento2/blob/2.3-develop/lib/web/mage/utils/template.js)
When we write $. We turn to this http://prntscr.com/mqitcp

More details are here https://devdocs.magento.com/guides/v2.3/ui_comp_guide/concepts/ui_comp_template_literals.html

Section 7: Use LESS/CSS to Customize the Magento Look and Feel

7.1 Explain core concepts of LESS

LESS is a CSS addon that significantly extends its functionality by adding certain programming functions - logic, variables, calculation operations, reusable pieces of code and others (we will talk more about this additional functionality in this section).

LESS, first of all, is for developers and allows you to significantly speed up the process of creating styles for the site. It also allows you to quickly change and expand the ready made styles.

LESS files have the .less extension. Browsers do not recognise files with the .less extension and cannot apply styles from them to the pages. So, all .less files must be precompiled into familiar css files. Thus, we use so called compilers. For example, Magento has built in compilers for these purposes,allowing you to compile LESS files into CSS files both on the server side and on the client side (you can read more about this here - https://devdocs.magento.com/guides/v2.3/frontend-dev-guide/css-topics/css-preprocess.html#less_modes).
Magento also allows to compile LESS into CSS using a powerful tool like Grunt (you can read more about it here https://devdocs.magento.com/guides/v2.3/frontend-dev-guide/css-topics/css_debug.html).

More information about style files structure, UI libraries, style files connecting and others in Magento you can find in the articles below. But here we will pay attention to the work of LESS and show its benefits.

Describe features like file import via @import directive, reusable code sections via mixins together with parameters and the usage of variables.

As we mentioned before, LESS is a CSS addon. So, all the rules and syntax from a .css file work in a .less one. In other words, writing a regular CSS code in a .less file, the code will work without any issues after compilation.

Let's consider additional functions of LESS and ways to use them in work.

@import directive

Other .less or .css files can be imported into the .less file (the same can be done with CSS files, but there are more additional parameters in LESS). So, here we use the @import directive. It has the following structure:

@import 'path_to_file/file_name';

where
path_to_file is a path to the imported file as on whether to which file it is added;
file_name is the name of the imported file with the .less or .css extension, respectively (if the file has the .css extension, it will be processed as a .css file. If there is a .less extension or any other or not specified, it will be processed as a .less file)

The **@import** directive allows adding third-party files (from other sites). It has the following structure:

@import url('path_to_file/file_name');

The **@import** directive has additional parameters. It has the following structure:

@import (parameter) 'path_to_file/file_name';

The following parameters are available:
reference uses a less file, but does not display it at compilation if there is no link to it
inline - includes the source file in the output, but does not compile it (similarly to files with the .css extension)
less treats the file as a less one, regardless of what the file extension is (the same happens when files have the .less extension or any other or it is not specified)
css treats the file as a css file, no matter what the file extension is
once adds a file only once by default
multiple adds a file several times during multiple import
optional allows you to continue compilation if the file is not found (if there is no this parameter, there will be an error and compilation will stop in case the imported file is not found)

More information about parameters you can find here - http://lesscss.org/features/#import-atrules-feature

Magento has a finalized **@magento_import** directive that allows adding files from several places (more details you can find in the article below).

Magento also requires to name pasted files putting underscores at the beginning. For example, _example.less.

Variables

Variables in LESS have similar functionality with variables in programming. They are used to save some value and use it later in other places.

For example, we can create a variable for the shade of red and use the variable value in different styles (for example, set the color of text, borders, background and others). It might be not clear why we use the variable in this case, because you can simply set the color for the properties. But if you decide to change the shade of red, you just need to change the variable value. So, it will be automatically changed in all places of files where it's used (if you do not use the variable, you will have to change this color for all properties in all files manually. It will take more time, especially when there are lots of styles).

Now you understand why variables are used in LESS and how convenient it is. Let's see how the variables look in .less files.

The variable begins with @ icon and then goes the variable's name.

@example-variable

The example of a variable's usage:

```
@color-example: #222222;
a {
color:@color-example;
}
```

In the example above, we set the *@color-example* variable the *#222222* value. After compiling the .less file into a .css file, there will be the following rule:

```
a {
color:#222222;
}
```

Variables in LESS have some features that differ their behavior from those in programming languages. For example, we can use the variable value before declaring it; or, changing the variable value anywhere in the .less file (for example, at the end of the file), the variable value will be changed everywhere, not just in the rules that follow after the variable value has changed. First, compiling a .less file into a .css file, all variables are searched. The values of these variables will correspond to the values that were the last (for example, if we changed the variable value several times, everywhere will be applied only the last one). Then these values are substituted to all places where they are used.

Let’s consider it on the example:

```
.example-1 {
color:@color-example;
}
@color-example: red;
.example-2 {
color:@color-example;
}
@color-example: green;

Having compiled the .less file into the .css file, you'll see the
following:

.example-1 {
color:green;
}
.example-2 {
color:green;
```

```
}
```

Despite this behavior of the variables, it is good to put them in a separate file. It is very convenient, since you do not need to search for variables across all files to change their values.

The next thing you need to know is **the scope of the variable**.

In the example above, you saw variables that are declared globally, so they will be available anywhere in the .less file. However, if you declare a variable inside the CSS rule, it will be visible only inside this CSS rule. When you try to access this variable outside this rule there will be a compilation error.

Above you've seen globally declared variables - which are available in any place of a .less file. If you declare the variable inside the CSS rule, it will be visible only inside this CSS rule. Trying to access this variable outside the rule there will be a compilation error.

Example:

```
.example-1 {
  @color-example: yellow;
  color:@color-example;
}
.example-2 {
  color:@color-example;
}
```

As a result, the file won't be compiled.

You can change the global variable value inside the CSS rule. It will not affect the variable value outside this rule.
You can also assign the value of one variable to another one as follows:

```
@color-example-1: yellow;
@color-example-2: @color-example-1;
```

As a result, if you change the value of *@color-example-1* variable, the value of *@color-example-2* variable will be changed automatically.

More information about variables you can find here:
http://lesscss.org/features/#variables-feature

Mixins

There are common situations when different elements use a similar set of CSS properties
In a .css file, it was necessary to copy such properties for each element. In a .less file, it can be done by reusing the CSS rule - using **mixins**.

For example, many elements on the page will have similar animation. For this, you can create an .animation-1 class with a set of animation properties:

```
.animation-1 {
   transition: 300ms ease-in-out;
   -moz-transition: 300ms ease-in-out;
   -webkit-transition: 300ms ease-in-out;
   -o-transition: 300ms ease-in-out;
}
```

And then apply this mixin in places where it is necessary:

```
.example-1 {
   .animation-1();
   width: 100%;
}
```

As a result, after compiling the .less file into a .css file, the .example-1 element will have the following:

```
.animation-1 {
   transition: 300ms ease-in-out;
   -moz-transition: 300ms ease-in-out;
   -webkit-transition: 300ms ease-in-out;
   -o-transition: 300ms ease-in-out;
}
.example-1 {
   transition: 300ms ease-in-out;
   -moz-transition: 300ms ease-in-out;
   -webkit-transition: 300ms ease-in-out;
   -o-transition: 300ms ease-in-out;
   width: 100%;
}
```

You can't deny that it's very convenient. When we have changed the animation for the *.animation-1* mixin, it automatically changes everywhere where the mixin is used.

In the example above, during the compilation, instead of the *.animation-1* mixin, there were added properties which were set to it.

There are also mixins with parameters. Calling these mixins, we pass them parameter values. Creating such mixin, we recommend setting parameter values by default (since there might occur problems when calling the mixin

without specifying the parameter value). Let's make a mixin with parameters from the example above and see how to call it:

```
.animation-1 (
@animation-speed: 300ms,
@animation-type: ease-in-out
) {
   transition: @animation-speed @animation-type;
   -moz-transition: @animation-speed @animation-type;
   -webkit-transition: @animation-speed @animation-type;
   -o-transition: @animation-speed @animation-type;
}
.example-1 {
.animation-1(
@animation-speed: 1500ms
);
}
```

Mixin parameters are set in parentheses (there can be one or several parameters) with default values (default values are set after a colon). Calling a mixin in parentheses, we indicate the value of those mixin parameters that will differ from the default values. If we didn't specify any parameters adding a mixin as - *.animation-1*, the mixin would be added with standard parameter values.

As a result, after compiling the .less file into a .css file, the *.example-1* element will have the following:

```
.example-1 {
   transition: 1500ms ease-in-out;
   -moz-transition: 1500ms ease-in-out;
   -webkit-transition: 1500ms ease-in-out;
   -o-transition: 1500ms ease-in-out;
}
```

Magento has an agreement to name mixin parameters with @_ instead of @. So, the *@animation-speed* parameter from the example above would be named as *@animation-speed*.

You can read more about mixins here:
http://lesscss.org/features/#mixins-feature

Demonstrate your understanding of the special variable @arguments.

Inside the mixin, we can use a special variable - **@arguments**. It displays all the mixin one parameters in the order in which they are set.
For example, a mixin to set the border using the **@arguments** variable will look as follows:

```
.border-example(
@width: 2px,
@type: solid,
@color: red
){
   border:@arguments;
}
.example-1 {
.border-example(5px);
}
```

After compilation we get the following result for the *.example-1* block:

```
.example-1 {
   border: 5px solid red;
}
```

This variable reduces the amount of code in .less files.

Demonstrate how to use the nesting code formatting, and the understanding of media queries together with nesting.

CSS uses cascading styles everywhere. For example, to apply one style to all links, another one to all paragraphs and the third one to all *span* tags in a block with an *.error* class, we need to write the following rules inside an *.example-1* block:

```
.example-1 a {
      color: green;
}
.example-1 p {
      background: grey;
}
.example-1 .error span {
      color: red;
      border: 1px solid red;
}
```

When there are many such child elements and appear an additional nesting, then the code readability will noticeably deteriorate. In LESS, you can use the nesting of elements in each other which looks more logical and readable. Our LESS example will look as follows.

```
.example-1 {
   a {
      color: green;
   }
```

```
    p {
        background: grey;
    }
    .error {
        span {
            color: red;
            border: 1px solid red;
        }
    }
}
```

After compiling the .less file into a .css file, the code will look like the example above.

There can be lots of levels of such nesting. Magento style guidelines recommend avoiding more than three nesting levels. To avoid lots of levels, we recommend using the BEM methodology, creating element classes. More information about the BEM methodology you can find here - https://en.bem.info/methodology/quick-start/

In nesting you can also use media queries. For example, you want an element with the *.example-1* class on 768 pixels screens or less to change the indent from 20 to 10 pixels. For this, you can apply the following rule:

```
.element-selector {
    padding: 20px;

    @media screen and (max-width: 768px) {
    padding: 10px;
    }
}
```

As a result, after compiling the .less file into a .css file, you get the following:

```
.element-selector {
   padding: 20px;
   }
@media screen and (max-width: 768px) {
   .element-selector {
   padding: 10px;
   }
}
```

When there are lots of changes for a specific media request, we recommend writing a separate media request (without nesting) and add rules to it. This will reduce the final code length and simplify file processing.

Describe how the & (Ampersand) works and its function.

To refer to the current selector in LESS we use **& (Ampersand)**. It is useful when we need to set styles for various element states, its pseudo-elements, the next element in a row and more.. Let's see how it looks on an example:

```
.example-1 {
   color: red;

   &:hover{
      color: green;
   }
   &:before {
      content: "!!!";
      display: inline-block;
   }
   & + * {
```

```
        background: grey;
    }
    &.disabled {
        opacity: 0.5;
    }
    &_modify {
        font-size: 2rem;
    }
}
```

As a result, after compiling the .less file into a .css file, we have the following:

```
.example-1 {
    color: red;
}
.example-1:hover {
    color: green;
}
.example-1:before {
    content: "!!!";
    display: inline-block;
}
.example-1 + * {
    background: grey;
}
.example-1.disabled {
    opacity: 0.5;
}
.example-1_modify {
    font-size: 2rem;
}
```

It may seem that using **& (Ampersand)** is sophisticated, but it is very convenient and saves space in the file (especially with long selector class names).

As you've seen on the example above, we added *_modify* to **& (Ampersand)** and a new selector - *.example-1_modify*. It's name begins with the name of a current element - *.example-1*.

It is very convenient when we work with elements named according to the BEM methodology. However, we do not recommend doing this, since in this case it will be quite difficult to find the necessary element. In its "LESS coding standard," Magento also does not recommend this. It's better to write selectors entirely.

Describe how calculations are possible as well.

There is another one function in LESS - arithmetic calculations for properties like addition, subtraction, multiplication and division operations. It is especially useful to make calculations with variables.

For example:

```
.example-1 {
@unit: 3px;
    border:@unit solid #ddd;
    padding: @unit * 3;
    margin: 20px + 30px;
}
```

As a result, after compiling the .less file into a .css file, we have the following:

```
.example-1 {
   border: 3px solid #dddddd;
   padding: 9px;
   margin: 50px;
}
```

CSS has a ***calc()*** property value, which allows you to calculate the values of various formats. For example, the addition of percent and pixels:

```
.example-1 {
   width: calc(50% + 10px);
}
```

If we write this in a LESS file, after compilation we get the following rule:

```
.example-1 {
   width: calc(60%);
}
```

You can’t deny that we expected something different. To let the ***calc()*** save its value during compilation, it is necessary to escape this value as follows:

```
.example-1 {
   width: ~"calc(50% + 10px)";
}
```

We’ve explained the main features and principles of LESS. To find out how LESS is implemented in Magento, keep reading the tutorial.

7.2 Explain Magento's implementation of LESS (@magento_directive)

As it was mentioned before, Magento uses LESS. As a result, all the main style files have the .less extension.
Style files in Magento have the following structure (the path is specified as to the theme folder):

- style files for various modules are in the respective module folders: */<Namespace>_<Module>/web/css*
- main theme style files are in the folder: */web/css*

Let's consider the main theme files (which are located in the */web /css* theme folder):

- **styles-m.less** contains basic website styles and styles for mobile devices. The file contains other files (for example, *_styles.less*).
- **styles-l.less** contains styles for desktops. The file contains other files (for example, *_styles.less*).
- **_styles.less** is a compound file that contains other files with styles. According to Magento's file naming standard, the underscore ("_") in the file name means that the file is not used as a separate file, but is part of other files.
- **/source** is a folder with configuration files which call Magento UI library mixins (more information about UI library in you can find in the articles below).
- **/source/_theme.less** is a file in which new values are set for standard variables of the Magento UI library.

- **print.less** are styles for a printed version of a site's page.
- **/source/_variables.less** is a file in which custom variables are placed.

As a result, after compiling .less files into .css files, the theme gets the same name files: styles-m.css, styles-l.css, print.css. These files are on all pages of the site (these files may differ on various pages, since different style files can be connected on different pages).

In addition to these main files, other style files (third-party library files, custom style files, etc.) can be connected to the site.

These additional files are mainly included in the file:

```
<your_theme_dir>/Magento_Theme/layout/default_head_blocks.xml
```

For this, in the *default_head_blocks.xml* file in the <head /> tag you can add the <css /> or <link /> tag (the first is used for stylesheets and the second one to connect either style files or JavaScript files). Syntax of these tags is similar, so let's look at how to do this using the <css /> tag:

```
<css src="<path>/<file>" media="print|<option>"/>
```

Here
<path> is a path to a file as to *<your_theme_dir>/web*. If you want to add a link to the file located in the module folder in your theme, use *<Namespace>_<Module>::<path_to_file>* (for example, adding Magento_Theme :: the path will start as to *<your_theme_dir>Magento_Theme/web/*).
<file> is the name of a connected file with a .css extension.

media is an attribute used to specify additional parameters. For example, a file with the *media=”print”* parameter will be used for a printed version of the site page. A file with the *media="screen and (min-width: 768px)"* parameter will be used only for devices with screen resolution from 768 pixels.
src_type="url" is an additional attribute indicating that the file is being connected from another server and the “src” path will be set not as to the site, but absolutely (use this parameter connecting styles from other sites).

The example of the file connection process:

```
<head>
<link src="https://fonts.googleapis.com/css?family=Roboto:300,700"
src_type="url" />
<css src="Magento_Theme::css/source/lib/test1.css"  />
<css src="css/test2.css"  />
</head>
```

As a result, the following files will be added:

```
<link href="https://fonts.googleapis.com/css?family=Roboto:300,700">
<your_theme_dir>Magento_Theme/web/css/source/lib/test1.css
<your_theme_dir>/web/css/test2.css
```

Demonstrate the process from magento-less files via php preprocessing into real LESS files with extracted @import directives.

Loading the page, Magento searches for all the .css files declared in the <head/> tag. If the system does not find these files, it will search for the

same name files with the .less extension. If it does not find it in the current theme, it will search for the .css file in the parent theme. If it does not find it, it will search for the same name file with the .less extension. If it does not find either a .css file or a .less file in the farthest parent theme, there will be created a link to a nonexistent .css file.

.Css files in theme are generally used when you are not going to change them developing a theme (for example, files of third-party libraries). Files which will be changed during theme development are recommended to be used with .less extension. It allows using all LESS functionality.

In Magento, all styles are initially in .less files. These files are added to each other using the **@import** directives (we'll talk about them later).

As a result, from lots of .less files we get several final ones (the main ones are *styles-m.less* and *styles-l.less*). These final files are compiled into the same name .css files and uploaded to the site.

In Magento we can compile .less files into .css file in several ways:

- **Server-side Less compilation** is a server-side compilation with Less PHP library (https://github.com/oyejorge/less.php). It is a compilation mode by default. It's available in any site mode (default, developer, production).
- **Client-side Less compilation** is a client-side compilation with native less.js library (http://lesscss.org/usage/#using-less-in-the-browser). It's available only in default and developer site modes.
- **Compile LESS using Grunt** is a compilation using Grunt. More information on how to use Grunt for such purposes you can find here -https://devdocs.magento.com/guides/v2.3/frontend-dev-guide/css-topics/css_debug.html

Server-side Less or Client-side Less compilation mode you can choose in the admin following the path *“Store > Configuration > Advanced > Developer > Frontend Development Workflow”*, where pick a needed mode in *“Workflow type”* option.

As it was mentioned before, one .less files can be added to others with @import directives.

We described this process in LESS above. However, Magento has a special LESS directive - **@magento_import**. This directive allows adding multiple files from different places according to the name pattern (for example, to add .less files with the same name from different modules).

It has the following structure:

```
//@magento_import '<path>/<file_name>';
```

Here
<path> is a path to the file as to the file into which the directive is added.
<file_name> is a name of the included file. Technically, you don’t need to specify a file extension (Magento will add the .less extension automatically).But would be good to add the extension to the file name.
// - a double slash in front comments the directive allowing to avoid conflicts with the original LESS syntax.

During preprocessing of style files, the built-in LESS preprocessor first finds all **@magento_import** directives and then replaces them with a list of standard **@import** LESS directives. During compilation, they are replaced with styles of those files that they import.

Let's consider the example of using the **@magento_import** directive in *<Magento_Blank_theme_folder>/web/css/styles-m.less* file:

```
//@magento_import 'source/_module.less';
```

As a result, after preliminary processing of files by the built-in LESS preprocessor, this directive will be replaced by the following:

```
...
@import '../Magento_Catalog/css/source/_module.less';
@import '../Magento_Cms/css/source/_module.less';
@import '../Magento_Customer/css/source/_module.less';
...
```

This list will be huge since the preprocessor goes through all the modules and adds **@import** directives for all module files at the given path and name in the **@magento_import** directive.

It's very convenient and helps to save time.

Where can the intermediate files be found?

During compilation, all intermediate files get to the ***var/view_preprocessed*** directory. In these files, all special **@magento_import** directives are replaced by the standard **@import**.

Then, all .less files from the ***var/view_preprocessed*** directory will be compiled into .css files.

What do you have to remember, when you change a less file? Which files will be re-processed on file changes?

Changing .less file, remember that after that you need to compile this file and files which are included directly or through several occurrences (when a file is included into another file which is also in another one - you need to recompile all these files). The same should be done when you create a new one, rename or delete an existing .less file. You need to recompile this file and all the files in which it's included.
If you use **Server-side Less compilation**, having changed the .less file, we recommend to do the following:

1. Clear the folder with static files - *"pub/static/frontend/<Vendor>/<theme>/<locale>"*, *"var/cache"* - and a folder with intermediate files - *"var/view_preprocessed"*.
2. Starc compilation and publication of static files. It can be done, having reloaded the site's page. But we recommend using **static files deployment tool**

   (https://devdocs.magento.com/guides/v2.3/config-guide/cli/config-cli-subcommands-static-view.html) since in the first case there will be compilation and publication only of those files that are used on the reloaded page. Besides, you won't get information on errors.

If you use **Client-side Less compilation** changing the majority of files, the changes will be applied immediately without cleaning the static files folders after the page is reloaded. It is necessary to clean the static files folders (*pub/static/frontend/<Vendor>/<theme>/<locale>*) when you've changed the file with **@import** and **@magento_import** directives or when you delete / rename files which you've included to other files.

Are the original files copied or symlinked in developer environments?

Symlinks files are symbolic links to files (like Windows shortcuts) i.e. when a system uses a symlink file, it opens a file to which it refers by a link. A symlink file doesn't take the place what is very convenient. It doesn't clutter a server with additional copies of files.

Depending on the compilation mode (client or server-side) you choose, the original files can be copied or there will be created symlinks to them. Let's consider the process in detail - here will be described the system with disabled cache.

When we use **Server-side Less compilation**, all our .less files (including those which will be compiled into .css files and those included in other .less files with **@import** and **@magento_import** directives) are firstly copied to the *var/view_preprocessed* directory where these files are compiled into the resulting .css files located the same folders. Then, the resulting .css files are copied to the *pub/static/frontend/<Vendor>/<theme>/<locale>* folder. As a result, original .less files are in the theme folder and their copy (processed by the processor) in the *var/view_preprocessed* folder. The resulting .css files are located in the *var/view_preprocessed* folder and their copy is in the public folder - *pub/static/frontend/<Vendor>/<theme>/<locale>* - from which they are available on the site.

With **Client-side Less compilation** the process is slightly different. The resulting files will differ from those in **Server-side Less compilation**, since the compilation takes place on the client-side and the resulting styles are added to the site during the page loading. If we look at above mentioned folders, we see that in the *var/view_preprocessed* folder are created the same files as in **Server-side Less compilation**. Except that the resulting

files will be the same as those original ones processed by the processor (for example, the styles-l.css file will be identical to the styles-l.less file in the same folder with the same import directives). Then these resulting files will be copied to the *pub/static/frontend/<Vendor>/<theme>/<locale>* folder and created symlinks to the files that are imported into the resulting files by @import directives

The exception for **Client-side** and **Server-side Less compilation** are .css files located in our theme folder and are not imported into other styles files. In this case, such .css files are located only in our theme folder, while symlinks to these files are in the *pub/static/frontend/<Vendor>/<theme>/<locale>* folder. So, changing our .css files in theme, we don't need to to clean any folders - the changes will immediately be displayed on the site.

7.3 Describe the purpose of _module.less, _extend.less, _extends.less

Demonstrate LESS has no fallback capabilities and therefor magento created @magento_import directives to enable FE devs to inject or replace parts of existing less structures of modules and themes.

As it was mentioned before, there is a special LESS directive in Magento - **@magento_import**.

This directive allows adding several files from different places for the name template what expands the functionality of the standard **@import** directive (read more about it in the previous article - 7.2).

Let's consider the main files that are added by **@magento_import** directive in the main style files - - styles-l.less, styles-m.less. and understand what they are used for:

_module.less file contains the main styles for the module where it is located. In this file, using **@import** directives, other additional style files of this module are connected. As a rule, we need to create this file in theme in case we want to change the styles of the module significantly.

_widgets.less contains the basic styles for the module widgets where it is located. This file is connected after the **_module.less** file and therefore the styles from it have a higher priority than the last.

_extend.less contains additional styles for the module in which it is located. As a rule, we create this file to make minor changes in the module or add styles for new elements that have not been stylized in the current module before. When we do not want to rewrite existing module styles, it's better to create such a file. This file is connected after the **_module.less** and **_widgets.less** files. So, the styles from it have a higher priority than in them.

_extends.less contains lots of abstract selectors. During development, these selectors can be used through mixins and extensions. Unlike the files above, this file is connected using the ***@import (reference) 'source/_extends.less'***.

module.less, **_widgets.less** and **_extend.less** files are located in modules folders along the following path:

<Module_Folder>/web/css/source/

_extends.less file is in the theme folder as follows:
<Theme_Folder>/web/css/source/_extends.less

Unlike layout files, .less files do not complement, but replace the parent theme files. In other words, if you create the **_module.less** or **_extend.les**s file in your theme, it will overwrite the parent theme's **_module.less** or **_extend.less** file and the styles from the parent theme file will not be applied. If you want to add something to a style file of a parent theme, you need to copy the content of the file into the same name one of your theme and make changes there.

7.4 Show configuration and usage of CSS merging and minification

Demonstrate the primary use case for merging and minification

CSS merging is a process when multiple css files merge into one common CSS file. The main goal of merging is to reduce the number of HTTP requests. Instead of sending many requests to each file individually, there will be only one request to the resulting file. More about the file merging process you can find in the "Understand the implications merging has in respect to folder traversal" section.
CSS minification is a process of the final CSS file reduction by removing extra spaces and duplicate styles. The main goal of the minification is to reduce the weight of the final CSS file.
CSS merging and minification allow adding site pages faster and as a result increase site performance.

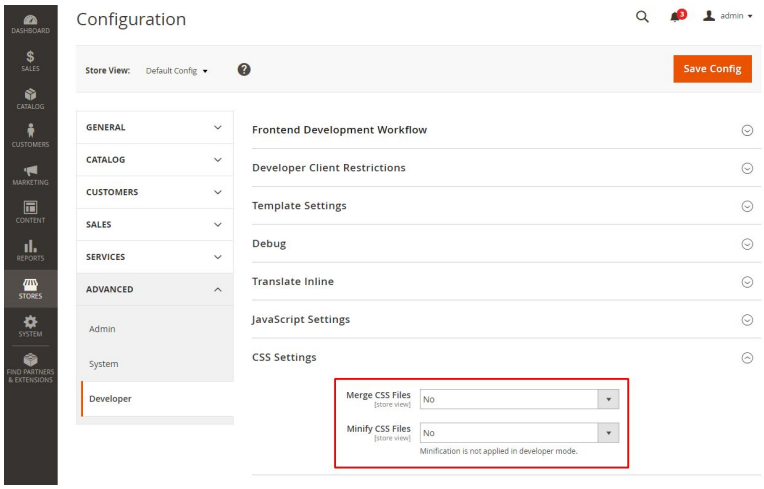
Magento 2 has built-in functionality responsible for CSS merging and minification. Keep reading to know how to turn these options.
If you're not satisfied with built-in Magento functionality and need more flexibility working with files, you can use task runners like Grunt and Gulp.

Determine how these options can be found in the backend

You can enable or disable CSS merging and minification in the admin panel only in Default or Developer mode.
For this, in the admin of the site we go to

Store > Configuration > Advanced > Developer > CSS Settings

Here we enable or disable CSS merging and CSS minification

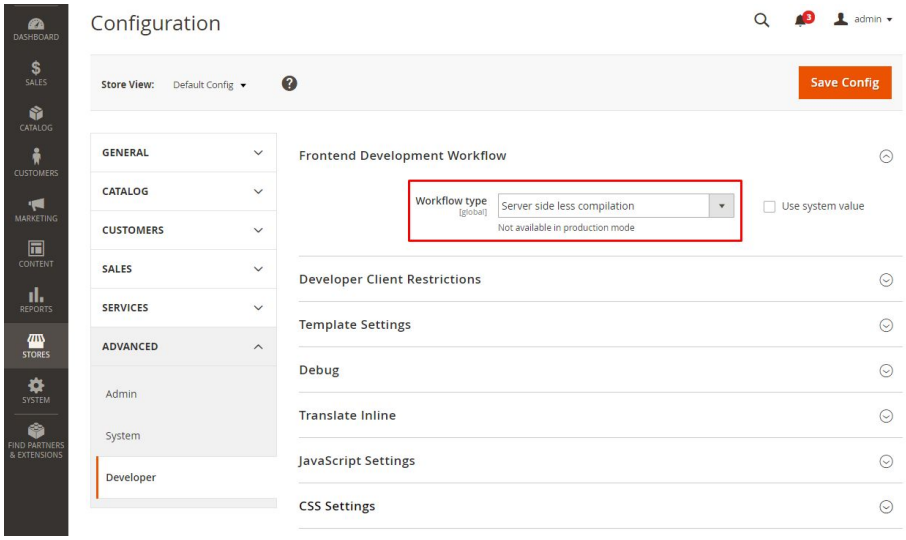


Then you need to clean the site cache.
It should be noted that CSS minification won't work in the Developer mode.

Pay attention that CSS merging and minification can only be enabled in case of the server-side compilation (with the client-side compilation, there

will be an error on the site's workflow). To run server-side compilation, you need to go to
In order to enable compilation on the server side, go to
Store > Configuration > Advanced > Developer > Frontend Development Workflow

Here we choose the "Server side less compilation" value for the "Workflow type" parameter.



In the Production mode, CSS merging and minification is automatic with no need to install the parameters above (there is no such parameters in the admin in the Production mode).

Understand the implications merging has in respect to folder traversal

As it was mentioned in previous articles, in the Blank default theme are used 3 main style files which apply on all pages of the sit (there are also added style files that may differ for for different pages of the site):

styles-m.css contains common styles of the site and styles for mobile devices
styles-l.css contains styles for devices with a screen width from 768 pixels
print.css is pages' styles for printed version

These files are generated from the same name less-files - styles-m.less, styles-l.less and print.less - which are gathered from many other files.
We want to pay attention to 2 files - styles-m.css and styles-l.css. Magento 2 follows the "mobile first" principle. So, the styles-m.css file is connected first, then the styles-l.css. The last one is available only on devices with a screen width from 768 pixels.

There are two ways to add our custom styles to our theme.

In our topic, we can add our own custom styles in two ways.

Method 1. Place custom styles in files that are included in the resulting styles-m.less and styles-l.less files, and as a result in styles-m.css and styles-l.css files (for this, you can supplement or overwrite the default theme style files. It was described in the article above.

Method 2. Add your .less file and plug the same name resulting file in layout. For this, in the file
<your_theme_dir>/Magento_Theme/layout/default_head_blocks.xml

In the <head> tag add the following instruction

```
...
<head>
...
     <css src="css/your_custom_style_file.css" />
...
```

```
</head>
...
```

And please the same name less file by the adress *<your_theme_dir>/Magento_Theme/web/css/your_custom_style_file.less*

As a result, your ***your_custom_style_file.css*** file will be at the site in the <head> tag after the standard styles-m.css but before the standard styles-l.css.

To let your styles connect after the styles-l.css file and be used for devices with screen width from 768 pixels, in file connection you need to specify the parameter - *media="screen and (min-width: 768px)"* - (you can specify any minimum width - the file will still be attached after styles-l.css). As a result, the connection of such a file will look as follows:

```
...
<head>
...
     <css src="css/your_custom_style_file.css" media="screen and
(min-width: 768px)" />
...
</head>
...
```

Enabling the CSS merging option, the files are combined as follows. Plug-in files with no additional parameters (such as *media="screen and (min-width: 768px)"*) are combined with the standard styles-m.css file and added after the styles of this file in the sequence in which they are connected. Files with additional parameters are combined with those that have the same additional parameters, i.e. all files which additional parameters coincide (for example, *media="screen and (min-width: 1024px)"*)

will be combined into one file with such a parameter in the sequence in which they were connected.
To merge the plug-in file with the standard styles-l.css, the plug-in file needs to have the *media="screen and (min-width: 768px)"* parameter.
Monitor this attaching additional files.

CSS files merging is individual for each page. Since we can plug different style files on each page, the merged files will differ for different pages of the site.

There might be conflicts during file merging. For example, if there is an error in your file, when your file with the error and the files following it are merged, the styles in the resulting file might be broken after the error.

Remember that CSS merging and minification should be enabled after the site stylization or this will interfere with the process.

7.5 Magento UI library usage

Demonstrate your understanding of magento's UI library, a LESS-based library of mixins and variables for many different standard design elements on website

Magento 2 has a useful thing - UI library. It uses the LESS processor and consists of a set of LESS files. These files use a set of mixins and variables to style the main site's elements (such as buttons, forms, navigation and others. The full list of elements available to style you can find here- https://magento-devdocs.github.io/magento2-ui-library/).

The UI library simplifies the theme creation and customization for a developer. In this article, we provide more information about the library and describe how to use it in your theme.

How can you take advantage of the UI library?

The UI library, as it was mentioned before, contains a set of variables and mixins that you can use in your theme to speed up the development process. Let's view some examples on how it can be used. I suggest looking at practical examples of how this can be used.

For example, you have a button with "example-button" class and you need to make it look standard. For that, we can use the **.lib-button()** mixin what looks as follows:

```
.example-button {
      .lib-button();
}
```

You won't deny that it's more convenient than copying all styles for all button states manually.
If you want the button to look like standard but with slight differences, you can use the same mixin but with changed parameters. For example, the button's text color is to be red and 25 pixels internal indent on each side. So, it has the following structure:

```
.example-button {
      .lib-button(
@_button-color: red,
@_button-padding: 25px
);
}
```

The **.lib-button()** mixin has lots of parameters. The full list you can view in the file with mixins for buttons here:

```
lib/web/css/source/lib/_buttons.less
```

Note that variables that begin with @_ are internal mixin variables and are only available inside it. Variables that begin with @ are global and available anywhere.

These were examples of using the UI library mixin.

Let's see how to use variables of the UI library.

For example, you need to make the text size in the block with the "example-block" class in the same size as the text in the buttons. So, use the variable **@button__font-size**

It looks as follows:

```
.example-block {
      font-size: @button__font-size;
}
```

If the **@button__font-size** is changed, the text size in the "example-block" also changes (it's very convenient in some cases).

How to change the values of variables read in this article below.

What do you have to do to enable it in your theme?

If you create your theme based on the standard Luma or Blank (specify this theme as the parent), the UI library will automatically be available in files that complement the standard LESS files of the parent theme (you can read more about LESS in Magento 2 here - https://belvg.com/blog/less-in-magento-2-0.html).

If there is no standard theme or you use your own independent LESS files, to use mixins and variables from the Magento UI library, you need to add a link to this library. For this, add the following import in your LESS file

@import 'source/lib/_lib';

Which file is primary used for basic setup of variables?

If you need to change the variables value in the UI library, use the **_theme.less** file, which should be located in your theme folder along this path:

```
app/design/frontend/Our_Vendor/Our_Theme/web/css/source/_th
eme.less
```

Note that this file will overwrite the parent theme file of the same name. So, if you need to save the contents of the **_theme.less** file of the parent theme, copy it to your file.

If you want to create your variables, we recommend using the **_variables.less** file, ehic is here:

```
app/design/frontend/Our_Vendor/Our_Theme/web/css/source/_va
```

```
riables.less
```

Where can UI library files be found?

The UI library files you can find at the following address:

```
lib/web/css/source/lib
```

There are files with mixins at the core of this folder. The “variables” subfolder contains variables.
The official UI library documentation you can find here -

https://magento-devdocs.github.io/magento2-ui-library/

There are all available to style elements and the library structure. The document describes the rules for naming files, variables, mixins and code standards in LESS files.

How can it be extended?

To customize and complement existing mixins, you can copy LESS files with them to your theme. For example, to redefine a file with mixins for buttons:

```
lib/web/css/source/lib/_buttons.less
```

You need to copy the file by the address

```
app/design/frontend/Our_Vendor/Our_Theme/web/css/source/lib
```

```
/_buttons.less
```

And here already make the necessary changes and additions.

The same can be done with variables files (but it’s easier to change the variables value in the _theme.less file, as it was mentioned before).

How can you change specific parts of the UI library?

Let’s sum up how to change and use specific parts of the UI library.

We can use mixins, changing the variable value of a mixin in the usage.

There are variables whose values are used in mixins and common style files. We can change the values of these variables in the _theme.less file in our theme. Changing the variables values, we can significantly customize the appearance of the site theme.

If we need to change entire files with a set of variables or mixins, we can copy these files to our theme where we will make global changes.

And finally, if there are not enough built-in variables and mixins of the Magento UI library, we can create our own.

Section 8: Customize the Look and Feel of Specific Magento Pages

8.1 Utilize generic page elements

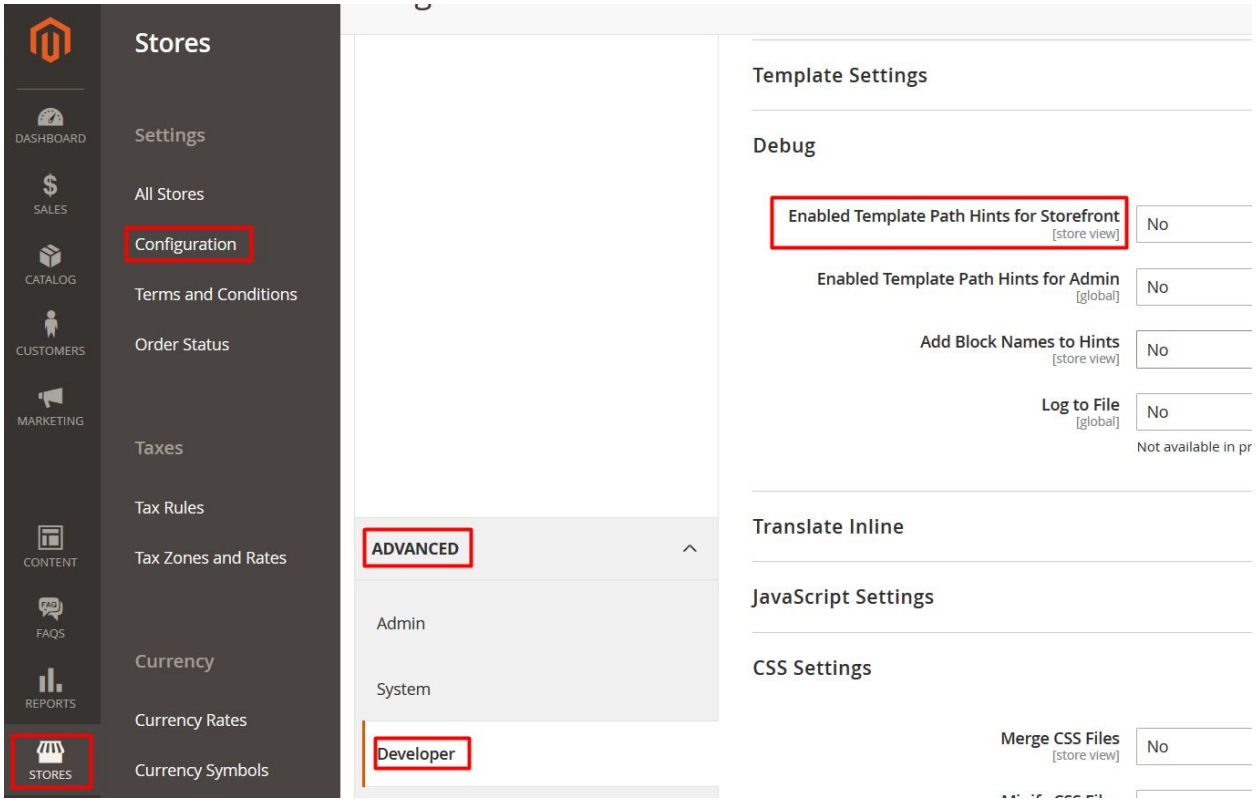
Demonstrate an understanding of customizing generic page elements that can be found on most pages: page header and footer, quick search, store view (language) switcher, mini cart, breadcrumbs, and sidebar menu.

Let's pay attention to the set of tasks you need to perform the most, if you create your theme in Magento - elements customization (page header, footer, search, language switch, mini carts, breadcrumbs and sidebar menu).

We will customize all these elements by overriding the php, phtml and xml templates of the stock element. For this, you need to copy the necessary file into your theme and make the necessary changes there.

If you don’t know where the necessary template is - enable hints

Stores -> Configuration -> Developer -> Advanced -> Debug -> Enabled Template Path Hints for Storefront (Yes)



Below you can find the paths to the main most customizable files from the blocks we need (you just need to override it in your theme).

If you need to write additional styles for an element, create a file by the path (/Vendor/Theme/Magento_Theme/web/css/source/_extend.less).
If you prefer scss/sass, you can create a structure convenient to you:

Page header

- Header block (/vendor/magento/module-theme/Block/Html/Header.php)

- Header links (/vendor/magento/module-theme/view/frontend/templates/html/header.phtml)
- Logo (/vendor/magento/module-theme/view/frontend/templates/html/header/logo.phtml)
- Menu bar (/vendor/magento/module-theme/view/frontend/templates/html/topmenu.phtml)
- Customer details UI component (/vendor/magento/module-customer/view/frontend/web/js/view/customer.js)
- Layout XML (/vendor/magento/module-theme/view/frontend/layout/default.xml)

Page footer

- Footer container (/vendor/magento/module-theme/view/frontend/templates/html/footer.phtml)
- Footer block (/vendor/magento/module-theme/Block/Html/Footer.php)
- Copyright (/vendor/magento/module-theme/view/frontend/templates/html/copyright.phtml)

Quick search

- Mini-search form (/vendor/magento/module-search/view/frontend/templates/form.mini.phtml)

Store view switcher

- Block (/vendor/magento/module-store/view/frontend/templates/switch/languages.phtml)

Mini-cart

- Minicart form (/vendor/magento/module-checkout/view/frontend/templates/cart/minicart.phtml)
- Layout XML (/vendor/magento/module-checkout/view/frontend/layout/default.xml)
- UI Component (/vendor/magento/module-checkout/view/frontend/web/js/view/minicart.js)

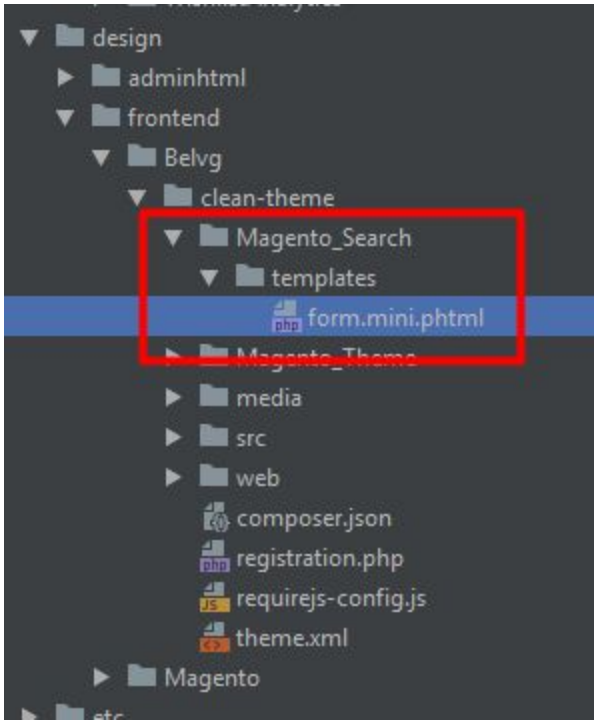
Breadcrumbs

- Template (/vendor/magento/module-theme/view/frontend/templates/html/breadcrumbs.phtml)
- Block (/vendor/magento/module-theme/Block/Html/Breadcrumbs.php)

Sidebar menu

- sidebar.main and sidebar.additional are names of containers into which you need to put the needed blocks through xml.

As an example, we create a voice search button in the search bar.
We create the Magento_search directory in the theme following the structure:



In the form.mini.phtml file we add the necessary element:

```
...
<div class="control">
   <button class="action primary" id="voice-search-trigger"><?= /*
@escapeNotVerified */ __('Voice Search') ?></button>
   <input id="search"
          data-mage-init='{"quickSearch":{
               "formSelector":"#search_mini_form",
               "url":"<?= /* @escapeNotVerified */
$helper->getSuggestUrl()?>",
               "destinationSelector":"#search_autocomplete"}
               }'
          type="text"
          name="<?= /* @escapeNotVerified */
$helper->getQueryParamName() ?>"
          value="<?= /* @escapeNotVerified */
$helper->getEscapedQueryText() ?>"
          placeholder="<?= /* @escapeNotVerified */ __('Search entire
store here...') ?>"
          class="input-text"
```

```
            maxlength="<?= /* @escapeNotVerified */
$helper->getMaxQueryLength() ?>"
            role="combobox"
            aria-haspopup="false"
            aria-autocomplete="both"
            autocomplete="off"/>
    <div id="search_autocomplete" class="search-autocomplete"></div>
    <?= $block->getChildHtml() ?>
</div>
...
```

As a result, we get a new element that we can style and more.

8.2 Customizing product detail page

How can design changes (page layout) be configured on product detail pages?

To change the design (page layout) of product detail pages you need to create a folder structure in a child theme on this path:

```
magento_root_flder/app/design/frontend/company_name/theme_name/Magento_Catalog/layout
```

Then copy the xml file

```
magento_root/vendor/magento/module-catalog/view/frontend/layout/catalog_product_view.xml
```

And add it to the path:

```
magento_root_flder/app/design/frontend/company_name/theme_name/Magento_Catalog/layout
```

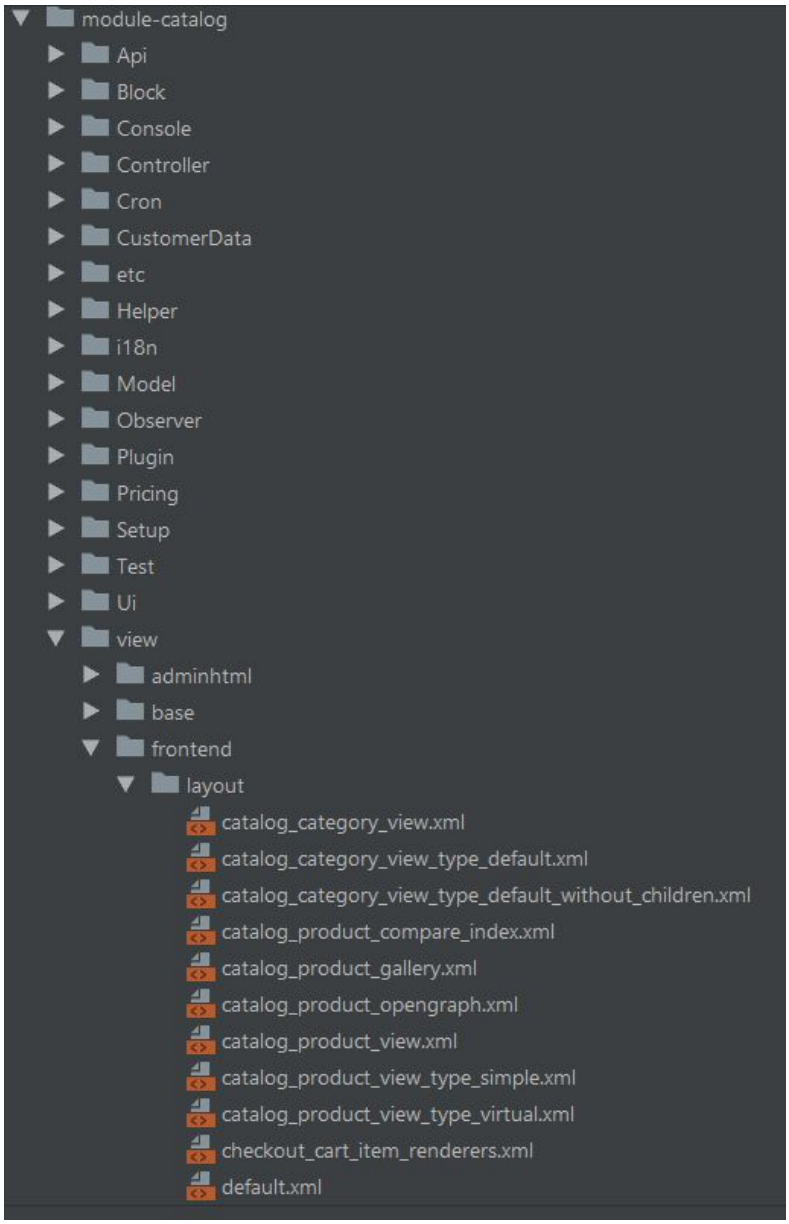
Then the newly copied file can be configured directly. Changes will affect all product detail pages.

How can design changes be configured for specific product types?

On the path:

```
magento_root/vendor/magento/module-catalog/view/frontend/layout/
```

Here are layout files for different product types on Magento 2



- catalog_product_view_type_simple.xml
- catalog_product_view_type_configurable.xml
- catalog_product_view_type_grouped.xml
- catalog_product_view_type_bundle.xml
- catalog_product_view_type_virtual.xml
- catalog_product_view_type_downloadable.xml

You need to override the needed file having copied it on the path:

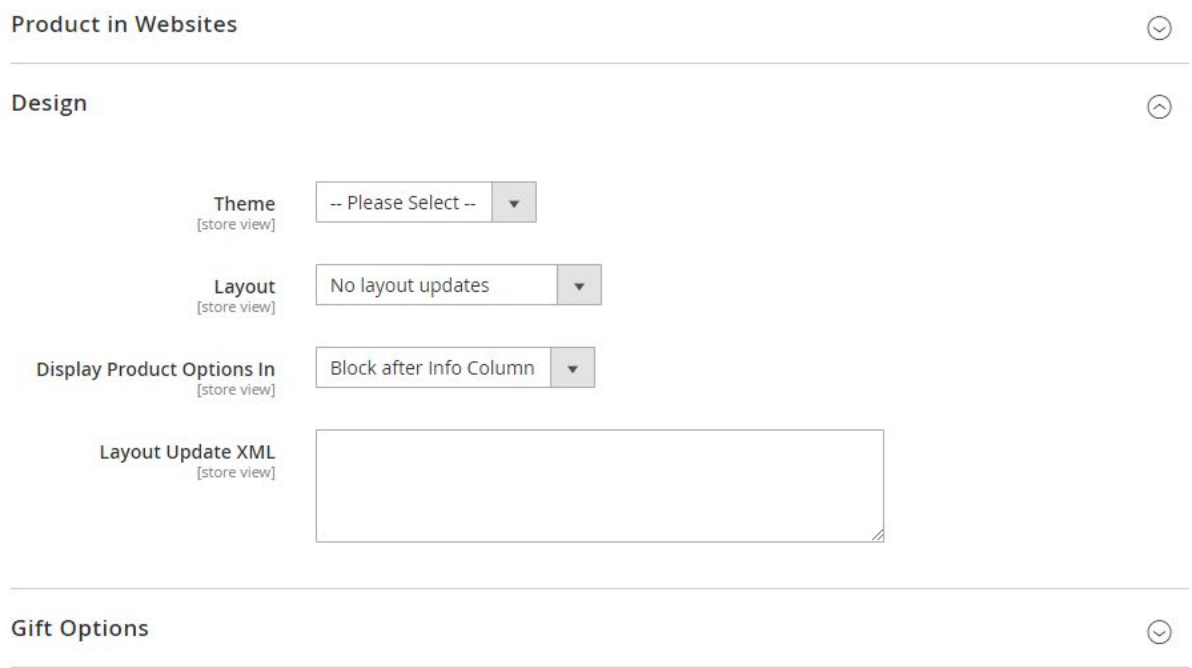
```
magento_root_flder/app/design/frontend/company_name/theme_name/Magento_Catalog/layout
```

Then we can make changes to be applied to the particular product type.

How can you use custom layout updates for specific product pages?

There are several ways to use custom layout update for a specified product.

In the first case we need to enter admin panel on the path:
Catalog > Products > *choose the product*
Then on the Design tab, you can expand the layout:



Changes apply to a specific product.

The second way is to edit `magento_root_flder/app/design/frontend/company_name/theme_name/Magento_Catalog/layout/catalog_product_view.xml`

Adding tap with the products id
`<catalog_product_view_id_productid>`

Or using product sku
`<catalog_product_view_sku_productsku>`

Demonstrate an understanding of how to use the container blocks provided by Magento to display additional information on category pages.

To display additional information on category pages using container blocks (for example, display a block in the sidebar product catalog, which is edited in adminpanel), you need to follow the following below:

Copy the following file to the child theme:
magento_root_flder/vendor/magento/module-catalog/view/frontend/layout/catalog_category_view.xml
On the path:
magento_root_flder/app/design/frontend/company_name/theme_name/Magento_Catalog/layout/catalog_category_view.xml

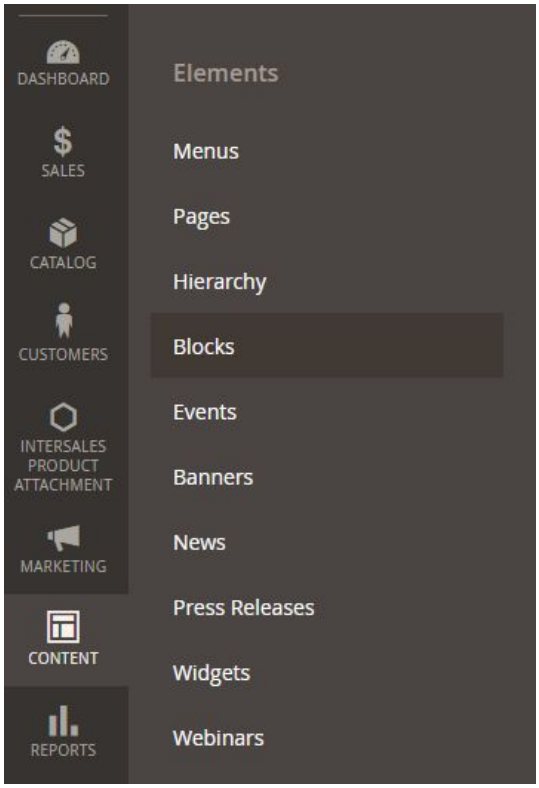
Make the following changes:

```
<referenceContainer name="columns">
   <container name="catalog.page.sidebar" as="catalog_page_sidebar"
htmlClass="sidebar sidebar-block" htmlTag="div"
before="sidebar_additional" after="div.sidebar.main">
```

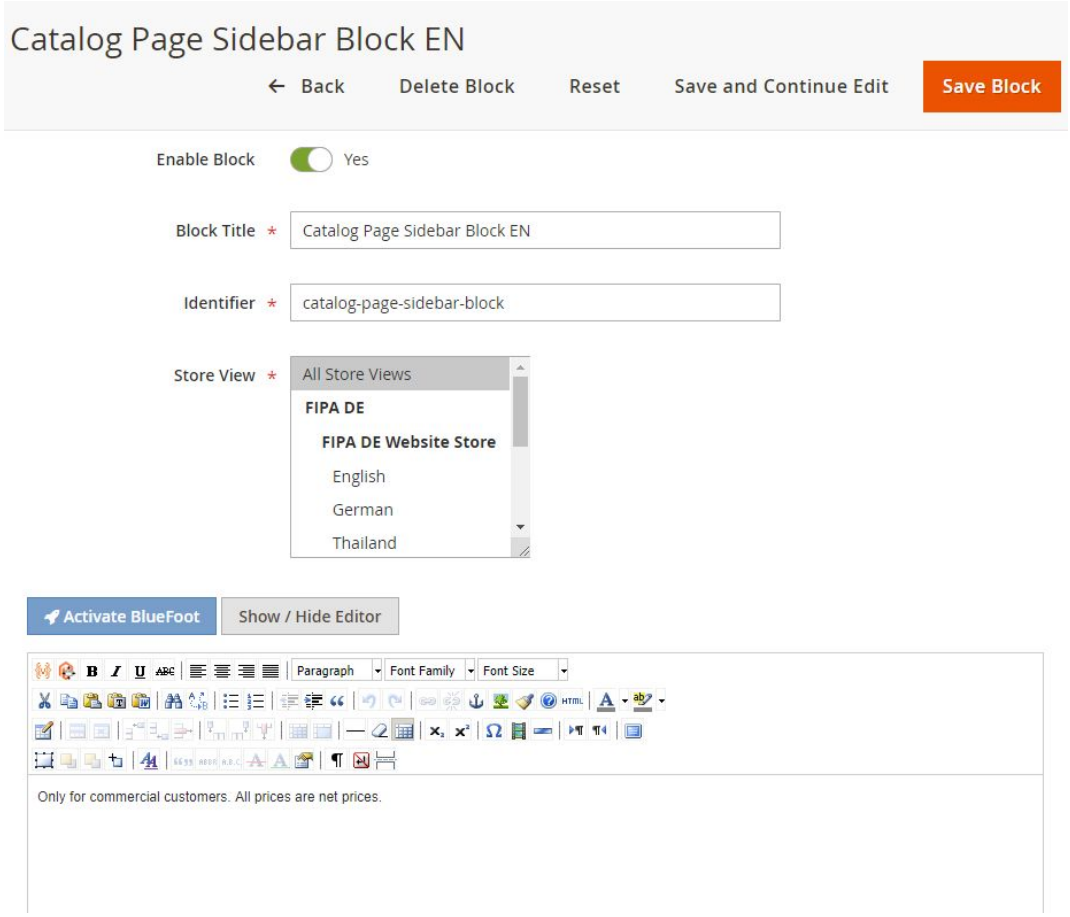
```
        <block class="Magento\Cms\Block\Block"
name="catalog-page-sidebar-block">
            <arguments>
                <argument name="block_id"
xsi:type="string">catalog-page-sidebar-block</argument>
            </arguments>
        </block>
    </container>
</referenceContainer>
```

The names of the properties of the container “name”, “as”, “htmlClass” are indicated in accordance with the content (come up with your own, which is more suitable)
We insert a block into the container, indicating its name and passing block_id to arguments.

Then we create a block in admin panel (Content > blocks > Create block)



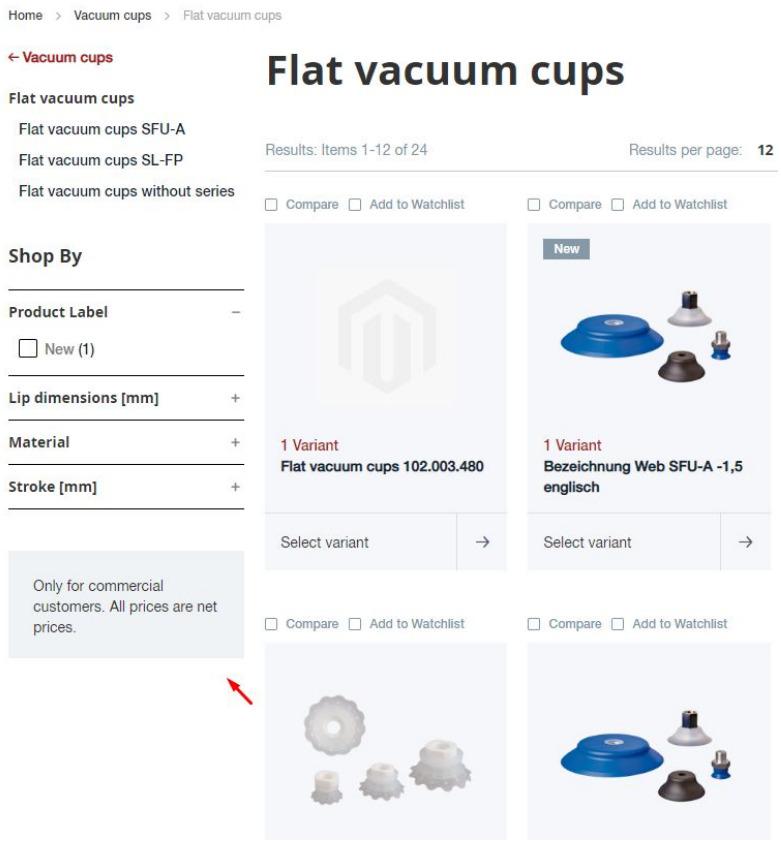
Where we specify the needed content



identifier is block_id, specified in the xml file:

```
<arguments>
    <argument name="block_id" xsi:type="string">catalog-page-sidebar-block</argument>
</arguments>
```

It will look as follows:




```
▼<div class="page-wrapper">
  ▶<header class="page-header">…</header>
  ▶<div class="breadcrumbs">…</div>
  ▼<main id="maincontent" class="page-main">
     <a id="contentarea" tabindex="-1"></a>
    ▶<div class="page messages">…</div>
    ▼<div class="columns">
      ▶<div class="column main">…</div>
      ▶<div class="sidebar sidebar-main">…</div>
       <div class="sidebar sidebar-block" style="clear: none;">Only for commercial
       customers. All prices are net prices.</div> == $0
      ▶<div class="sidebar sidebar-additional">…</div>
       ::after
     </div>
   </main>
  ▶<div class="page-bottom">…</div>
```

There is a way to output phtml block in the container. It is necessary to repeat the previous steps, except for the inserted code:

```
<referenceContainer name="columns">
   <container name="catalog.page.sidebar" as="catalog_page_sidebar"
htmlClass="sidebar sidebar-block" htmlTag="div" before="sidebar_additional"
after="div.sidebar.main">
       <block class="Magento\Framework\View\Element\Template"
name="custom-content"
template="Magento_Catalog::/category/custom-content.phtml" />
   </container>
</referenceContainer>
```

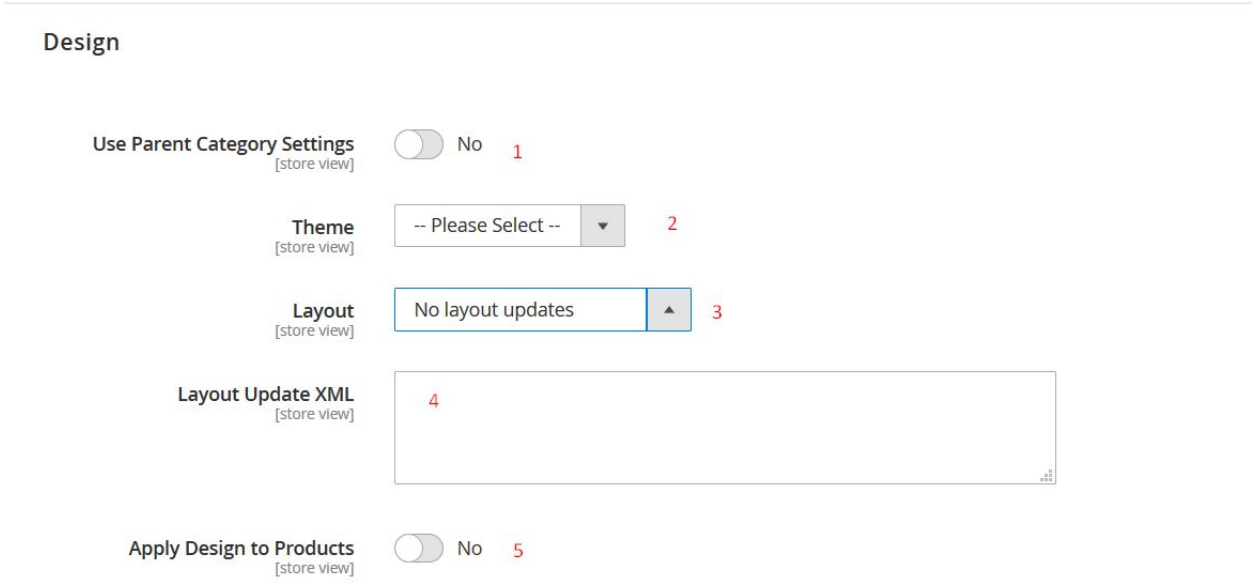
Add the needed content to the *custom-content.phtml* file.
There is no need to make changes in the admin panel.

8.3 Customizing category pages

How can design changes (page layout) be configured on category pages?
How can the layered navigation be configured?

The category page appearance can be configured in two places - in the design section and the display settings. Let’s consider both ways:

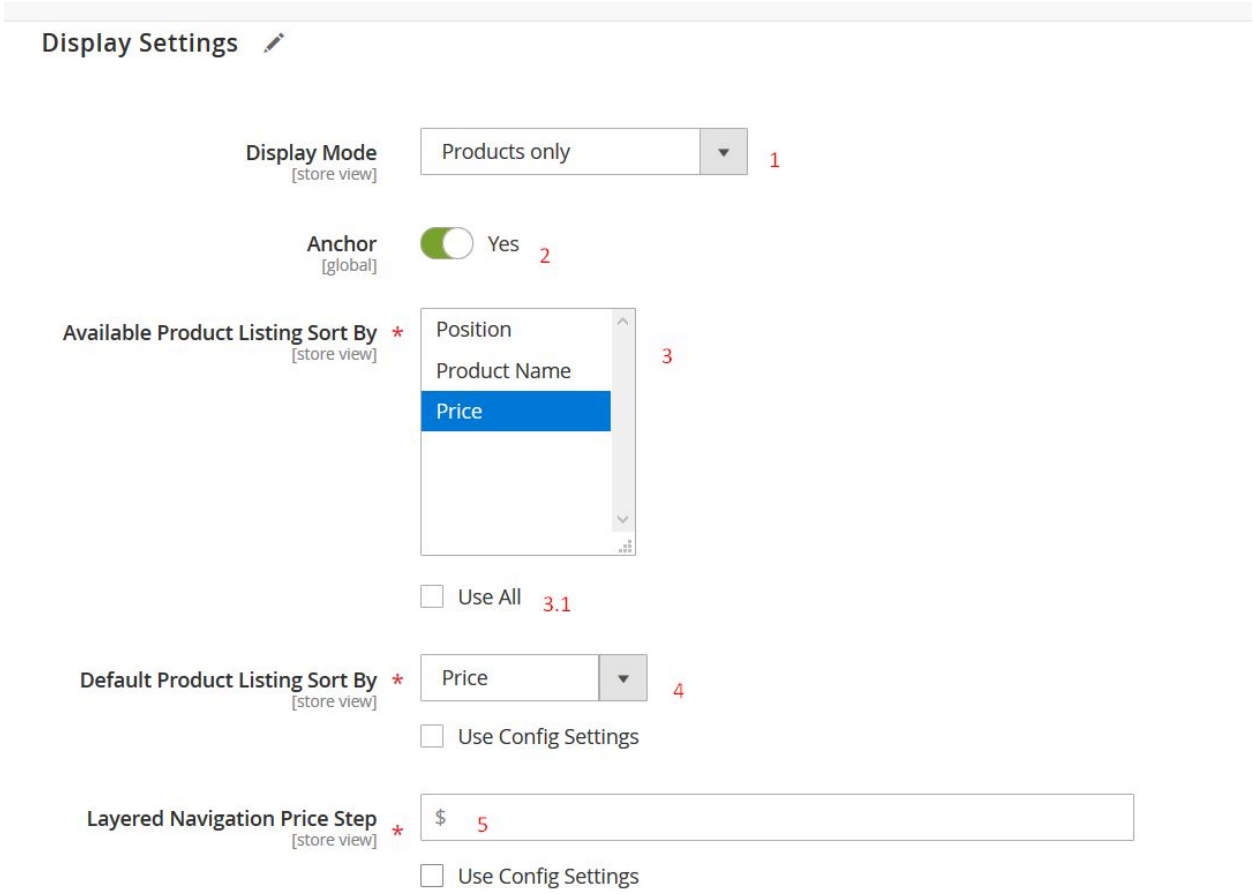
Design



1. If “Use parent category settings” is applied, the other settings are inactive (more details below).
2. Theme. Choose a theme that is different from the default one for this store.
3. Layout. Choose one of the described in the page_layouts.xml file.
4. Layout update. If you need to remove/move/add an item of a category, you can insert the code here using
   <referenceBlock name="block_name" remove="true" />
   <move element="block_name" destination="block_name" />
   <block class="" name="block_name">
     <arguments>
       <argument>your_argument</argument>
     </arguments>
   </block>

5. “Apply design to products” allows applying theme, layout and layout update settings with products in this category.

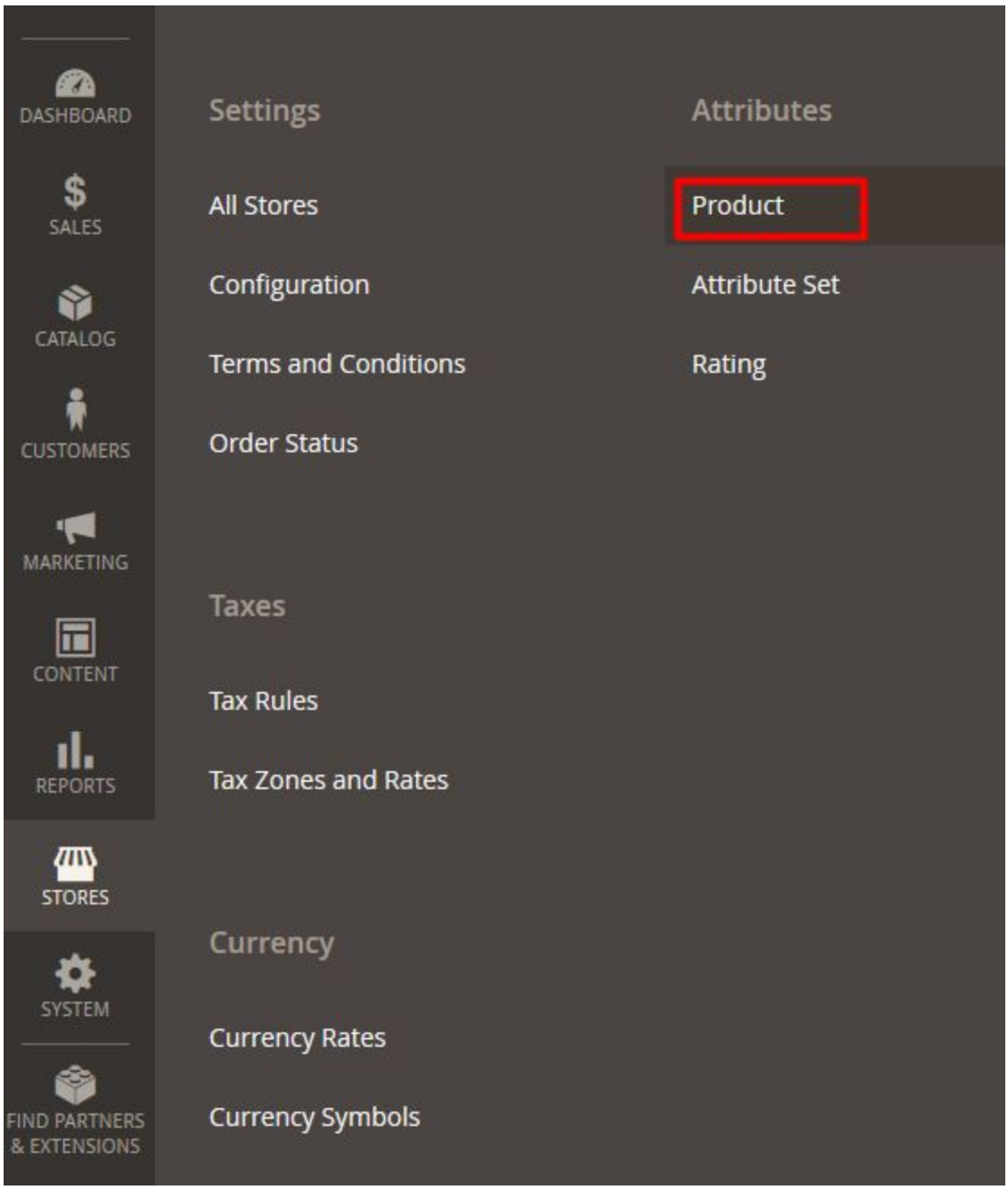
Display Settings:



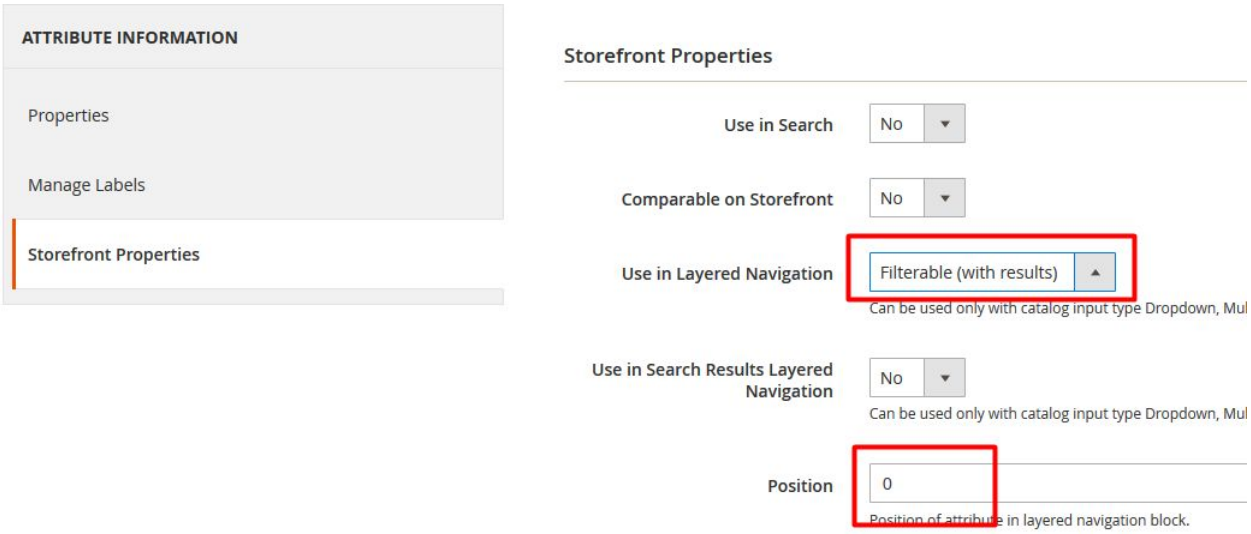
1. Display mode. Choose between “Products Only”, “Products and Static Block” and Static Block Only”.
2. Anchor. When enabled, all child products from child categories will be displayed in the category.
3. Sort By. If you uncheck “all” (3.1), you can choose the parameter to sort products into categories.
4. Default Products Listing Sort By parameter is here.
5. Step Settings to filter price.

To set up layered navigation, you can change the attributes’ order in the admin on the path:

Stores > Attribute > Product:



Choose “use for the layered navigation” to make the attribute work like a filter and set the order to it (all attributes have a value of 0 by default).

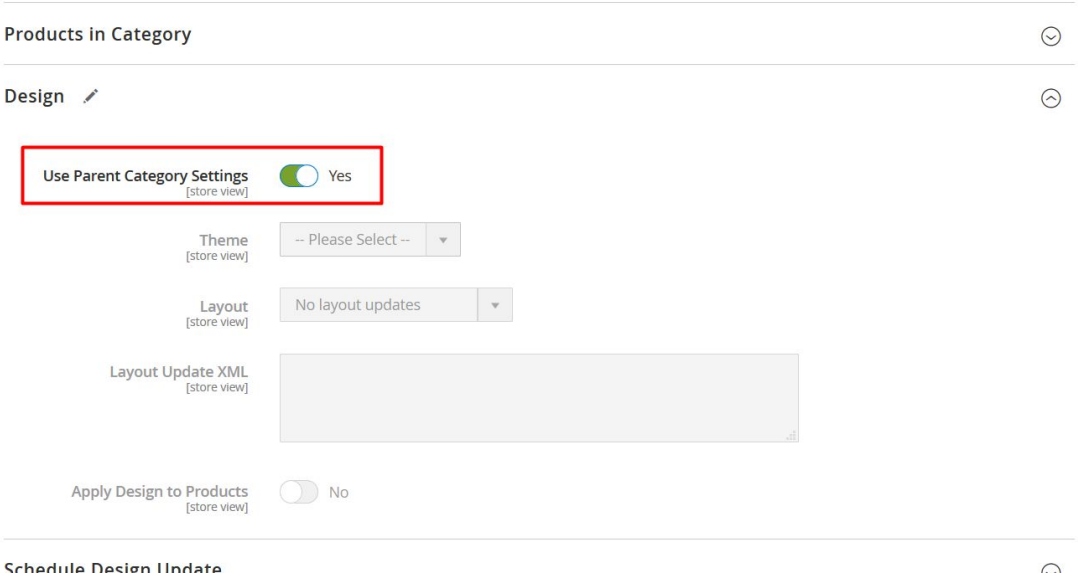


You also can change the navigation block by overdriving files

```
/vendor/magento/module-layered-navigation/view/frontend/layout
/vendor/magento/module-layered-navigation/view/frontend/templates
```

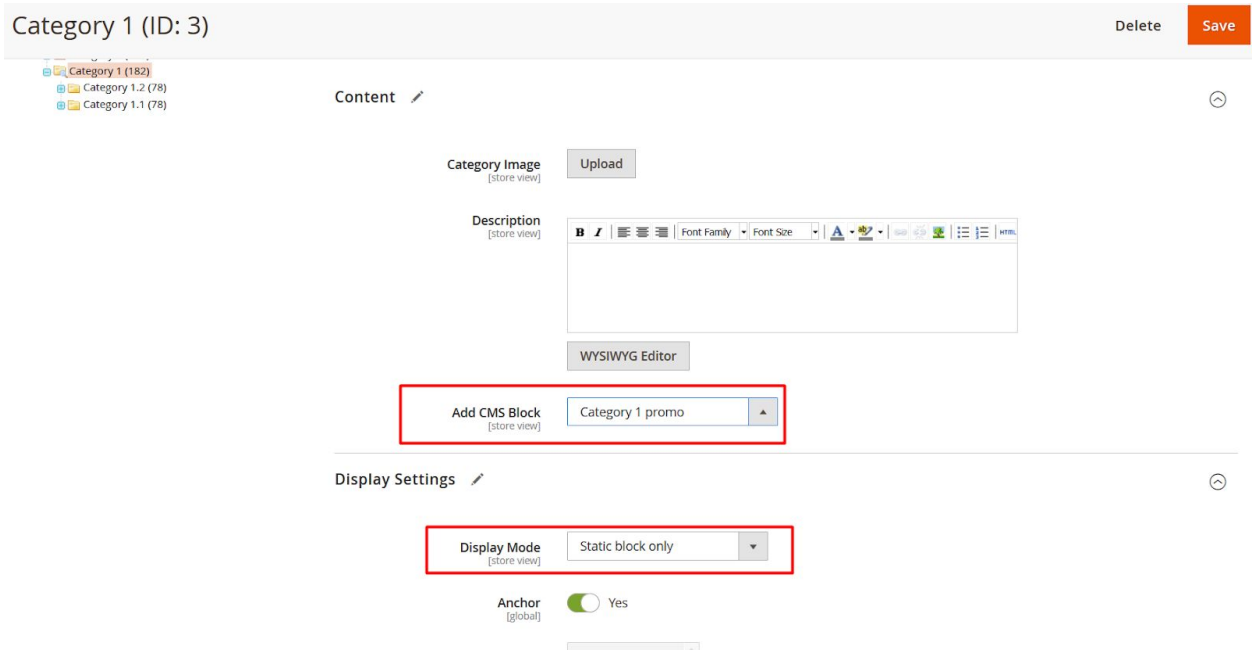
Demonstrate an understanding of configuring design inheritance for category pages.

Here we need to choose “Use Parent Category Setting”. As you see, the other settings are inactive since they are inherited from the higher category:



How can a CMS block be configured as a category landing page?

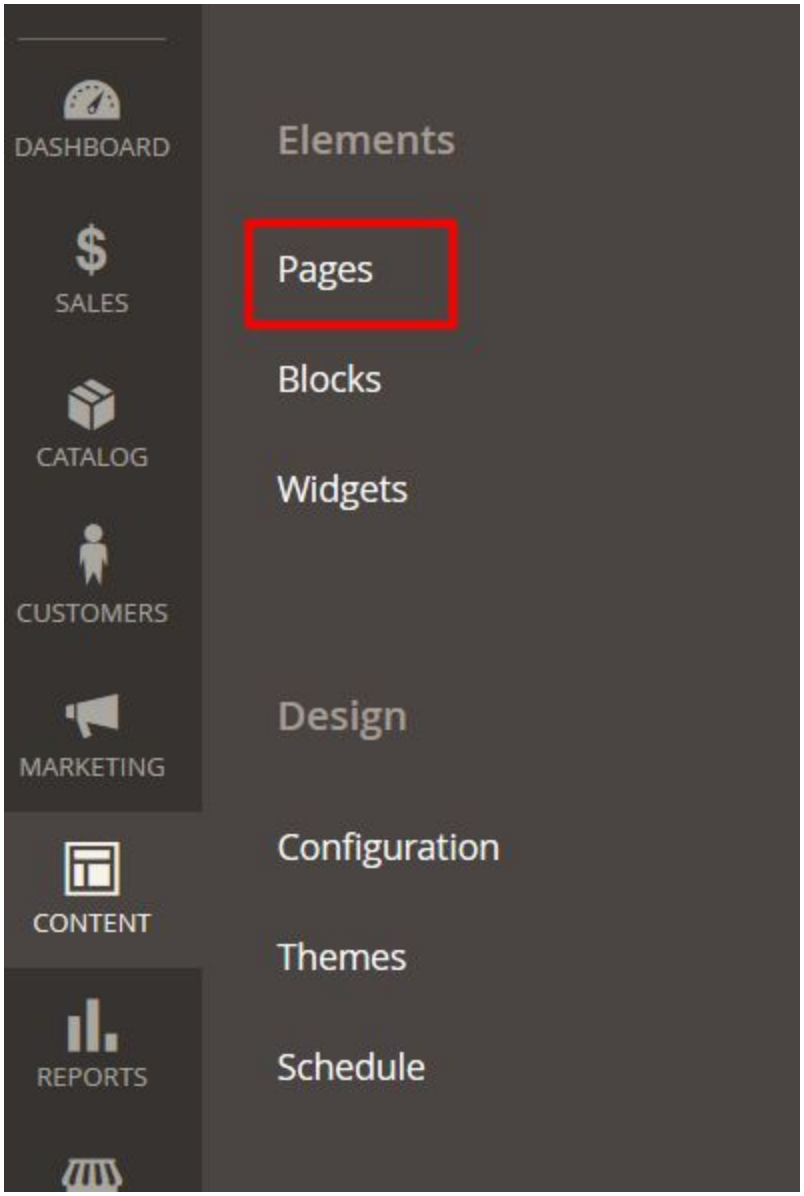
Select the previously created block with content in the Content section. Then select “Static block only” or “static block and products” in the display settings, depending on the design.



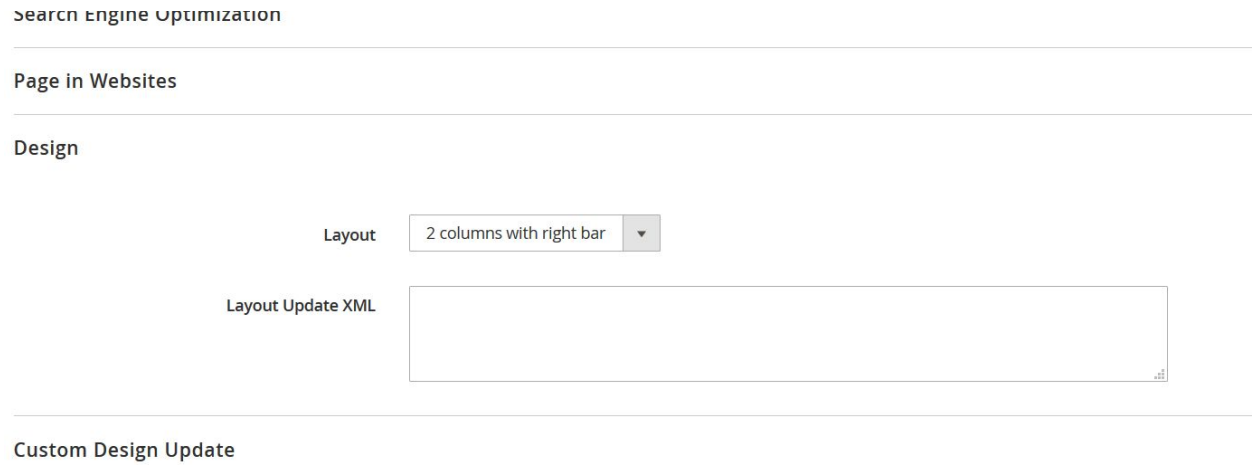
8.4 Customizing CMS pages

How can design changes (page layout) be configured on CMS pages?

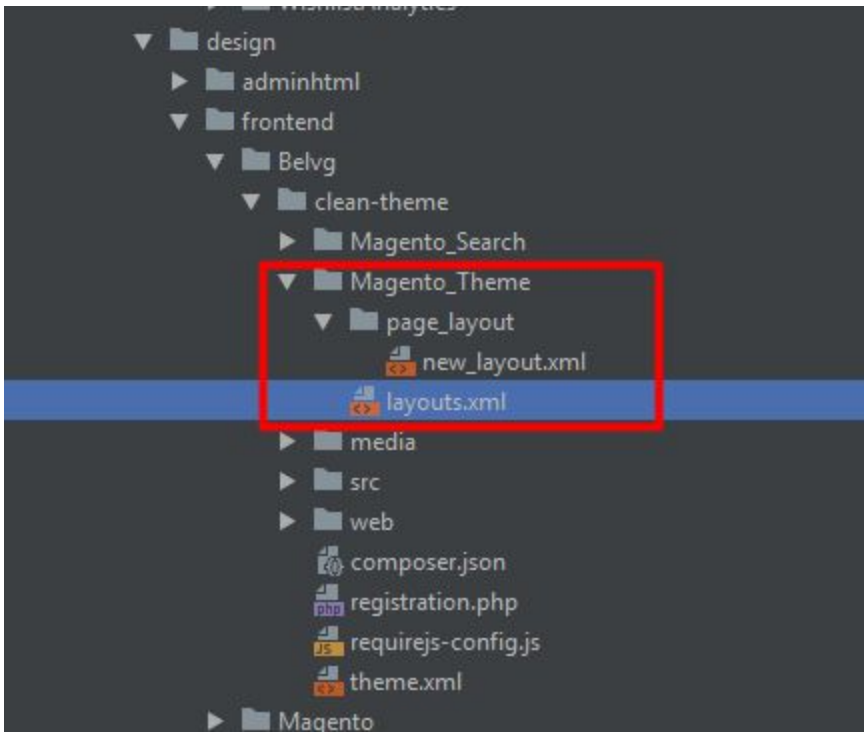
Find and set up pages you can on the path: Content > Pages.



In the Design tab, you can select the layout available in the page_layouts.xml list, as well as make additional changes to it (remove, move or add an element, as was already shown above in the category settings).



You can also create your layout. Create the Magento_Theme directory in your theme and a structure theme, copying the original one:

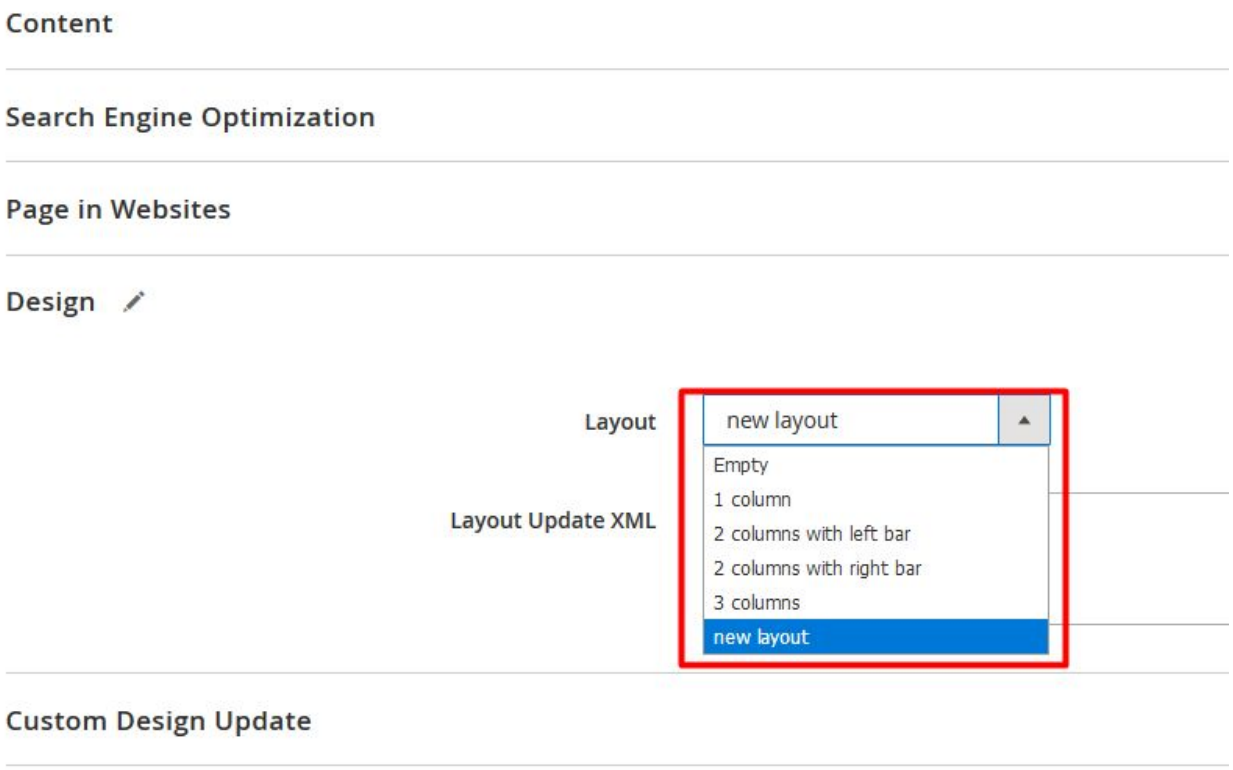


Indicate the needed containers and blocks in the new layout and register it in the layouts.xml

```
<page_layouts xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/PageLayout/
etc/layouts.xsd">
```

```
    ...
    <layout id="3columns">
        <label translate="true">3 columns</label>
    </layout>
    <layout id="new_layout">
        <label translate="true">new layout</label>
    </layout>
</page_layouts>
```

Now in the page settings we can choose the new layout:



Demonstrate an understanding of static variables in CMS blocks and pages.

Demonstrate an understanding of the use of CMS template directives (var, store, block, ...).

There is a list of directives that simplify work with Magento. Applying them while making content in the editor, you won't see the outdated information on the page. All links are up to date, the working hours and your store number are changed as soon as they are changed in settings.

Let's consider them in detail:

- {{var your_variable}} is a template variable to insert the store or customer data.
  If the variable is not included in the template, it will not be displayed in the end.
- {{customvar code="your_variable" }}
  A full list of custom variables can be found here: System> Custom Variables. There you can create new ones.
- {{store url="your_path" }} is used to create links.
  The example above will provide the following link:: https: //your.domain/your_path
- {{block}} renders the block.
  Specifying the class, the block type you've mentioned is rendered
  {{block class="\Vendor\Module\Block\YourBlock" }}
  Or you can display the CMS block by id or identifier:
  {{block id="your_block_id" }}
  or
  {{block id="your_block_identifier" }}
- {{media url="/icons/your_image.jpg"}} renders the link from the media folder you've specified, for example,
  https://your.domain/media/icons/your_image.jpg

- {{trans}} inserts a transatable line. Set the formatting to it or use the following construction:
  {{trans "Hi, %your_variable" your_variable=$method() }}
- {{css file="css/filename.css"}} inserts the content of the stylesheet into the html of the page as <style>content</style>
- {{inlinecss file="css/filename.css"}} renders css file into inline. Unlike the previous directive, the style will be inserted into the style = ”” attribute of the elements to which the styles are applied.
- {{if}} renders the variable if it doesn’t have the value in template.
  {{if your_variable}}
  Some text {{var your_variable}},
  {{else}}
  Some other text
  {{/if}}

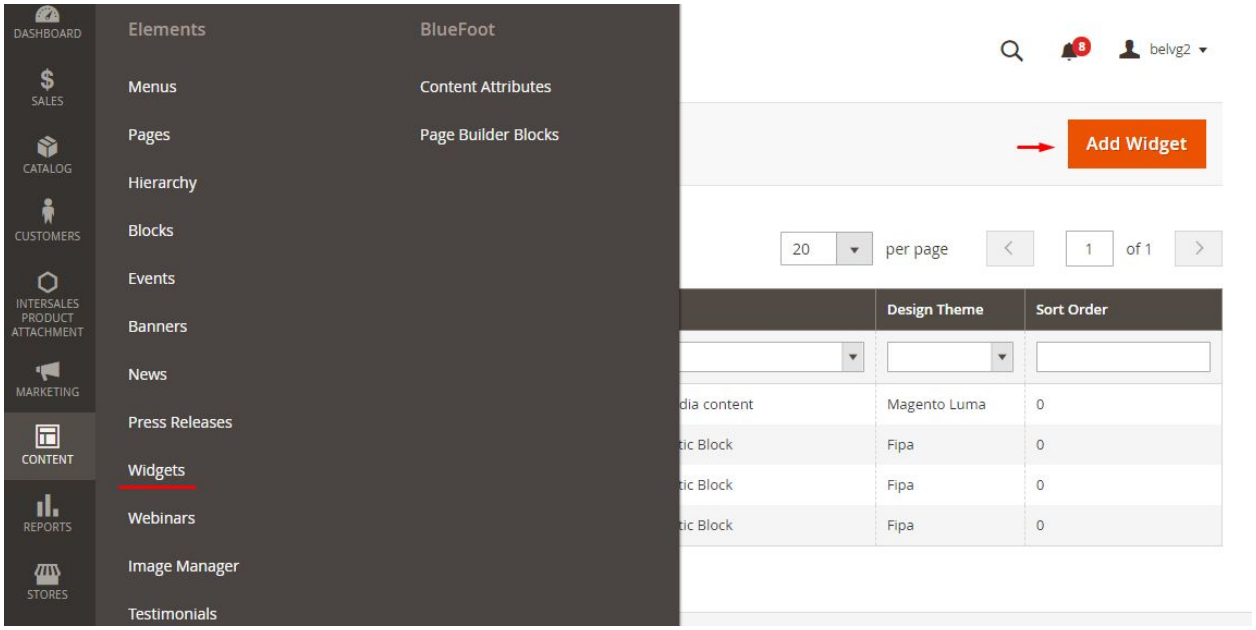
8.5 Customizing widgets

How is a widget instance created? Where can widgets be used?

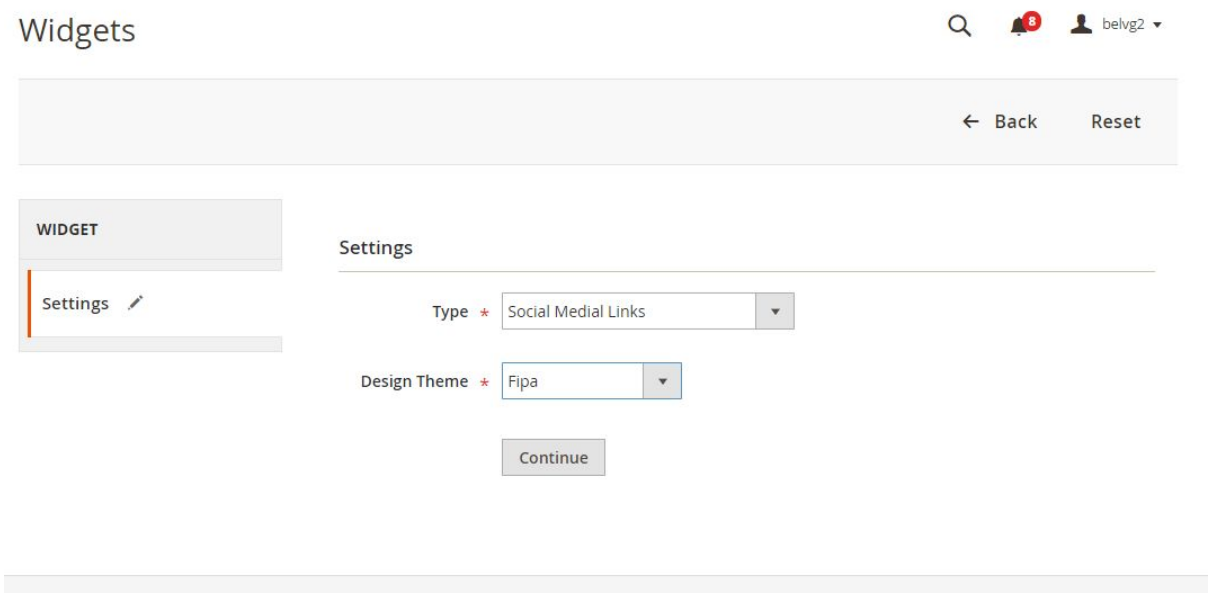
Widgets, especially custom ones, play an important role in Magento 2, particularly in terms of functionality. Using the functionality of Custom Widget in Magento 2 allows us to create custom widget templates. A custom widget can sometimes provide a better way to edit or add quality content inside CMS blocks or pages.

To use the widget and display it, for example, on the catalog page, we need to do the following:

Go to Admin panel > Content > Widgets > press Add widget button

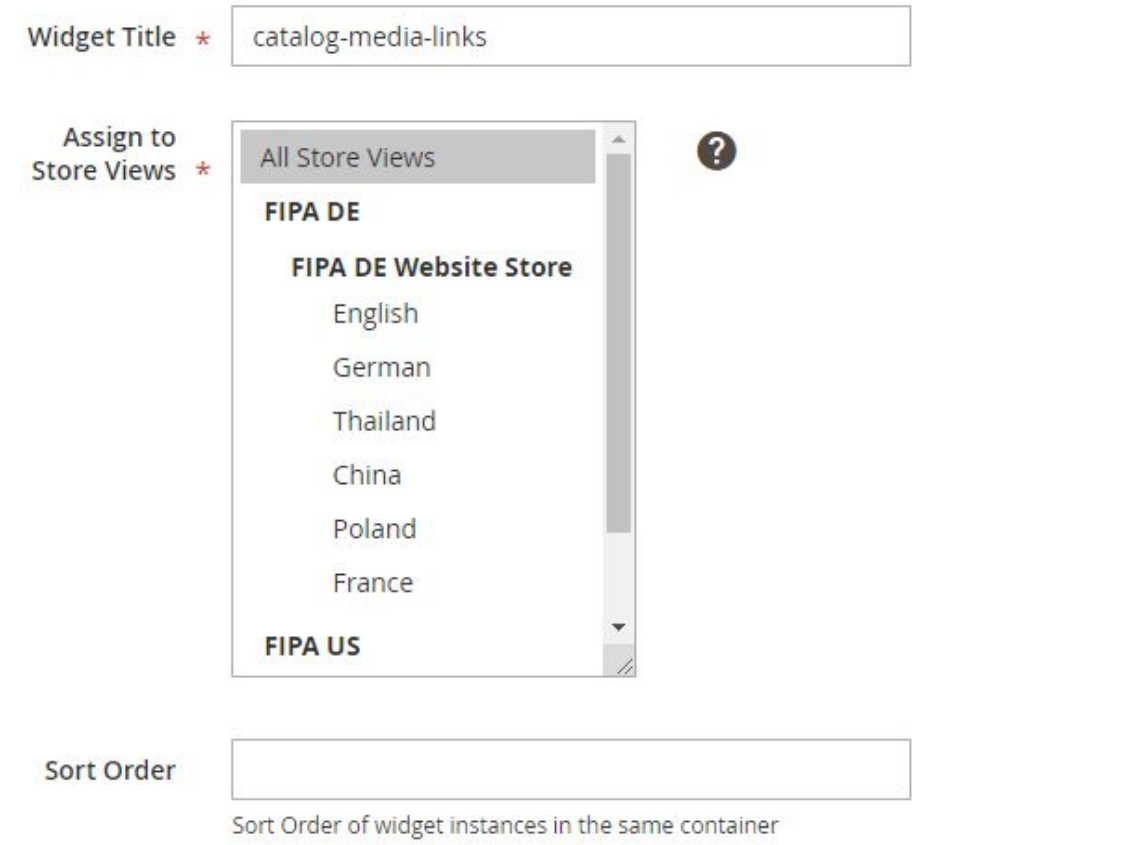


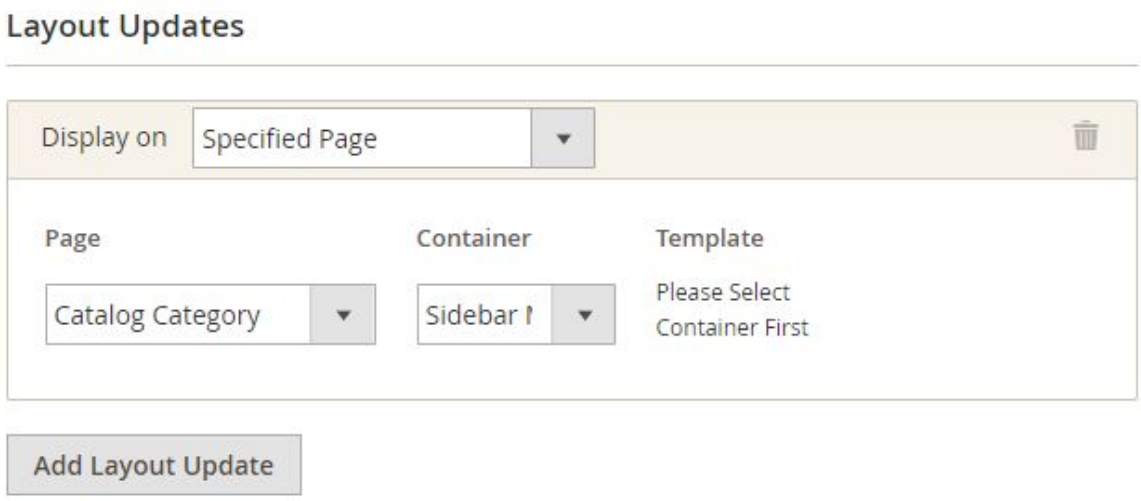
Choose the widget type (the social media link, for example) in setting and a theme to which we apply the widget and click “continue”



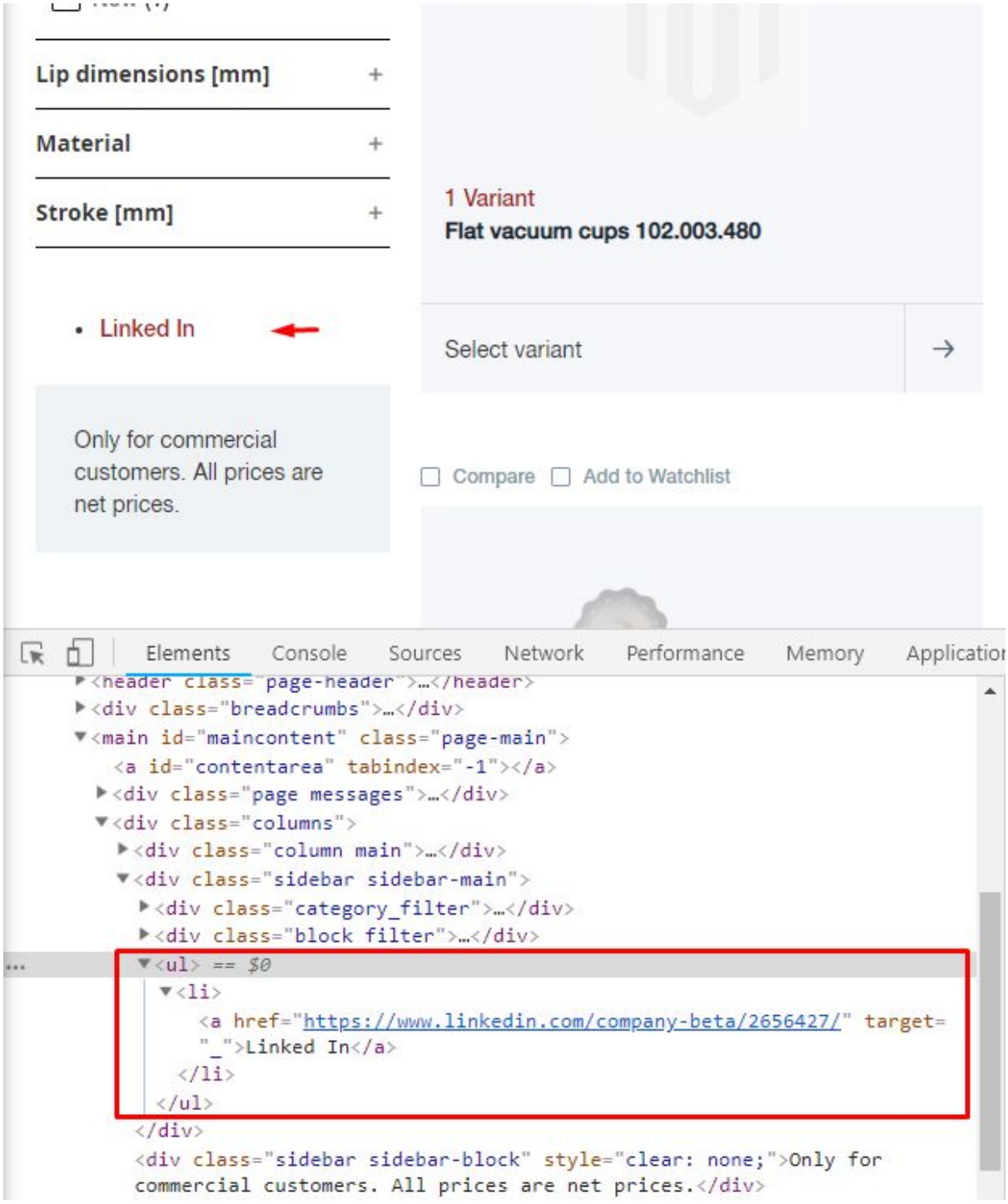
In the next section, specify the widget’s name and store view.

Then layout update. Choose where to display your widget, it can be any place on the site:





In our case, this is the sidebar on the catalog page:


Lip dimensions [mm]
Material
Stroke [mm]
Linked In
Only for commercial customers. All prices are net prices.
1 Variant
Flat vacuum cups 102.003.480
Select variant
Compare
Add to Watchlist
Elements
Console
Sources
Network
Performance
Memory
<header class="page-header">…</header>
…
<main id="maincontent" class="page-main">

…

…

…
…
 == $0

Linked In

Only for commercial customers. All prices are net prices.

How can a custom widget target be created? Demonstrate an understanding of configuring a widget instance.

There is also an opportunity to create your widgets in Magento:

First, we need to create a new module that requires the namespace and module folders. In the example, we will use *Belvg* for the namespace and *CustomWidget* for the module name.

app/code/Belvg/CustomWidget/composer.json

This file will be loaded using Composer every time we run it. But in fact we do not use Composer with our module.

Now we need to register our module in Magento. So, we create the *register.php* file in the following catalog:

app/code/Belvg/CustomWidget/registration.php

```
<?php
\Magento\Framework\Component\ComponentRegistrar::register(
 \Magento\Framework\Component\ComponentRegistrar::MODULE,
 'Belvg_CustomWidget',
 __DIR__
);
```

Now we create the last registration file - *module.xml.*

app/code/Belvg/CustomWidget/etc/module.xml

```
<?xml version="1.0" ?>
     <config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:Module/etc/modul
e.xsd">
     <module name="Belvg_CustomWidget" setup_version="1.0.0"/>
</config>
```

Next, we will create a widget configuration that is almost identical to the configuration used in the advanced widget. We create the *widget.xml* file in the /etc folder.

app/code/Belvg/CustomWidget/etc/widget.xml

```
<?xml version="1.0" ?>
<widgets xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Widget:etc/
widget.xsd">
 <widget class="Belvg\CustomWidget\Block\Widget\Samplewidget"
id="belvg_customwidget_samplewidget">
   <label>Belvg Sample Widget</label>
   <description></description>
   <parameters>
     <parameter name="widgettitle" sort_order="10" visible="true"
xsi:type="text">
       <label>Title</label>
     </parameter>
     <parameter name="widgetcontent" sort_order="20" visible="true"
xsi:type="textarea">
       <label>Content</label>
     </parameter>
   </parameters>

 </widget>
</widgets>
```

In the code above, we get *title* and *content* as parameters that will be displayed wherever the widget is called.

The <widget> tag contains the class of blocks - Belvg\CustomWidget\Block\Widget\Samplewidget.

The next step is to create the user block what will use our widget. For example, we will extend the *ProductList* block from *Magento_CatalogWidget*. This will give us all the existing widget functions with a list of products in the catalog.

We will override the *createCollection* method from *ProductList* to add the sort order to the collection. Add methods of getting attributes *"collection_sort_by"* and *"collection_sort_order"* with a rollback to the default values that are defined at the beginning of the class. These custom parameters are defined in the *widget.xml* configuration.

app/code/Belvg/CustomWidget/Block/Widget/Samplewidget.php

```
<?php

namespace Belvg\CustomWidget\Block\Widget;

use Magento\Framework\View\Element\Template;
use Magento\Widget\Block\BlockInterface;

class Samplewidget extends Template implements BlockInterface
{

 protected $_template = "widget/samplewidget.phtml";

}
```

Belvg\CustomWidget\Block\Widge\Samplewidget is declared by the code above. In this file, we assign a custom template file inside the *$_template* variable.

Then we add the widget template.

app/code/Belvg/CustomWidget/Block/view/frontend/templates/widget/samplewidget.phtml

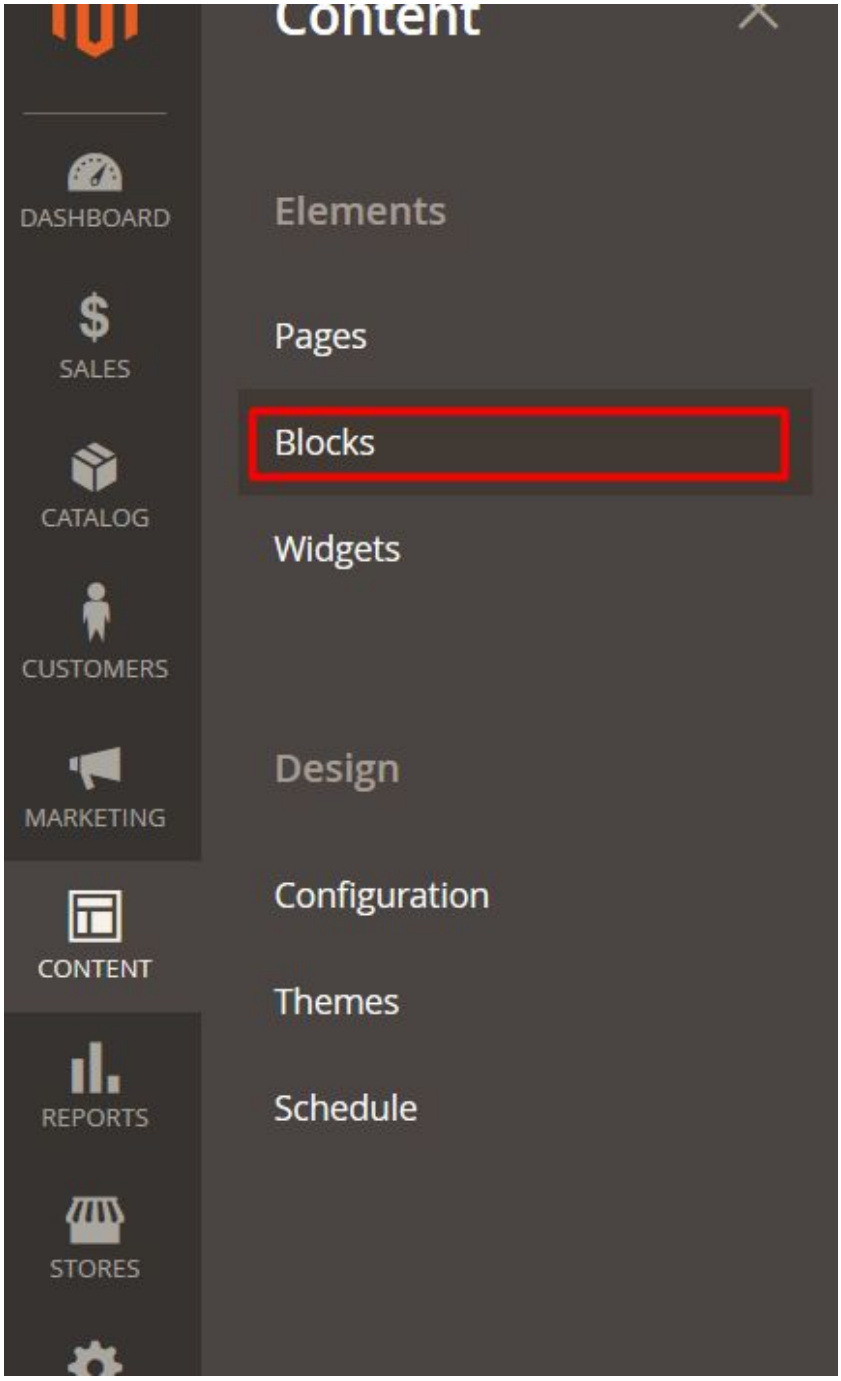
```
<?php if($block->getData('widgettitle')): ?>
 <h2 class='belvg-widget-title'>
<?php echo $block->getData('widgettitle'); ?></h2>
<?php endif; ?>
<?php if($block->getData('widgetcontent')): ?>
 <h2 class='belvg-widget-content'>
<?php echo $block->getData('widgetcontent'); ?></h2>
<?php endif; ?>
```

To display our custom widget on any page of the site, you need to use the method described at the beginning of the article.

8.6 Customizing CMS blocks

How do you create and insert CMS blocks?

You can create, develop and edit blocks in the Content > Blocks section.



A block can be inserted using a variable in xml and in phtml. Remember that to insert a block using one of these methods, we need "Your_block_id" which can be either the block id number or its identifier.

Using a variable, you can insert one block into another, on the page and more using the following construction:

```
{{block id="your_block_id"}}
```

Using xml:

```
<block type="Magento\Cms\Block\Block" name="your_block_name">
<arguments>
<argument name="block_id" xsi:type="string">your_block_id</argument>
</arguments>
</block>
```

The following part of a code can be displayed in phtml:

```
echo $this->getLayout()
          ->createBlock('Magento\Cms\Block\Block')
          ->setBlockId('your_block_id')
          ->toHtml();
```

Demonstrate an understanding of the use of CMS template directives (var, store, block, ...).

You can find the “insert variable” button in the content editor. It allows you to insert your site data. They will dynamically change if the information changes. There are no broken links when changing the site address or working hours!
You can find some of them below:

- `{{var your_variable}}` is the template variable used to insert store or customer data. You can also create your own custom ones and use them.

If the variable is not included in the template, it will not be displayed in the end.

```
{{customvar code="your_variable" }}
```

A full list of custom variables you can find on the path: System > Custom Variables. There you can also create new ones.

- `{{store url="your_path" }}` is used to create links
  The example above will give a link that will look like:
  https://your.domain/your_path
- `{{block}}` renders block.
  If you specify a class, the block of the type you specified is rendered.

```
{{block class="\Vendor\Module\Block\YourBlock" }}
```

Or you can display the CMS block by id or identifier:

```
{{block id="your_block_id" }}
```

Or

```
{{block id="your_block_identifier" }}
```

- {{media url="/icons/your_inage.jpg"}} renders a link from the media folder that you've specified. For example:
  https://your.domain/media/icons/your_image.jpg
- `{{trans}}` is inserted into the translatable line. You need to specify formatting to it or use the following construction:

```
{{trans "Hi, %your_variable" your_variable=$method()
}}
```

- `{{css file="css/filename.css"}}` inserts the content of the stylesheet as <style>content</style> into the html pages
- `{{inlinecss file="css/filename.css"}}` renders a css file into inline. Unlike the previous directive, the style will be inserted into the style="" elements attribute to which the styles are applied
- `{{if}}` renders the variable if it has the value in the template

```
{{if your_variable}}
```

```
Some text {{var your_variable}},
{{else}}
Other text
{{/if}}
```

There is a block with content for example:

```
<p>{{config path="general/store_information/name"}} want you to click
to this link:</p>
<p><a href="{{store url="custom_link"}}">click me!</a></p>
```

And place it on the page:

As a result, we have the following link:
https://your_domain.com/custom_link/

8.7 Customizing customer account pages

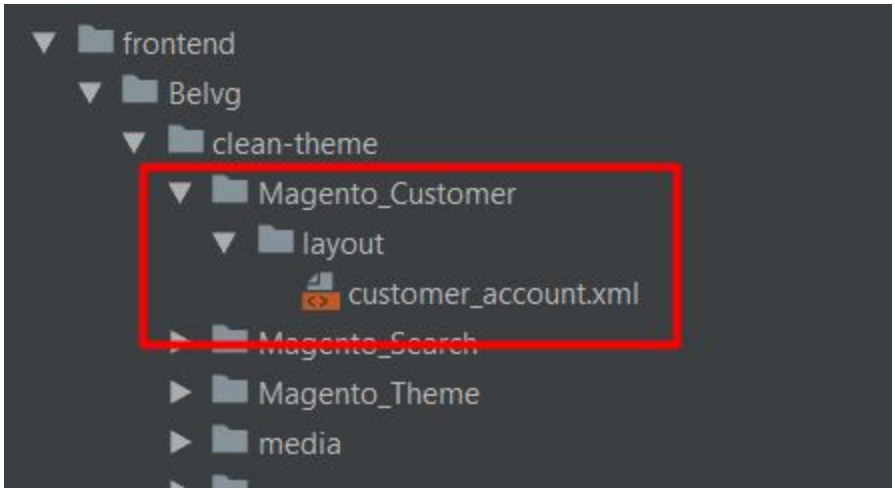
How do you remove or add an item from the customer account navigation using layout XML?

User pages inherit the design from the main customer_account.xml file (it is also possible to override less globally by writing an xml override for a specific page.
We can delete/move/insert an element using the following lines:

```
<referenceBlock name="block_name" remove="true" /> to delete
(<referenceContainer> for the container)
<move element="block_name" destination="block_name" /> to move
<block class="" name="block_name">
   <arguments>
       <argument>your_argument</argument>
   </arguments>
</block> to insert new element (it's possible to insert a container
using the <container> tag)
```

For example, let’s remove the wish list, rename the “Account information” link into the “Change account Information” and ad our list to the navigation end at out blog;

Create the structure in our theme:

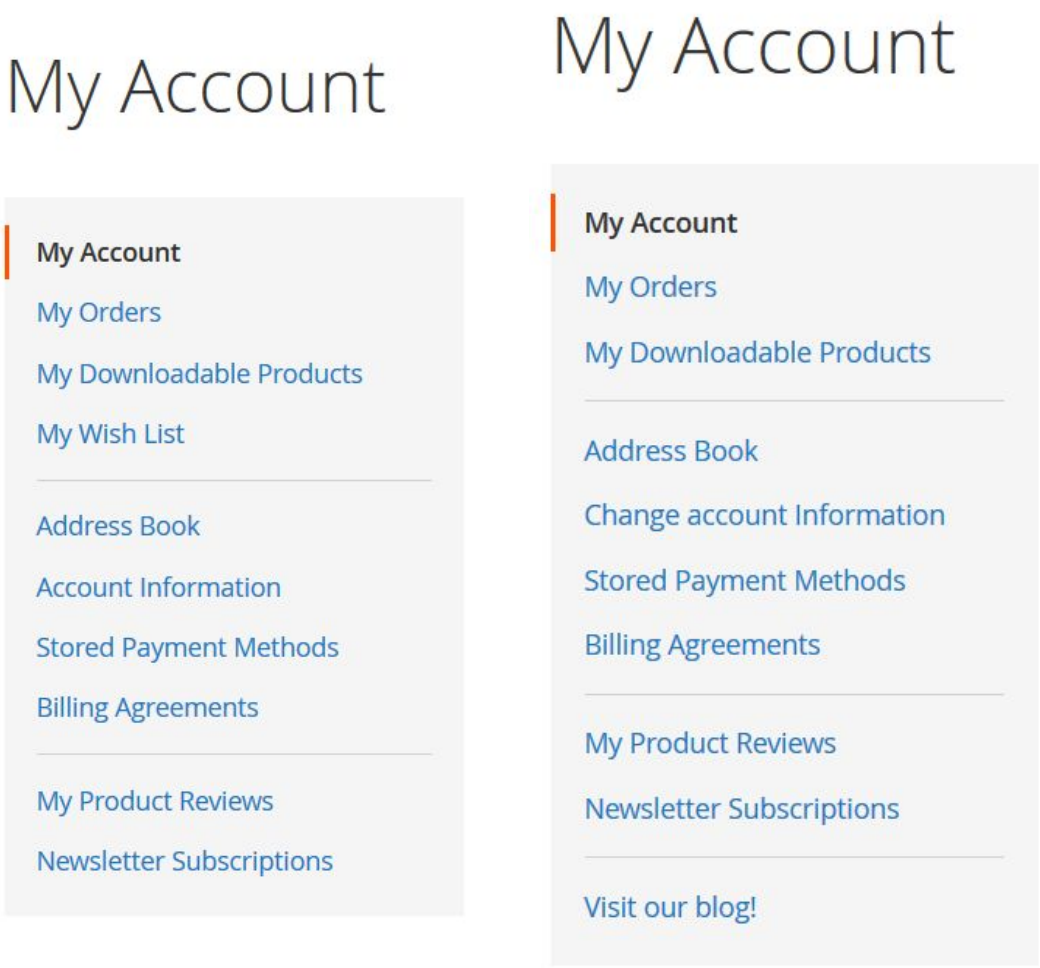


Add the following lines to the file:

```
<?xml version="1.0"?>
<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
layout="2columns-left"
xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/
page_configuration.xsd" label="Customer My Account (All Pages)"
design_abstraction="custom">
   <body>
       <referenceBlock
name="customer-account-navigation-account-edit-link">
           <arguments>
               <argument name="label" xsi:type="string"
translate="true">Change account Information</argument>
           </arguments>
       </referenceBlock>
       <referenceBlock
name="customer-account-navigation-wish-list-link" remove="true"/>
       <referenceContainer name="customer_account_navigation">
           <block class="Magento\Customer\Block\Account\Delimiter"
name="customer-account-navigation-delimiter-3"
template="Magento_Customer::account/navigation-delimiter.phtml">
               <arguments>
                   <argument name="sortOrder"
xsi:type="number">20</argument>
```

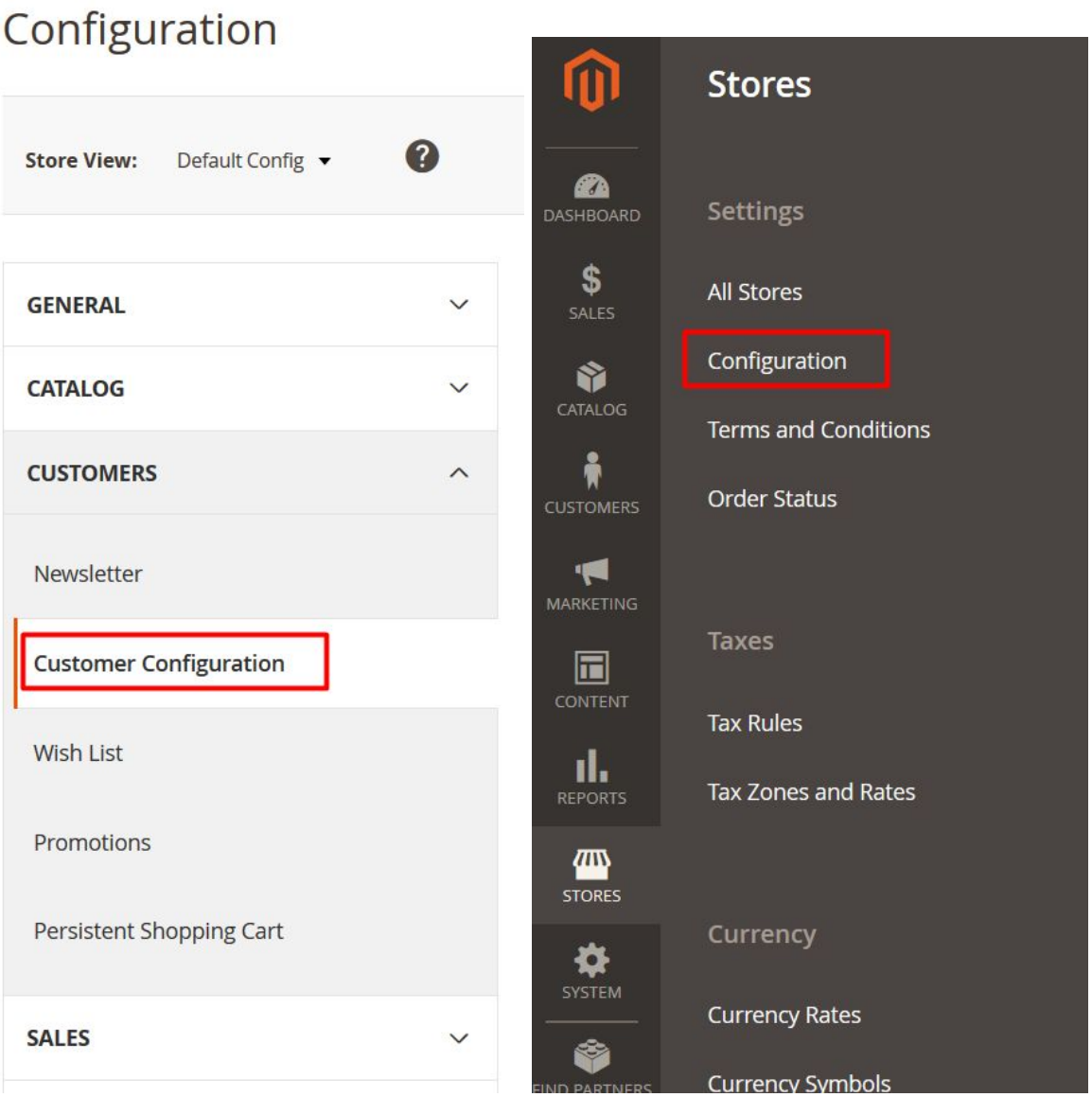
```
                </arguments>
            </block>
            <block
class="Magento\Customer\Block\Account\SortLinkInterface"
name="custom-link-to-blog">
                <arguments>
                    <argument name="label" xsi:type="string"
translate="true">Visit our blog!</argument>
                    <argument name="path"
xsi:type="string">https://belvg.com/blog/</argument>
                    <argument name="sortOrder"
xsi:type="number">10</argument>
                </arguments>
            </block>
        </referenceContainer>
    </body>
</page>
```

The result is following:



Demonstrate an understanding of formatting customer addresses.

User address settings can be found along the path: Stores > Configuration > Customers >Customer Configuration > Address templates.



Here you can configure the display of fields filled by the user using the following construction:

```
{{depend variable}}
{{var variable}}
{{/depend}}
```

They can be deleted/added/moved among each other.

8.8 Customizing one-page checkout

Demonstrate an understanding of the container blocks provided in the Magento checkout to display additional information.

Checkout in Magento 2 is made up of KnockoutJS components which are rendered using the Knockout JS template system. Magneto 2 defines all these components and their parent/child relationships in an XML file that can be expanded or redefined in your own theme or module.

The XML file is along the path:

```
app/design/frontend/Vendor_Name/Theme_Name/Magento_Checkout/layout/checkout_index_index.xml
```

If you want to change the Checkout (for example, add a block), you need to override the file creating a new one along the path:

```
app/design/frontend/Vendor_Name/Theme_Name/Magento_Checkout/layout/checkout_index_index.xml
```

Add the following code (to output CMS blocks) to the XML:

```
<referenceBlock name="checkout.header.wrapper">
 <container name="additional-block-wrapper"
label="additional-block-wrapper" htmlTag="div"
htmlClass="additional-block-wrapper">
   <block class="Magento\Cms\Block\Block" name="custom-block">
```

```
     <arguments>
       <argument name="block_id"
xsi:type="string">custom-block-checkout</argument>
     </arguments>
   </block>
 </container>
</referenceBlock>
```

Add the following code (to output template blocks) to the XML:

```
<referenceContainer name="page.bottom.container">
 <container name="custom-footer-wrapper"
label="custom-footer-wrapper" htmlTag="div"
htmlClass="custom-footer-wrapper">
   <block class="Magento\Framework\View\Element\Template"
name="custom-footer"
template="Magento_Theme::checkout-footer-custom.phtml" />
 </container>
</referenceContainer>
```

Template file:

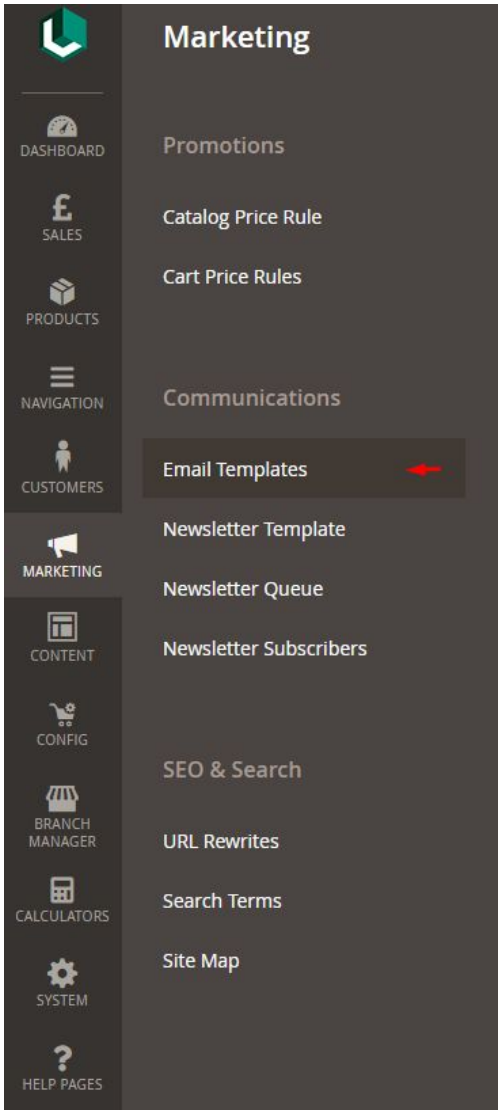
```
app/design/frontend/Vendor_Name/Theme_Name/Magento_Theme/templates/ch
eckout-footer-custom.phtml
```

8.9 Understand customization of transactional email templates

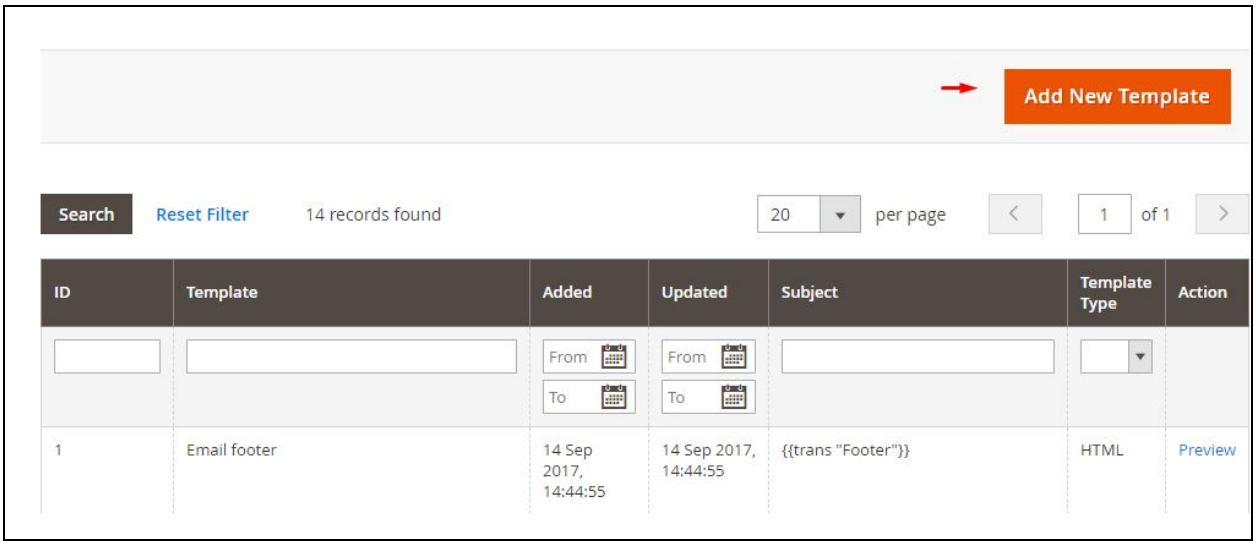
How do you create and assign custom transactional email templates?

Email templates creation and edition in Magento 2 is similar to the process in Magento 1. The main difference between Magento 1 and Magento 2 is that these templates have been renamed from “Transactional Emails” to “Email Templates”. You can still find and customize Magento 2 email templates by default, as well as add new templates to your store.

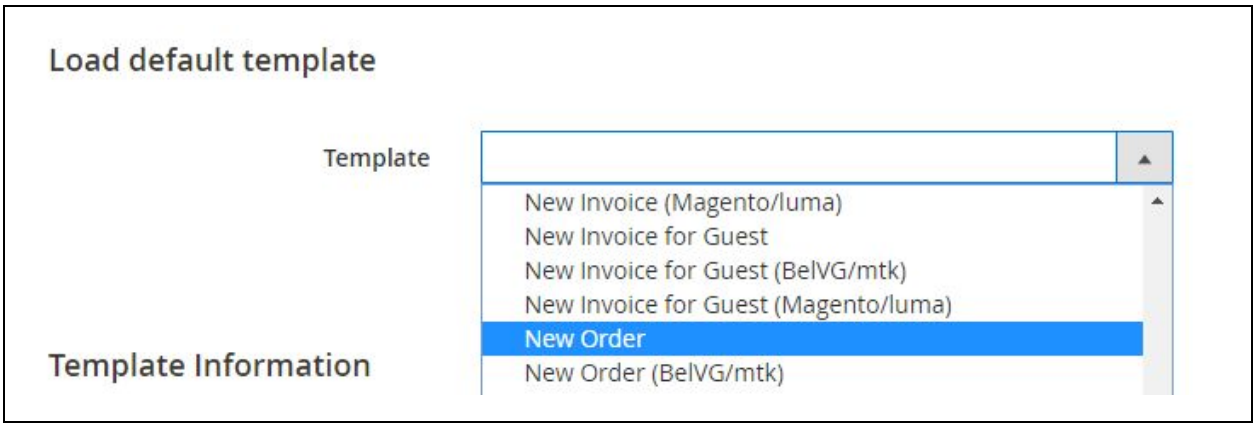
Go to the magento admin > Marketing tab > Email Templates



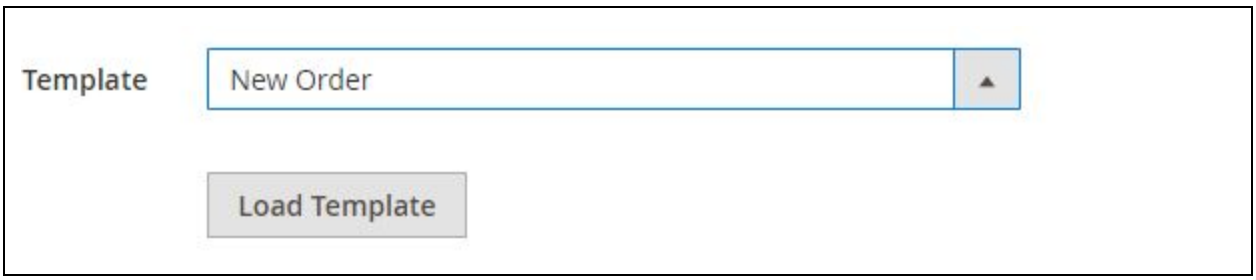
Push the “Add New Template” button



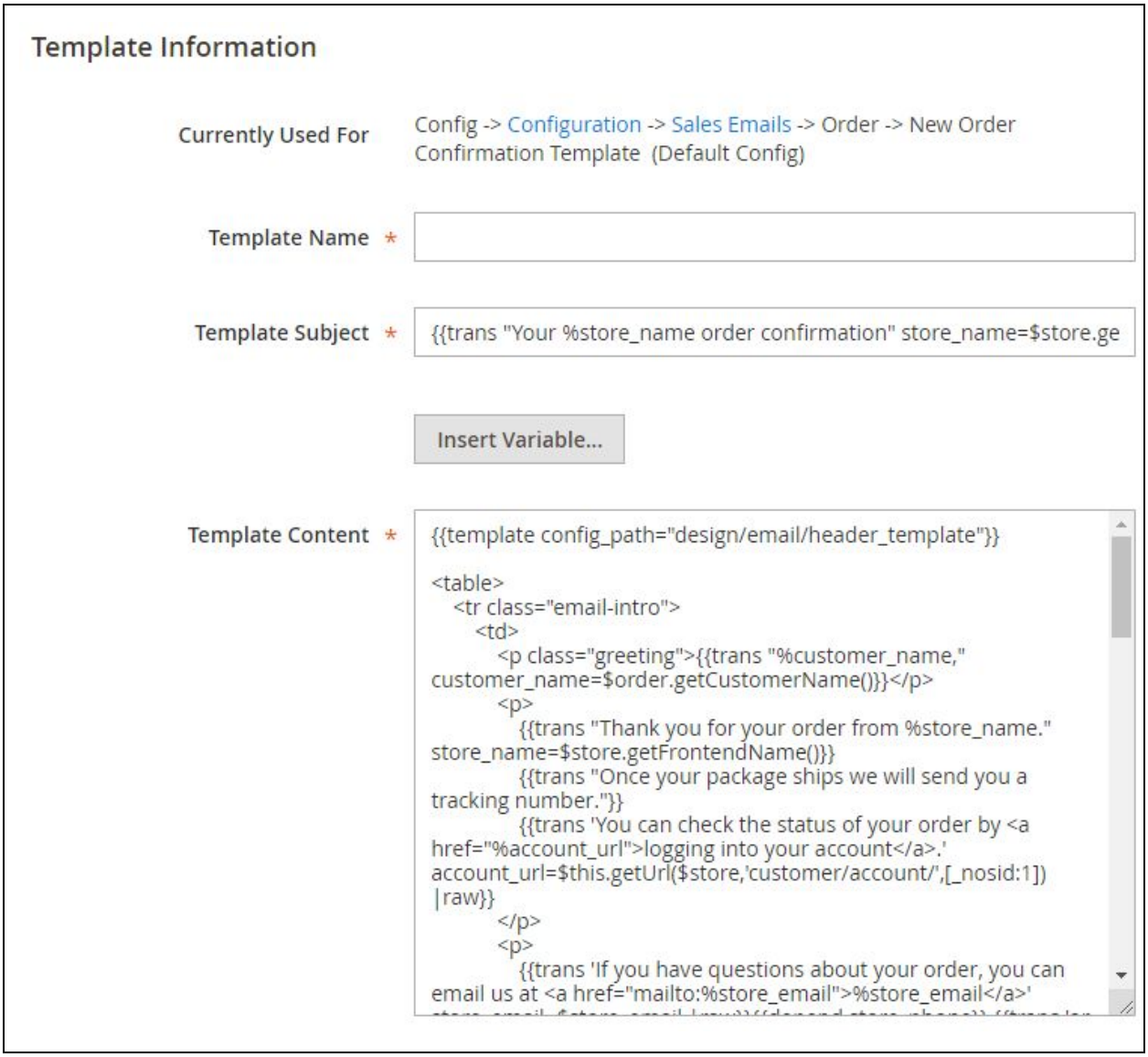
Choose the needed Template from the dropdown menu:



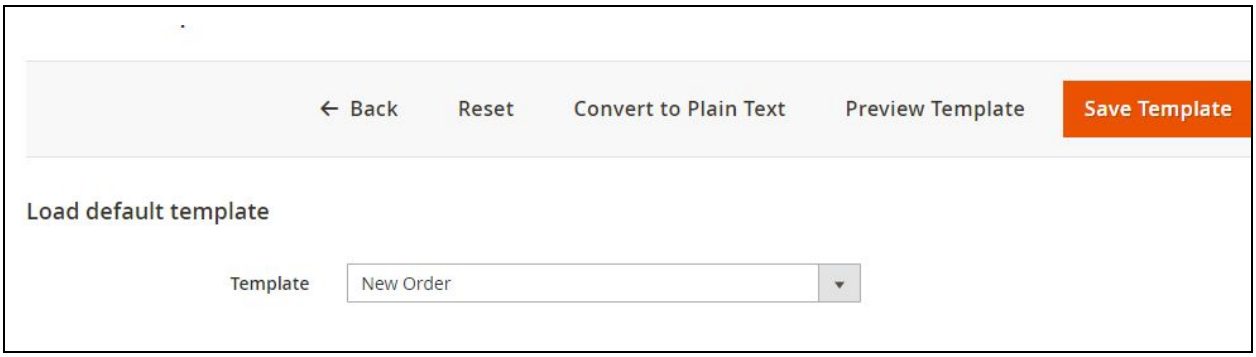
Push the “Load template” button



Write the new name for the email template

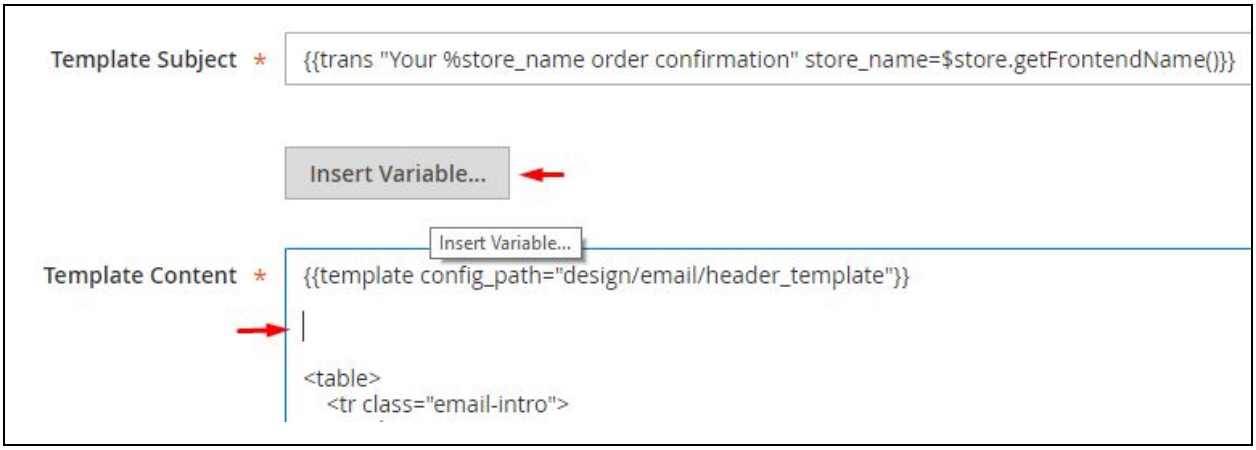


Save the template pushing on “Save Template”

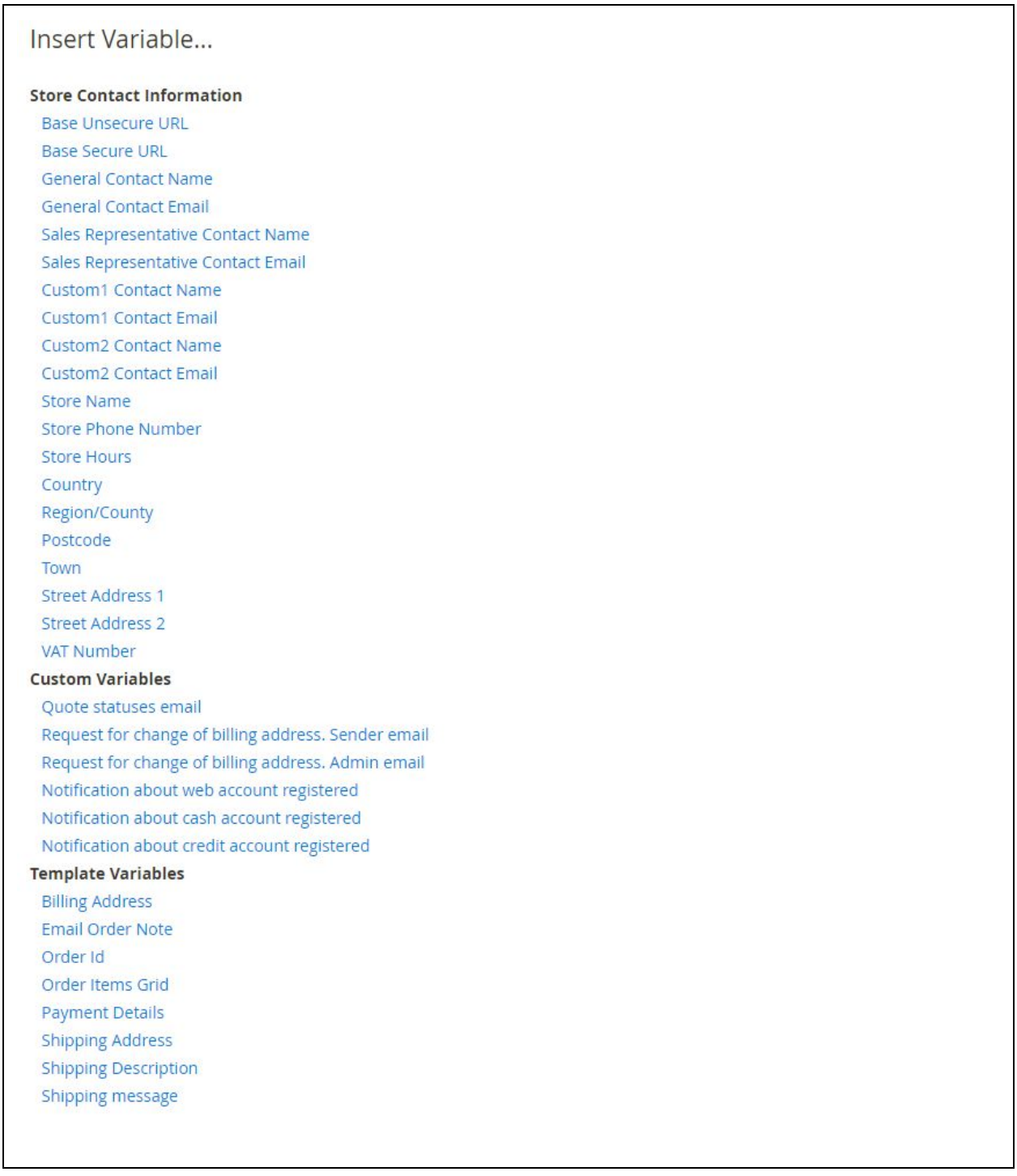


How do you use template variables available in all emails?

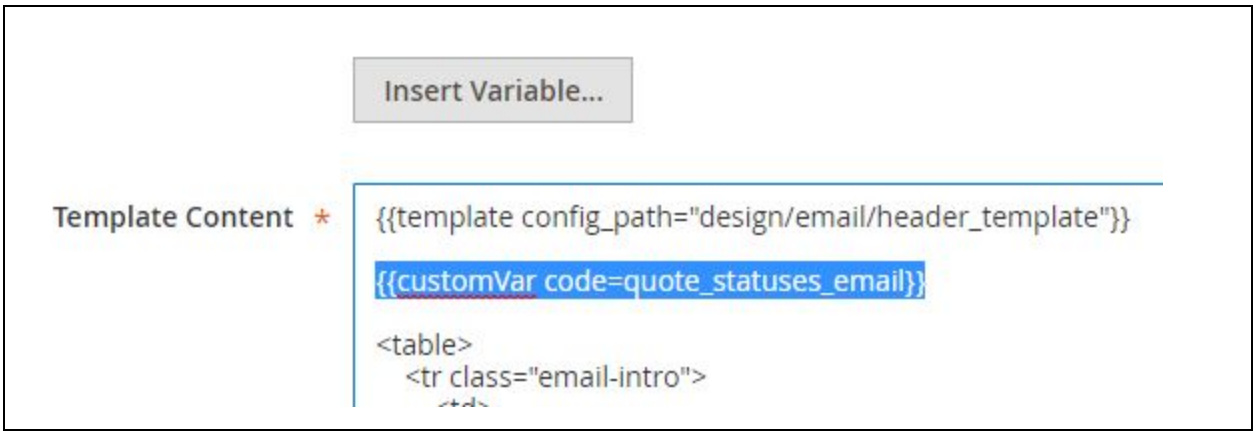
You can add a variable to any email template by placing the cursor right where you want to add the variable and clicking “Insert Variable”



A list of variables:



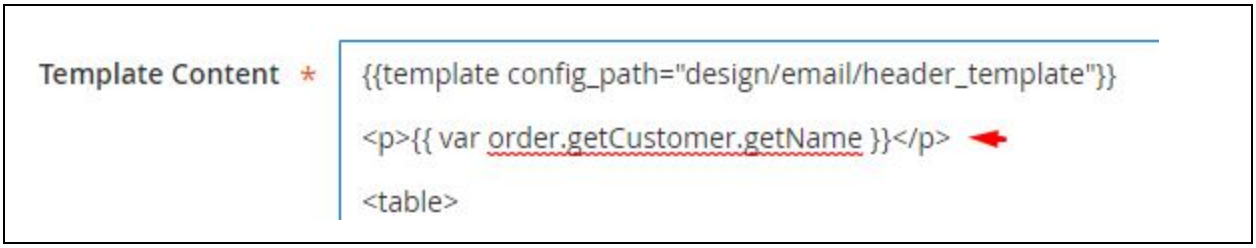
For example, choose Quote statuses email:



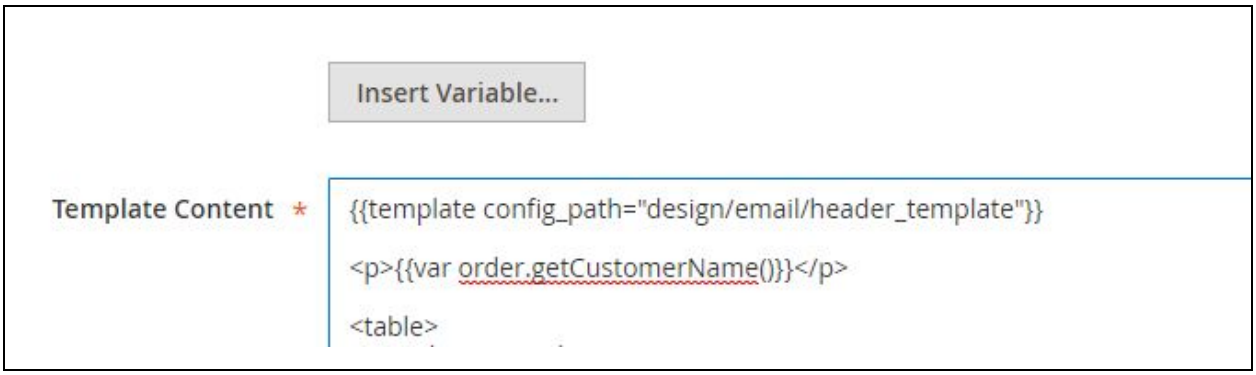
How do you access properties of variable objects (for example, var order.getCustomer.getName)?

To access the value of any variable in the letter template, you need to put the desired variable in double curly braces {{}} and insert it in the desired place in the template, in the Template Content field.
For example, `var order.getCustomer.getName:`

(this variable doesn't work)




```
( {{var order.getCustomerName()}} works ):
```



Note that different email templates can display different variables depending on the value.

How can you create a link to custom images from transactional email templates?

If you don't want to create a plugin, you can add your images from the folder - web/images - module level and themes and specify the path to the images

```
<img src="{{view url='Package_Modulename/images/custom-image.png'}}"
height="60" width="60" alt="{{trans 'Custom image'}}" />
```

There is another way to add images

```
<img src="{{media url="custom-image.png"}}">
```

where the value of the url parameter is the path as to the media folder. You can add an image through the wysiwyg editor and then add a link to it

```
<img src="{{media url="wysiwyg/custom-image.png"}}">
```

How do you create links to store pages in transactional email templates?

You need to do is insert the <a> link in the Template Content field and in the href attribute specify the variable on the Base Unsecure URL: {{config path="web/unsecure/base_url"}} or Base Secure URL: {{config path="web/secure/base_url"}}
after the brackets indicate the page id:

For example, the link to the help page (www.site.com/help)

```
<a href="{{config path='web/secure/base_url'}}/help">go to the help page</a>
```

Alternative way go to home inside url the path is relative to the main site

Section 9: Implement Internationalization of Frontend Pages

9.1 Create and change translations

Demonstrate an understanding of internationalization (i18n) in Magento. What is the role of the theme translation dictionary, language packs, and database translations?

Magento 2 has good functionality to translate anything into lots of languages. Magento includes search functions for all translatable strings within a specific path. You can use this for the entire Magento 2 or just for a module or theme.

There are certain translation conditions. There is a **dictionary** - a comma-separated value (.csv) file containing at least two columns: the original phrase in the en_US locale and the translation of this phrase into another locale. For example, translation from English (en_US) to French (fr_FR):

```
/app/design/frontend/BelVG/<theme_name>/i18n/fr_FR.csv
```

```
"Contact Form","Votre message à FIPA"
"Send message","Envoyer le message"
"Choose a file","Sélectionner le(s) fichier(s)"
"Drop here ...","Vous pouvez télécharger jusqu'à 5 fichiers par glisser-déplacer."
"Enter message","Saisir le message"
"Enter e-mail","Saisir l'adresse e-mail"
"Contact","Contact"
"Send callback preferences","Envoyer le souhait de rappel"
"We'll call you when it's convenient.","Nous vous appellerons quand vous aurez le temps."
"Service Hotline","Assistance téléphonique"

```

The translations specified for a theme or module are the first two sources of translation data. These translations can be overridden by the language pack or in the database.

The language pack is a set of dictionaries for a particular language along with meta-information.
Magento allows creating the following language packs:

- A set of .csv files for modules and themes. These package files are intended for deployment in modules. For instance:

```
__/app
  |__/code
  | |__/Magento
  |   |__/Catalog
  |   | |__/i18n
  |   |   |-- de_DE.csv
  |   |__/Checkout
  |   | |__/i18n
  |   |   |-- de_DE.csv
  |   |__/Customer
  |     |__/i18n
  |       |-- de_DE.csv
  |__/design
    |__/frontend
      |__/<Vendor>
```

```
|__/<theme>
  |__/i18n
    |-- de_DE.csv
```

- Language packs containing the entire dictionary in one catalog.

To create a language pack in the .csv file, there are required additional columns that indicate the themes or modules in which translations were found.

To create a language pack, you need to find all the lines in the Magento application. You can run this command to get this information:

```
bin/magento i18n:collect-phrases -m
```

The command searches for all Magento 2 files, including modules and themes, looking for all translatable strings.

When launched with the -m flag, two additional columns are added: **type** and **module**. A **type** is either a theme or a module. The **module** column represents the module that uses this translation.

You can also search for an individual module (s):

```
bin/magento i18n:collect-phrases app/code/BelVG/CustomModule
```

When you get the translated lines, run this command (substituting the ABSOLUTE path to the CSV file and specifying the target language):

```
bin/magento i18n:pack [-m|--mode={merge|replace}]
[-d|--allow-duplicates] <source> <locale>
```

```
bin/magento i18n:pack
/var/www/html/magento2/app/code/ExampleCorp/SampleModule/i18/de_DE.csv de_DE
```

Next, you need to create a new language module along the path *app/i18n/Belvg_lng_pack/*
The name of the module is up to you.
The module contains standard files:

- **registration.php** uses *\Magento\Framework\Component\ComponentRegistrar::LANGUAGE* component type
- **language.xml:**

```
<?xml version="1.0"?>

<language xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:App/Language/package.xsd">
   <code>de_DE</code>
   <vendor>splendid</vendor>
   <package>de_de</package>
   <sort_order>100</sort_order>
   <use vendor="oxford-university" package="en_us"/>
</language>
```

You can include multiple **<use />** nodes in the **language.xml** file. If Magento 2 does not find a string in the included language pack, it will search through each **<use />** (and subsequently the **<use/>** nodes of this language pack) until it finds a suitable translation.

<sort_order> is a priority for package downloading If there are several language packs for the store.

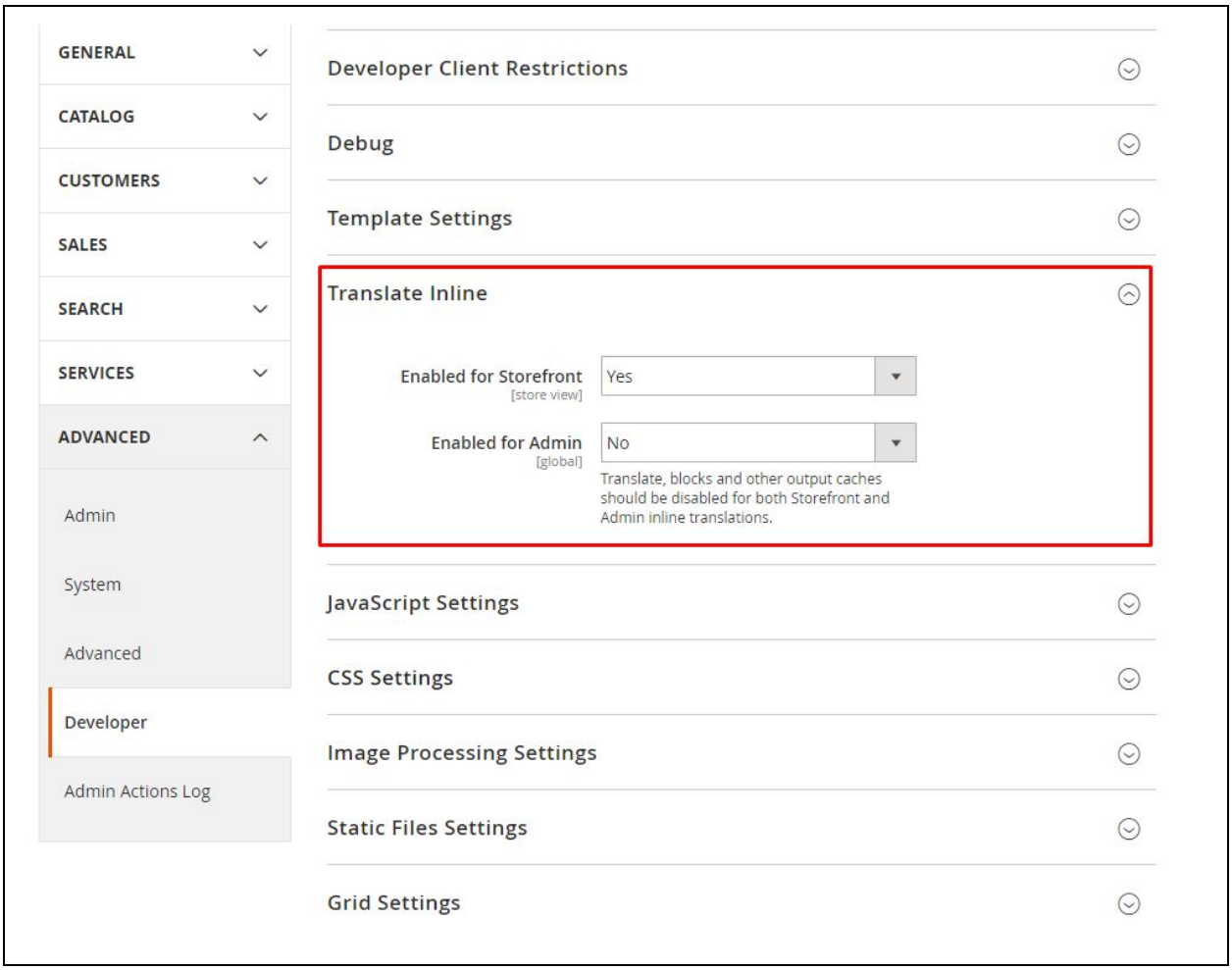
Next, inside the language pack, add the CSV file generated above, called by contents of the <code /> node with the suffix .csv.

```
(/app/i18n/splendid/de_DE/de_DE.csv)
```

Database translations

Database translations are the simplest but it’s hard to transfer from installing Magento 2 to another. Translations can be found in the **translation** table.

The easiest way to create a new translation is to enable the built-in translation (Store > Configuration > Developer > Translate Inline).



You can also insert new lines into the **translation** table manually. When translating using the built-in translation, disable caches: **Translations, Block HTML** and **Full Page.**

Understand the pros and cons of applying translations via the translate.csv file versus the core_translate table. In what priority are translations applied?

Each method has its pros and cons.

For example, translations using **translate.csv** are much more convenient to transfer from one instance to another. But the **inline translate** function is better when you update the **translate.csv** file

If we talk about the translation order, **module translation** is involved at the beginning, then **theme translation**, **translation package**, and **database translation** at the end.

So, using the **inline translation** is better to override any translation.

9.2 Translate Theme Strings for .PHTML, EMAILS, UI Components, .JS Files

To use string translation in .phtml files, you need to use the following construction:

```
<?= __('Something to translate');?>
```

The variant with variable:

```
<?= __('Hello, mr %', $name);?>
```

To use string translation in Email templates file, you need to use the following construction:

```
{{trans "Something to translate"}}
```

The variant with variable:

```
{{trans "%items are shipping today." items=shipment. getItemCount}}
```

Demonstrate an understanding of string translation in JavaScript.

Directly in JavaScript files:

The first step is to call the jquery and mage/translate libraries in the js file:

```
require([
   'jquery',
   'mage/translate'
], function ($){...});
```

Then place the needed string to the $.mage.__(‘’) construction;

```
$.mage.__('Close');
```

The variant with variable:

```
$.mage.__("%1 items in your cart").replace("%1", numberOfItems);
```

There is another method with connecting only the mage/translate library and using the $t variable

```
require([
   'mage/translate'
], function ($t) {...});
```

After that, put the necessary string in the construction $t('');

```
$t('Some translate string');
```

To use translation in XML file UI Component, use the translate=”true” argument for the needed element with the following text:

```
<action name="approve">
   <argument name="data" xsi:type="array">
       <item name="config" xsi:type="array">
           <item name="type" xsi:type="string">approve</item>
           <item name="label" xsi:type="string"
translate="true">Approve</item>
       </item>
   </argument>
</action>
```

String translation methods in html templates UI components files:

data-bind="i18n: 'Save in address book'"

```
<label class="label" for="billing-save-in-address-book">
   <span data-bind="i18n: 'Save in address book'"></span>
</label>
```

data-bind="attr: {title: $t('Go to checkout')}"

```
<button
       id="top-cart-btn-checkout"
       type="button"
       class="action primary checkout"
       data-bind="attr: {title: $t('Go to checkout')}">
   <!-- ko i18n: 'Go to checkout' --><!-- /ko -->
</button>
```

translate="'Columns'"

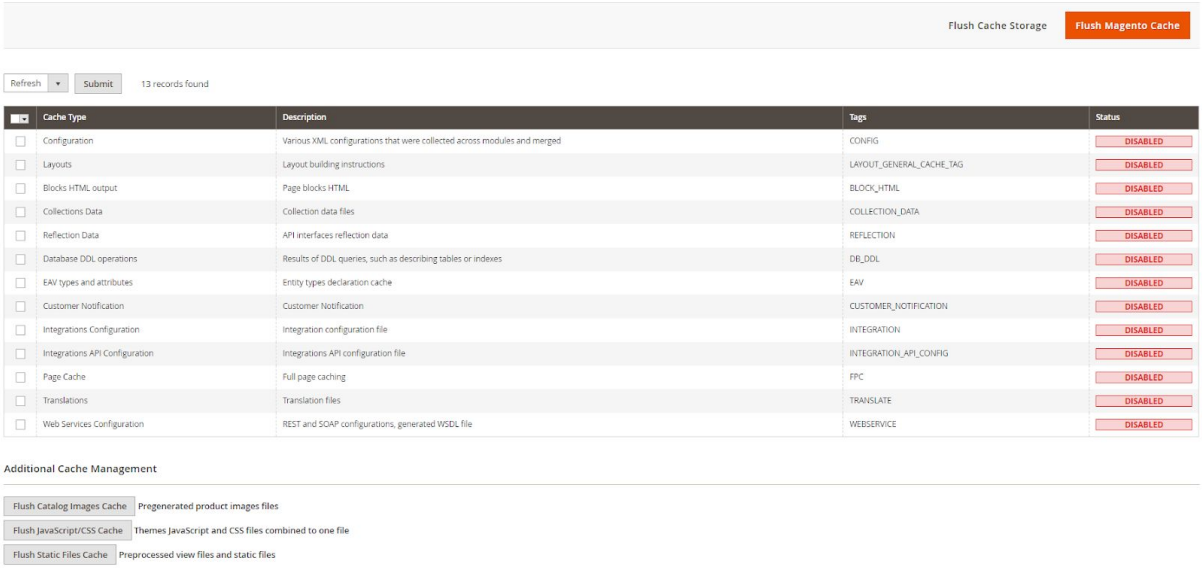
```
<span class="admin__action-dropdown-text" translate="'Columns'"/>
```

Section 10: Magento Development Process

10.1 Determine ability to manage cache

Demonstrate an understanding of configuring the Magento cache types for development and production.

Magento has an advanced data caching system, which can significantly speed up the site. Magento caching system consists of many types of cache. Let's take a closer look at them. To do this, open the the admin panel and go to *System > Tools > Cache Management*:



There are 13 main cache types. Let’s consider each of them briefly and then discuss how to work with them.

Configuration contains integrated configuration from all modules and store-specific settings. The cache type codename (used when accessing this cache type on the command string) is **config**.

Layouts contains compiled layouts of pages (these compiled layouts are formed from all layout files that are used on specific pages).The cache type codename is **layout**.

Blocks HTML output contains fragments of HTML page code for various blocks. The cache type codename is **block_html**.

Collections Data contains database query results. The cache type codename is **collections**.

Reflection Data contains dependencies between the Webapi and the Customer modules. The cache type codename is **reflection**.

Database DDL operations contains the DDL queries results such as tables or indexes descriptions. The cache type codename is **db_ddl**.

EAV types and attributes contains metadata connected with EAV attributes (such as attribute mapping, search parameters, object type declarations, etc.).The cache type codename is **eav**.

Customer Notification contains various temporary notifications appearing in the user interface. The cache type codename is **customer_notification**.

Integrations Configuration contains a compiled integration configuration file. The cache type codename is **config_integration**.

Integrations API Configuration contains a compiled integration APIs configuration of the Store's Integrations. The cache type codename is **config_integration_api**.

Page Cache contains the generated HTML code of the full page.The cache type codename is **full_page**.

Translations contains files with translation. The cache type codename is **translate**.

Web Services Configuration contains REST and SOAP configurations, generated WSDL file and the Web API Structure. The cache type codename is **config_webservice**.

You can manage the cache in Magento both through the admin panel and the command line.

Though the admin panel. Open *System > Tools > Cache Management*. Here we can disable, enable or update a certain cache type (for this we select the necessary cache type, the necessary action and click "Submit").

Remember that you can't enable and disable the cache through the admin panel in the Production mode.

Moreover, we can clear all Magento cache types using the following buttons:

Flush Magento Cache deletes all elements in the Magento cache (var / cache) by default according to the associated Magento tag and does not affect other processes or applications (equivalent to the *bin/magento cache:clean* command).

Flush Cache Storage removes all elements from the cache regardless of the Magento tag. It clears cache storage, which may affect other application processes that use the same storage (equivalent to the *bin/magento cache:flush* command).

Flush Catalog Images Cache deletes pre-generated product image files.

Flush JavaScript/CSS Cache deletes combined JavaScript and CSS theme files.

Flush Static Files Cache deletes generated static files.

The command line is another cache management option. First, let's see the status of all cache types. Use the following command for this:

bin/magento cache:status

As a result, we will something like this

Current status:
config: 1
layout: 0
block_html: 0
collections: 1
reflection: 0
db_ddl: 0
eav: 0
customer_notification: 0
config_integration: 0
config_integration_api: 0
full_page: 0
translate: 0

config_webservice: 0
0 indicates that this cache type is inactive; **1** - active.

To activate / deactivate a certain cache type, you need to use the following commands, respectively:

bin/magento cache:enable [type]

bin/magento cache:disable [type]

Instead of ***[type]***, we substitute the cche type code name without quotes (see code names for cache types above). If we want to apply the command to several cache types, we indicate their code names separated by spaces. And if we want to apply the command to all cache types, we do not specify any code name.

When we run the *bin/magento cache:enable [type]* command, this cache type is clear automatically.

There are 2 command to clean cache:

bin/magento cache:clean [type]

bin/magento cache:flush [type]

The first command is an equivalent to the **Flush Magento Cache** command in the admin panel, the second one - **Flush Cache Storage** (the effect of both commands was described above).

Cache cleaning is only for active cache types.

10.2 Understand Magento console commands

How do you switch between deploy modes?

Magento has 3 operation modes for deployment in a development or production environment. They are **developer mode**, **production mode** and **default mode**.

Developer mode is only for development. You customize and extend **Magento** here.

1. The symlink of the necessary files is published in the pub / static directory.
2. It has advanced debugging.
3. It has automatic code compilation.

Production mode is for deployment in a production system. All sites that are in this live must have the **production mode**. The **production mode** uses full caching of web pages, which ensures smooth and efficient work for clients on the site.

1. Static files are stored only in the cache.
2. Errors are never displayed to the user since they are registered in the file system.

Default mode usually stands when we launch the site for the first time. This mode is between developer and production mode and is their hybrid. Remember that there are no basic characteristics for production in default mode, so you need to switch to developer mode or production mode. After we switch the default mode to another mode, we won't be able to return to it, because this mode is not intended for long-term use of Magento and site support.

1. A symlink is created in the pub / static directory for each requested file on the site.
2. Customers cannot see errors. Errors are collected in reports, stored and managed on the server..

Switching to developer mode or production mode, we recommend cleaning the following folders, except for .htaccess files (they cannot be deleted, they register configuration changes for each catalog):

```
var/cache
var/di
var/generation
var/view_preprocessed
pub/static
```

Switching between deployment modes in the console is as follows:

```
bin/magento deploy:mode:set {mode} [-s|--skip-compilation]
```

`{mode}` - obligatory parameter(developer or production);
`[-s|--skip-compilation]` - optional parameter(used to skip compilation when switching to developer mode)

```
bin/magento deploy:mode:set developer
```

If everything is correct, the following string will appear in the console:

Enabled developer mode.

To see the current mode, you need to use the command:

```
bin/magento deploy:mode:show
```

We get the following string:

Current application mode: developer.

What bin/magento commands are commonly run during frontend development?

We can also use the following commands during development. To see the entire list of Magento commands enter the location of the Magento 2 files in the root:

```
bin/magento
```

There appear commands with descriptions which perform different functions that can be used by developers. .