

Magento 2 Certified Professional **JavaScript Developer Guide**

Section 1: Technology

1.1 Demonstrate understanding of RequireJS

What is the main purpose of the RequireJS framework?

RequireJS library is a file and module loader that implements **AMD API** ([Asynchronous Module Definition](#)). AMD API is used as a specification? with an aim to define modules such that the module and its dependencies can be asynchronously loaded.

Which capabilities does RequireJS provide to create and customize JavaScript modules?

Below is the example of RequireJS library usage:

```
require(['vendor/library'], function(library) {  
    let text = library.getText();  
    alert(text);  
});
```

Dependencies array, necessary for JS function (in our example, the only vendor/library element) is the first argument, passed into the **require** function. After that, RequireJS passes dependencies objects into the function, which was passed in the second parameter.

This is the example of defining RequireJS module:

```
define([], function(){  
    var library = {};  
    library.getText = function()  
    {  
        return 'Hello World';  
    }  
});
```

```
    }  
    return library;  
  });
```

Define method is very similar to require method. As the first argument, we pass the dependencies array that our module will need (for now, it is empty), and as the second argument - the function that will return the module object. RequireJS does not impose any limitations on the type of the returned value.

By default, RequireJS turns the module name into the path to the file; therefore, if the module name is **vendor/library**, then RequireJS will add the .js extension to the final segment and try to get `/vendor/library.js` file.

To use your own path, specify it in the RequireJS configuration:

```
require.config({  
  baseUrl: '/scripts',  
});
```

After that, RequireJS will search the library at the `/scripts/vendor/library.js` path.

What are the pros and cons of the AMD approach to JavaScript file organization?

The main advantage of RequireJS is that it allows not to worry about how and from where the needed files will be loaded. Instead, the developer will specify the module X as a dependency, and RequireJS will do all the work.

RequireJS also allows to separate JavaScript code into independent modules and load them only when necessary.

The major disadvantage of RequireJS is the necessity to wrap JS code into requirejs methods (define and require). This may particularly come as a hurdle during legacy modules adaptation.

Another con of the AMD approach to JavaScript file organization is a large number of HTTP requests, because each module is loaded separately.

RequireJS in Magento 2

Magento 2 uses RequireJS for loading all the JavaScript code. By default, it contains the RequireJS call and a launch configuration, as well as the mechanism for expanding this configuration.

Out-of-the-box Magento 2 loads RequireJS modules into

`/static/AREA/THEME_VENDOR/THEME_NAME/LOCALE`. This path is also used to publish frontend static assets from Magento modules. Which means that if our module contains, for instance, `app/code/Vendor/Module/view/adminhtml/web/library.js` file, it will be automatically published as

`/static/adminhtml/Magento/backend/en_US/Vendor_Module/library.js` and then can be used as RequireJS module:

```
<script type="text/javascript">
    require('Vendor_Module/library', function(library){
        // custom code
    });
</script>
```

What is a requirejs-config.js file?

The key function of `requirejs-config.js` files is to allow adding changes into RequireJS library configuration.

During `setup:static-content:deploy`, Magento merges all the `require-config.js` module files for corresponding areas and creates a combined file that contains all of the `requirejs` config.

In which cases it is necessary to add configurations to it?

To make an example, we will create a new `requirejs-config.js` file:

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

```
var config = {
  paths:{
    "customLibrary":"Vendor_Module/library"
  }
};
```

This file will be merged into the combined requirejs-config.js file in the following format:

```
(function() {
var config = {
  paths:{
    "customLibrary":"Vendor_Module/library"
  }
};
```

```
require.config(config);
})();
```

This code is created for every requirejs-config.js file, which allows to pass the values, defined in module file, into the combined file, and apply such RequireJS features as aliases or shim.

What tools does it provide?

Aliases

Alias can be created for any requirejs module:

```
require.config({
  paths: {
    "customLibrary": "vendor/library"
  },
});
```

Then, specifying customLibrary in module dependencies will be enough:

```
require(['customLibrary'], function(library) {  
    Let text = library.getText();  
    alert(text);  
});
```

Shim

Many libraries like jQuery, that were developed earlier than RequireJS or AMD standard, have a custom plugin system. Such a system is based on the modification of a single global library object, while additional modules are connected with separate JS files that would interact with this global object.

In the case with RequireJS, the global jQuery object will not be created until the respective RequireJS module is called; otherwise, the jQuery may turn out to be not defined during the load process.

Apply RequireJS configuration in order to add jQuery plugin into Magento 2:

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

```
var config = {  
    paths:{  
        "jquery.plugin":"Vendor_Module/jquery.plugin.min"  
    }  
};
```

This record will create RequireJS module with jquery.plugin name that will refer to the initial file of the plugin itself. Afterward, it can be used in the following way:

```
require(['jquery','jquery.plugin'], function(jQuery, jQueryPlugin){  
    //customCode  
});
```

Although this code will indeed initiate the loading of both files, it will work partially. RequireJS module loading is asynchronous, meaning that a certain file loading order is not specified. If, in certain cases, our plugin is loaded after the core library and will work

correctly, then in other cases it may be loaded and called before jQuery is loaded, leading to errors.
Use shim directive in RequireJS configuration to prevent such errors from happening.

```
var config = {
  paths:{
    "jquery.plugin":"Vendor_Module/jquery.plugin.min"
  },
  shim:{
    'jquery.plugin':{
      'deps':['jquery']
    }
  }
};
```

Shim directive allows to specify for RequireJS that jquery.library module depends on the jquery module and must be loaded only after the jquery loading is complete.

What are global callbacks?

Global callbacks allow you to specify the method one must call after all the dependencies are loaded.

How can mappings be used?

Map directive allows users to selectively change modules and their locations.

What are RequireJS plugins?

RequireJS loader plugins allow to load various dependencies, as well as optimize them if necessary.

Such plugins are marked with ! sign before the module name.

```
require(['loaderPlugin!moduleToBeLoaded'], function(module) {
});
```

RequireJs will load the loaderPlugin module first and then pass it to the module after the ! sign as an argument into the load() method.

What are the text and domReady plugins used for?

Text plugin allows to load text files, for instance, html templates.

```
require(["module", "text!template.html"],
    function(module, html) {
        //html variable will contain text from template.html file
    }
);
```

domReady plugin allows to execute the module code only after DOMContentLoaded browser event.

```
require(['domReady!'], function () {
    //code will be executed only when DOM is ready
});
```

1.2 Demonstrate understanding of UnderscoreJS

Demonstrate understanding of utility functions

UnderscoreJS is a cross-browser library that provides additional functions for working with arrays, objects and functions, templates and more.

UnderscoreJS connects to RequireJS as 'underscore' and is usually used as '_'.

Example:


```
define ([
    'underscore'
], function (_) {
    var max = _.max ([1, 2, 3]);
    return max;
});
```

What are the benefits of using the underscore library versus native javascript?

UnderscoreJS library provides functions for certain common tasks, which greatly simplifies the developers' work, especially in older browsers. However, currently more and more browsers supports native JavaScript functions, which raises the question - which is more superior?

Therefore, if some function requires the support of older browsers, then it is better to rely on UnderscoreJS library. If not, then use native JavaScript; this way, script execution speed will be higher.

Use underscore templates in customizations

The Underscore template compiles JavaScript templates into functions that can be used for rendering.

The example of usage:

`_.template (templateString, [settings])`

Describe how underscore templates are used

`<% = ...%>` substitutes the result of expression calculation

`<% - ...%>` substitutes the HTML escaped calculation result

`<% ...%>` calculates an expression and does not substitute anything

Magento uses underscore templates in the mage/template script. However, it is used less frequently than KnockoutJS.

What are the pros and cons of underscore templates?

Advantages of underscore templates:

- support in older browsers,
- the ability to include logical expressions in templates.

Disadvantages of underscore templates:

- does not support internationalization (i18n),
- does not support observables.

Describe how the underscore templates are used in Magento together with the text RequireJS plugin.

The use of underscore templates we will describe using the Magento_Ui/js/lib/knockout/bindings/tooltip.js file as an example.

```
define ([
    ...
    'mage / template',
    'text! ui / template / tooltip / tooltip.html',
    ...
], function (... , template, tooltipTpl, ...) {
    ...
    template (tooltipTpl, {
        data: config
    })
    ...
})
```

1. RequireJS text plugin connects text resources and is used by the mask **'text!path-to-text-resource'**
2. RequireJS text plugin substitutes the contents of a text resource into a tooltipTmpl parameter
3. mage/template is called with template text and data.
4. mage/template calls the underscore template

Consequently, to use the underscore template with RequireJS in Magento 2, you need to connect the 'mage/template', 'text!path-to-text-resource' and call the mage/template function.

1.3 Demonstrate understanding of jQuery UI widgets

A large part of Magento 2 frontend is built with jQuery Widgets. Using either standard or custom jQuery UI widgets, one can create menus, modal windows, confirmation dialogues, and many other elements.

What is a jQuery library?

A standard jQuery widget call looks the following way:

```
$('#tabs-id').tabs({configObject});
```

Widget system simplifies the process of jQuery plugins development, providing additional ability to control the state of the code. At its core, this is another JavaScript objects system.

jQuery UI widget factory is applied to add new widgets:

```
jQuery.widget("widgetNamespace.widgetName", {  
    _create: function(){  
    },  
});
```

```
doSomething:function(){  
    console.log("Hello World");  
}  
});
```

Afterward, the widget can be created in the following way:

```
$('#element').widgetName({configObject})
```

Namespaces are not used directly from the client code, but allow to avoid collisions with other jQuery plugins. Namespaces are used by jQuery as an array keys and can be viewed by looking up the global jQuery object in browser:

```
console.log(jQuery.widgetNamespace.widgetName)
```

What different components does it have (jQuery UI, jQuery events and so on)?

1. jQuery UI
2. jQuery event system
3. Global AJAX event listeners

How does Magento use it?

Magento 2 strongly depends on the RequireJS library and its modules. Also, nearly all the Magento 2 JavaScript code is initialized with RequireJS modules. This is also applied to jQuery widgets, which can be created in the following way:

```
lib/web/mage/list.js  
define([  
    "jquery",  
    'mage/template',
```

```

    "jquery/ui"
], function($, mageTemplate){
    "use strict";

    $.widget('mage.list', { /*...*/ });
    /*...*/
    return $.mage.list;
})

```

This file determines RequireJS module, which, in its turn, creates a jQuery widget in a conventional way.

It is possible to apply such a widget with the following code:

```

require([
    'jquery',
    'mage/list'
], function($, list){
    $('#element').list({configObject});
})

```

The **list** variable will not be directly applied in this code. Therefore, we need RequireJS to load the module together with the widget, so that we can later use it directly from the jQuery object.

In addition to this, there are two other ways to create jQuery widgets in Magento 2: data-mage-init attribute and x-magento-init script tag.

Therefore, this call

```

<div id="element"
data-mage-init='{ "mage/list": {configObject} }'></div>

```

will be fully equivalent to the one we mentioned above.

This works because RequireJS module, accountable for our widget creation, returns the widget instance after it was created, allowing to use data-mage-init and x-magento-init constructions.

On the other hand, this method of widgets calling complicates their modification. In another system, this would not be a complicated task:

```
jQuery.widget('widgetNamespace.widgetName', {widgetDefinition});

/* ... */

jQuery.widget('widgetNamespace.widgetName',
jQuery.widgetNamespace.widgetName, newDefinition));
```

Until the widget is not called, this action can be performed as much as you need. JQuery even allows to call parent widget methods with `_super` and `_superApply` methods. However, this will not work in Magento 2 if the widget is called using the data-mage-init and x-magento-init constructions, due to the fact that they use the widget instance that returns immediately after it was declared in the original RequireJS module. This means that all the widget extensions will not be used in other RequireJS modules.

It is possible to sidestep this limitation by expanding the widget with RequireJS mixins:

```
app/code/Vendor/Module/view/base/requirejs-config.js
```

```
var config = {
  "config": {
    "mixins": {
      "mage/dropdown": {
        'Vendor_Module/js/dropdownMixin':true
      }
    }
  }
};
```

```
app/code/Vendor/Module/view/frontend/web/js/dropdownMixin.js
```

```

define(['jquery'], function(jQuery){
    return function(originalWidget){
        jQuery.widget(
            'mage.dropdownDialog',
            jQuery['mage']['dropdownDialog'],
            {
                open:function(){
                    return this._super();
                }
            }
        ));

        return jQuery['mage']['dropdownDialog'];
    };
});

```

Therefore, we intercept the original widget call, expand it with the features we need and return the altered version of the widget object to be used by data-mage-init and x-magento-init constructions.

1.4 Demonstrate understanding of KnockoutJS

Describe key KnockoutJS concepts

Knockout is a client-side rendering library that implements the Model-View-ViewModel pattern. The main feature of the library is an automatic UI refresh when the data changes.

Describe the architecture of the Knockout library: MVVC concept, observables, bindings

Model-View-ViewModel (MVVC for short) is a software architectural pattern that serves to separate the graphical user interface development from the backend development. MVVC contains 3 components:

Model contains data,

View displays data on the screen,

ViewModel is an intermediary between Model and View. It allows to receive data for the View and influence the Model.

Bindings

The Knockout library also has a number of bindings that facilitate various interactions between data and elements. Bindings are specified as an attribute of the data-bind html element or as an html comment:

```
<select data-bind="
  options: availableCountries,
  optionsText: 'countryName',
  value: selectedCountry,
  optionsCaption: 'Choose...'"></select>

<!-- ko if: isVisible-->
  <span>Visible</span>
<!-- /ko -->
```

List of bindings:

Appearance:

- visible - show / hide element depending on the condition
- text - display text and html tags as text
- html - display html and render html tags
- css - add css class depending on the condition
- style - add css style depending on the condition
- attr - set attribute values

Flow:

- `foreach` - duplicate the element content for each item in the specified array
- `if` - add the condition block contents to the Document Object Model (later DOM) if they are true
- `ifnot` - add the condition block contents to the DOM if they are false
- `with` - create a new bundling property context in which you can directly address the sub-properties of a certain property.
- `component` - embed an external component in the DOM

Working with form fields:

- `click` - function call when clicked
- `event` - function call when a certain event occurs
- `submit` - function call when *submit*
- `enable` - enable the element when the condition is true
- `disable` - disable the element when the condition is true
- `value` - change the value of the form field
- `textInput` - similar to `value`, it works with text fields only and provides for all types of user input, including autocomplete, drag-and-drop, and clipboard events
- `hasFocus` - change the value of the property when it is received (into *true*) and when the focus is lost (into *false*)
- `checked` - change property value when selecting checkbox or radio elements
- `options` - provide options for dropdown
- `selectedOptions` - provide selected options from the dropdown
- `uniqueName` - set a unique name for form fields with an empty name

Template rendering:

- `template` - render another template

Magento bindings:

- `i18n` - perform a string translation into the current language of the site

Observables

Observables are JavaScript objects that serve to notify subscribers about changes and automatically detect dependencies. Knockout Observables allow to change the values of variables and automatically notify all the subscribers by sending an event.

Example:

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 clicking on the Change button:

The name is alice

Demonstrate understanding of knockout templates

What is the main concept of knockout templates?

Knockout uses the data-bind attribute and HTML comments to calculate expressions. Additionally, Magento provides an alternative syntax for Knockout templates that can be found in the official Magento documentation:

https://devdocs.magento.com/guides/v2.3/ui_comp_guide/concepts/magento-binding-s.html

What are the pros and cons of knockout templates?

Benefits:

- Support for older browsers up to IE 6
- Simple library
- Separating HTML and JavaScript
- Dynamic data binding

Disadvantages:

- Poor performance when the objects are numerous
- Since Magento switches to PWA and uses React, there is a probability that Knockout will sooner or later become deprecated.

Compare knockout templates with underscore JavaScript templates

Underscore template is rendered only when calling the `_.template(template)(params)` method, while the Knockout template is automatically re-rendered if data changes.

Underscore uses `<%= ... %>`, `<% ... %>`, `<%- ... %>` blocks to execute scripts; Knockout, at the same time, uses the `data-bind` attribute in html elements and html comments.

Demonstrate understanding of the knockout-es5 library

Knockout observables are modified the same way as functions and their values are obtained similarly:

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

The knockout-es5 library simplifies the process of obtaining value or altering Knockout observables. The library uses ES5 getter/setter.

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

It is also possible to use the + =, - =, * =, / = operators

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

which is identical to the following expression:

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

In order to apply a plugin to the model, address the ko.track(someModel). Additionally, as the second parameter, it is possible to specify an array of model fields to limit the fields this plugin is applied to:

```
ko.track(someModel, ['firstName', 'lastName', 'email']);
```

Now to get the original observable, you need to call

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

The plugin also allows you to simplify the value of the calculated functions.

For example, in order to write

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

Instead of

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

one needs to call the following code:

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

Section 2: Magento JavaScript Basics

2.1 Demonstrate understanding of the modular structure of Magento

What file structure is used to organize JavaScript modules? Where does Magento locate a module's JavaScript file? Where does Magento store the JavaScript library?

Javascript files in Magento 2 can be located at the following places:

1. **Library level** (*lib/web*) - this directory is used solely for core Magento resources, available for all modules and themes
2. **Module level** (*app/code/Vendor/Module/view/AREA/web*) - also available for all modules and themes
3. **Theme level for a particular module** (*app/design/AREA/Vendor/Theme/Vendor_Module/web*) - available for current and inheriting themes
4. **Theme level** (*app/design/AREA/Vendor/Theme/web*) - available for current and inheriting themes

Describe how static content is organized in Magento

Static content allows to redefine resources by simply creating a file with the same name, but on a level higher according to the fallback scheme. Magento 2 fallback system, in its turn, will search for the requested file following the directories in a certain order. This order depends on the file type and module context availability.

For JavaScript files, as well as style files, images and fonts, the order is the following:

- a. If the module context is unknown
 1. Static files of the current theme for a certain locale (*THEME_DIR/web/i18n/LOCALE/*)

2. Static files of a current theme (*THEME_DIR/web/*)
3. Parent themes until an unparented theme is not found
(*PARENT_THEME_DIR/web/i18n/LOCALE/*, *PARENT_THEME_DIR/web/*)
4. Library files (*lib/web/*)
- b. If the module context is known
 1. Current theme files for locale and module
(*THEME_DIR/web/i18n/LOCALE/Vendor_Module/*)
 2. Current theme files for the module (*THEME_DIR/Vendor_Module/web/*)
 3. Parent themes, until an unparented theme is not found
(*PARENT_THEME_DIR/web/i18n/LOCALE/Vendor_Module/*,
PARENT_THEME_DIR/Vendor_Module/web/)
 4. Module files for the current AREA (*app/code/Vendor/Module/view/AREA/web/*)
 5. Module files for the base area (*app/code/Vendor/Module/view/base/web/*)

How does Magento expose a module's static content to the web requests?

Due to the fact that static resources are used for creating a web page directly and therefore must be available at the user's browser, Magento utilizes static content publishing mechanism. To make static files available externally, Magento publishes them in *pub/static/frontend/VENDOR/THEME/LOCALE/* directory. Afterward, file *app/code/Vendor/Module/view/frontend/web/js/customModule.js* will be available at the path *pub/static/frontend/VENDOR/THEME/MODULE/Vendor_Module/js/customModule.js*.

What are the different ways to deploy static content?

In developer and default modes static content directory is filled on-demand. In production mode, these files are not created automatically and must be deployed with CLI command **setup:static-content:deploy**.

```
bin/magento setup:static-content:deploy [<languages>]
bin/magento setup:static-content:deploy en_US en_GB
```

This command takes the input LOCALES list, for which one must publish static files, and allows to optionally exclude from the publication certain themes, locales, areas and file types.

Additionally, there are several strategies for static resources publication:

1. Standard strategy - in this case, all files from all modules are published.
2. Quick strategy - in this case, only one locale is published for each theme. In other locales, only altered files are published and the rest of them are copied. This strategy decreases deploy time and increases the number of files copied.
3. Compact strategy - allows to avoid file duplicates by copying similar files in the directory base.

2.2 Describe How to use JavaScript modules in Magento

Use requirejs-config.js files to create JavaScript customizations

Requirejs-config.js holds an important place in Magento JavaScript ecosystem. The requirejs-config.js files are collected from the current theme, its parent themes, from all included modules into one file and the link to them is sent to the browser.

Here is an example of requirejs-config.js file Magento (vendor / magento / module-catalog / view / frontend / requirejs-config.js):

```
var config = {
  map: {
    '*': {

      compareList:          'Magento_Catalog/js/list',
      relatedProducts:      'Magento_Catalog/js/related-products',
      upsellProducts:       'Magento_Catalog/js/upsell-products',
      productListToolbarForm: 'Magento_Catalog/js/product/list/toolbar',
      catalogGallery:       'Magento_Catalog/js/gallery',
      priceBox:             'Magento_Catalog/js/price-box',
      priceOptionDate:      'Magento_Catalog/js/price-option-date',
      priceOptionFile:      'Magento_Catalog/js/price-option-file',
```

```

    priceOptions:      'Magento_Catalog/js/price-options',
    priceUtils:        'Magento_Catalog/js/price-utils',
    catalogAddToCart:  'Magento_Catalog/js/catalog-add-to-cart'
  },
  config: {
    mixins: {
      'Magento_Theme/js/view/breadcrumbs': {
        'Magento_Catalog/js/product/breadcrumbs': true
      }
    }
  }
};

```

To get to know all the available options, follow the link
<https://requirejs.org/docs/api.html#config>

How do you ensure that a module will be executed before other modules?

Bear in mind that the order of dependency listing does not correlate with the load order.
 To define the order of execution, one needs to apply Shim option in requirejs-config.js.

For instance, jquery will be loaded before the jquery/jquery-migrate.

```

'shim': {
  'jquery/jquery-migrate': ['jquery'],
},

```

Using define or require, enforce loading of the jquery module before the other module:

```

require(['jquery'], function ($) {
  ...
});

define(['jquery'], function ($)
  return function(...){

```



```
    ...  
  };  
});
```

How can an alias for a module be declared?

The best way to describe this course of action is with the following example:

vendor/magento/module-ui/view/base/requirejs-config.js:

```
var config = {  
  map: {  
    '*': {  
      uiElement:      'Magento_Ui/js/lib/core/element/element',  
      uiCollection:    'Magento_Ui/js/lib/core/collection',  
      uiComponent:     'Magento_Ui/js/lib/core/collection',  
      uiClass:         'Magento_Ui/js/lib/core/class',  
      uiEvents:        'Magento_Ui/js/lib/core/events',  
      uiRegistry:      'Magento_Ui/js/lib/registry/registry',  
      consoleLogger:   'Magento_Ui/js/lib/logger/console-logger',  
      uiLayout:        'Magento_Ui/js/core/renderer/layout',  
      buttonAdapter:   'Magento_Ui/js/form/button-adapter'  
    }  
  }  
};
```

Now it is possible to execute `require(['uiElement'], function(uiElement){ ... })` instead of `require(['Magento_Ui/js/lib/core/element/element'], function(uiElement){ ... })`.

What is the purpose of `requirejs-config.js` callbacks?

A callback is a function to execute after the deps loading. Callbacks are helpful in the situation when `require` is defined as a config object before `require.js` is loaded and one must specify a function to require after the loading of the configuration's deps array.

Describe different types of Magento JavaScript modules

A module is a separate *.js file that can import other require.js modules. It looks the following way:

```
// File (app/code/Vendor/Module/view/frontend/web/js/script.js)
define(['jquery'], function ($)
    return function (...){
        ...
    };
});
//
```

Another module must look the following way to be applied by the first module.

```
define([
    'jquery',
    'Vendor_Module/js/script'
], function ($, myModule)
    return function (...){
        ...
        myModule(...);
        ...
    };
});
```

Plain modules

A plain module is a regular one that is not inherited from another module. You can find an example of a plain module above (Vendor_Module/js/script). In the examples below, we will use modules that are inherited using the jQuery.widget, uiElement.extend functions.

jQuery UI widgets

jQuery widget allows to create a UI widget with a custom handler. Let us consider the following example:

```
vendor/magento/module-multishipping/view/frontend/web/js/payment.js:
define([
    'jquery',
    'mage/template',
    'Magento_Ui/js/modal/alert',
    'jquery/ui',
    'mage/translate'
], function ($, mageTemplate, alert) {
    'use strict';

    $.widget('mage.payment', {
        options: {
            ...
        },

        _create: function () {
            ...
        },
    });

    return $.mage.payment;
});
```

UiComponents

UiComponents enables to create a widget with a custom handler. UiComponents are inherited from uiElement.

Let us examine it using the following file as an example

vendor/magento/module-catalog/view/frontend/web/js/storage-manager.js:

```

define([
    'underscore',
    'uiElement',
    'mageUtils',
    'Magento_Catalog/js/product/storage/storage-service',
    'Magento_Customer/js/section-config',
    'jquery'
], function (_, Element, utils, storage, sectionConfig, $) {
    'use strict';

    ...

    return Element.extend({
        defaults: {
            ...
        },

        initialize: function () {
            ...

            return this;
        },
    });
});

```

2.3 Demonstrate ability to execute JavaScript modules

Magento javascript init methods is a default interface for launching RequireJS modules. It allows to avoid direct JavaScript introduction with using standard <script> tags.

Magento 2 has two basic means for this purpose:

1. **<script type="text/x-magento-init" />** html tag
2. **data-mage-init** html attribute

Both methods allow to initialize RequireJS module, pass JSON values to this module and specify DOM element this module must use.

What is the purpose and syntax of the text/x-magento-init script tag?

To consider in detail how this method works, we can take any module, in which we need to create RequireJS module and phtml template that will utilize this module.

```
app/code/Vendor/Module/view/frontend/web/test.js
```

```
define([], function () {  
    let jsComponent = function(config, node)  
    {  
        console.log(config);  
        console.log(node);  
    };  
  
    return jsComponent;  
});
```

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

```
<div id="example_div">Contents</div>
```

```
<script type="text/x-magento-init">  
    {  
        "#example_div": {  
            "Vendor_Module/test":{"config":"value"}  
        }  
    }  
</script>
```

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

Such script tag is commonly ignored by the browser because browsers do not recognize the type of **text/x-magento-init**. Magento makes use of this behavior

peculiarity and, after the page is loaded, uses JSON-content of such tags for initialization and launch of Magento JavaScript Components. These components are different from common RequireJS modules, for they return the function that Magento can later execute.

The syntax of **x-magento-init** tags' JSON-content looks the following way:

```
{
  "DOM_ELEMENT_SELECTOR": {
    "REQUIREJS_MODULE": CONFIG_OBJECT
  }
}
```

Where **DOM_ELEMENT_SELECTOR** is a CSS selector of DOM-node that will be passed to the component, **CONFIG_OBJECT** is the component's RequireJS name and **CONFIG_OBJECT** is a JSON object generated with PHP.

This initialization method allows to avoid hardcoding of the necessary selectors and configuration parameters in JavaScript or JavaScript generation directly from PHP.

CONFIG_OBJECT and **DOM_ELEMENT_SELECTOR** are passed as parameters into the returned by a RequireJS function component. After the page is loaded in the browser console, the following content will appear:

```
> Object {config: "value"}
> <div id="example_div">Contents</div>
```

Also, if our component does not require DOM element to function, we can call it the following way:

```
<script type="text/x-magento-init">
  {
    "*": {
      "Vendor_Module/test":{"config":"value"}
    }
  }
</script>
```

What is the purpose and syntax of the data-mage-init attribute?

Data-mage-init html attribute is an alternative method for initializing Magento JavaScript Component for a certain DOM element.

How is it used to execute JavaScript modules?

We can modify our phtml file the following way:

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

```
<div data-mage-init='{ "Vendor_Module/test":  
  { "config": "value" } }'>Content</div>
```

The result will be the same. Div with the same attribute and embedded config JSON object will be passed to the function, returned by **Vendor_Module/test** RequireJS module.

It is necessary to use single quotation marks (data-mage-init='...') in this case, because the embedded JSON will be processed in a strict mode that requires JSON to have only double quotation marks.

Also, in case JS component does not return the function, Magento tries to find **REQUIREJS_MODULE** in jQuery prototype, and if such does not exist, it will be called in the following way:

```
$.fn.REQUIREJS_MODULE = function() { ... };  
return;
```

What is the difference between the text/x-magento-init and the data-mage-init methods of JavaScript module execution?

Tag text/x-magento-init can be used to call modules without the reference to a certain DOM element.

How do you execute a JavaScript module in an AJAX response and in dynamic code stored in a string?

To execute this, use imperative call of a JS module:

```
<script>
require([
    'jquery',
    'module'
], function ($) {
    $(function () {
        $('[data-role=example]')
            .accordion({
                header: '[data-role=header]',
                content: '[data-role=content]',
                trigger: '[data-role=trigger]',
                ajaxUrlElement: "a"
            });
    });
});
</script>
```


2.4 Describe jQuery UI widgets in Magento

Describe how Magento uses jQuery widgets, and demonstrate an understanding of the \$.mage object

\$.mage is a namespace for Magento widgets. It uses a set of widgets and functions; below is a list of the most common ones:

1. [Accordion widget](#)
2. [Alert widget](#)
3. [Calendar widget](#)
4. [Collapsible widget](#)
5. [Confirm widget](#)
6. [DropDownDialog widget](#)
7. [Gallery widget](#)
8. [List widget](#)
9. [Loader widget](#)
10. [Menu widget](#)
11. [Modal widget](#)
12. [Navigation widget](#)
13. [Prompt widget](#)
14. [QuickSearch widget](#)
15. [Tabs widget](#)

There is also a number of less popular functions and widgets:

- `__` - look translate
- `cookies.get(name)` - get cookie
- `cookies.set(name, value, options)` - change cookie
- `cookies.clear(name)` - clear cookie
- `dataPost` - a widget that creates post form and sends it to the server
- `dateRange` - date range widget that creates 2 calendar widget
- `escapeHTML(str)` - execute escape HTML
- `formKey` - widget that generates formKey and saves into cookie, in case formKey is absent from the cookie

- `init` - initiates data-mage-init scripts
- `isBetween(value, from, to)` - checks whether there is a number between two other
- `isEmpty(value)` - checks whether the line is empty using trim
- `isEmptyNoTrim(value)` - checks whether the line is empty not using trim
- `isValidSelector(selector)` - checks whether css selector is valid
- `pageCache` - a widget for full page cache that loads private content via an additional ajax query
- `parseNumber(value)` - transforms a line into a number
- `redirect(url, type, timeout, forced)` - executes redirect to the specified url
- `sidebar` - a sidebar widget
- `stripHtml(value)` - deletes html tags from the line
- `translate` - translates the line into the language of the current website locale
- `translateInline` - inline translation widget
- `validation` - form validation widget

What is the role of jQuery widgets in Magento?

jQuery JavaScript library serves to implement client functionality, widely using standard, customized and custom jQuery widgets for this purpose.

A developer has a choice - either to create a new widget from scratch, using the `$.Widget` objects as a base to inherit from, or directly inherit from the existing jQuery UI or third-party widgets. If you name your custom widget the same as the widget you inherited it from, it is possible to extend the initial widget.

How are Magento jQuery widget modules structured?

A widget is created using the `$.widget` method and contains options and ways to work with the widget (`_addClass`, `_create`, `_delay`, `_destroy`, `_focusable`, `_getCreateEventData`, `_getCreateOptions`, `_hide`, `_hoverable`, `_init`, `_off`, `_on`, `_removeClass`, `_setOption`, `_setOptions`, `_show`, `_super`, `_superApply`, `_toggleClass`, `_trigger`, `destroy`, `disable`, `enable`, `instance`, `option`, `widget`).

The following example demonstrates how to create a widget using `vendor/magento/module-multishipping/view/frontend/web/js/payment.js`:

```
define([
    'jquery',
```

```

        'mage/template',
        'Magento_Ui/js/modal/alert',
        'jquery/ui',
        'mage/translate'
    ], function ($, mageTemplate, alert) {
        'use strict';

        $.widget('mage.payment', {
            options: {
                ...
            },

            _create: function () {
                ...
            },
        });

        return $.mage.payment;
    });

```

Describe how Magento executes jQuery widgets

Magento executes Magento jQuerys two ways:

- Connection via RequireJS
- data-mage-init and text/x-magento-init

In order to connect via RequireJS, add a jQuery and the widget into dependencies. Then call the widget using the jQuery object:

```

require(
    [
        'jquery',
        'jquery/validate'
    ]
);

```

```

    ], function ($) {
        $.validator.addMethod("my-email", function(value, element) {
            return this.optional(element) ||
            /^[a-zA-Z0-9.!#$%&'*/+=?^_`{|}~-]+@(?:\S{1,63})$/i.test(value);
        }, 'Please enter a valid email address.');
```

How are Magento jQuery widgets executed with data-mage-init and text/x-magento-init?

To simplify the process of working with widgets in Magento, it is possible to initiate them. There are two alternative ways of initializing widgets:

1. data-mage-init = "..."- attribute, on the basis of which the widget will be initialized
In the example, the tabs widget is initialized.

```

<div class = "product data items" data-mage-init = '{"tabs":
{"openedState": "active"}'>
...
</ div>
```

2. script [type = "text/x-magento-init"] - a script tag that contains the widgets initialization data.

In the example, the Magento_SendFriend/js/back-event widget is initialized for all tags "a" with the role = back attribute.

```

<script type="text/x-magento-init">
{
    "a[role='back']": {
        "Magento_SendFriend/js/back-event": {}
    }
}
</script>
```

2.5 Demonstrate ability to customize JavaScript modules

Describe advantages and limitations of using mixins.

Magento 2 provides the functionality that allows to listen to any RequireJS module after it was initialized and alter or moderate it immediately after initialization and before it is returned. This allows to redefine methods and add new parameters to JavaScript objects, not altering its original code.

```
app/code/Vendor/Module/view/frontend/requirejs-config.js
var config = {
    'config':{
        'mixins': {
            'Magento_Customer/js/view/customer': {
                Vendor_Module/js/customer-mixin:true
            }
        }
    }
};
```

```
app/code/Vendor/Module/view/frontend/web/js/customer-mixin.js
define([], function(){
    'use strict';
    alert('Mixin is working');
    return function(targetModule){
        targetModule.newCustomProperty = value;
        return targetModule;
    };
});
```

Afterward, when we load any page that uses **Magento_Customer/js/view/customer** RequireJS module, we will see our alerts, which means the code was called. Also, when

we look at the **Magento_Customer/js/view/customer** module code, there will be a new parameter **newCustomProperty**.

mixins configuration key in *requirejs-config.js* file is a new parameter that Magento added into RequireJS system. Syntax:

```
'mixins': {  
    'REQUIREJS_MODULE': {  
        MIXIN_MODULE: true  
    }  
}
```

MIXIN_MODULE is a common RequireJS module that will return the function via a single parameter (**targetModule**), and this function is called after the original **REQUIREJS_MODULE**, which is then passed to mixin function. What is returned **MIXIN_MODULE** will be considered as **REQUIREJS_MODULE**.

This way, we can add methods and parameters to RequireJS modules without altering the code directly.

Method override:

```
app/code/Vendor/Module/view/frontend/web/js/customer-mixin.js  
define([], function(){  
    'use strict';  
    alert('Mixin is working');  
    return function(targetModule){  
        targetModule.defaultMethod = function(){  
            //custom replacement code  
        }  
  
        return targetModule;  
    };  
});
```

In case the module returns uiClass-based module, we can use the extend method:

```

app/code/Vendor/Module/view/frontend/web/js/customer-mixin.js
define([], function(){
    'use strict';
    alert('Mixin is working');
    return function(targetModule){
        return targetModule.extend({
            defaultMethod:function()
            {
                var result = this._super(); //call parent method
                //custom code
                return result;
            }
        });
    };
});

```

The disadvantages of this approach manifest themselves only when mixins are used by several developers. In case several mixins are available for one and the same method, one of them will overwrite the results of the rest. Use **mage/utils/wrapper** module to sidestep these constraints. The functionality of this module is very close to Magento 2 around plugins and is used in the following way:

```

app/code/Vendor/Module/view/frontend/web/js/customer-mixin.js
define(['mage/utils/wrapper'], function(){
    'use strict';
    alert('Mixin is working');
    return function(targetModule){
        var customFunction = targetModule.defaultFunction;
        var customFunction = wrapper.wrap(customFunction,
function(original){
    //custom code
    var result = original(); //original method call
    //custom code
    return result;
});

        targetModule.defaultFunction = customFunction ;
    };
});

```

```

        return targetModule;
    };
});

```

The **mage/utils/wrapper.wrap** method receives two arguments - the original method we must change and the function we will change it for. The **original** parameter is the reference of the method, needed to be changed. Wrap method returns the function that will, in its turn, call our method. The original method can be called optionally.

What are cases where mixins cannot be used?

Mixins can not be used in case JS code was called without RequireJS.

How can a new method be added to a jQuery widget?

To add a new method to a jQuery widget, use mixin:

```

app/code/Vendor/Module/view/frontend/requirejs-config.js
var config = {
    "config": {
        "mixins": {
            "mage/dropdown": {
                Vendor_Module/js/dropdown-mixin':true
            }
        }
    }
};

```

```

app/code/Vendor/Module/view/frontend/web/js/dropdown-mixin.js
define(['jquery'], function($) {
    return function(originalWidget) {
        $.widget('mage.dropdownDialog', $['mage']['dropdownDialog'],
        {
            //custom methods

```



```

        customMedhot:function(){
            //custom code
        }
    });

    return $('[mage']['dropdownDialog'];
};
});

```

How can an existing method of a jQuery widget be overridden?

You can also apply mixins to override an existing method of a jQuery widget.

```

app/code/Module/view/frontend/requirejs-config.js
var config = {
    "config": {
        "mixins": {
            "mage/dropdown": {
                'Vendor_Module/js/dropdown-mixin':true
            }
        }
    }
};

```

```

app/code/Module/view/frontend/web/js/dropdown-mixin.js
define(['jquery'], function($) {
    return function(originalWidget) {
        $.widget('mage.dropdownDialog', $('[mage']['dropdownDialog'],
        {
            //overriden methods
            open:function(){
                //custom code
                //parent method call
                return this._super();
            }
        }
    }
});

```

```
    });  
  
    return $('[mage'] ['dropdownDialog'];  
  };  
});
```

What is the difference in approach to customizing jQuery widgets compared to other Magento JavaScript module types?

jQuery widgets by default contain the features that allow to override and modify the widget. All we need to do is to override the widget:

```
//original definition  
jQuery.widget('widgetNamespace.methodName', {/* ... initial method definitions ... */});  
  
/extending widget  
jQuery.widget('widgetNamespace.methodName',  
jQuery.widgetNamespace.methodName,  
  {/*... new method definitions here ...*/});
```

This will work until the widget is not called. The problem is that Magento allows to use widgets with the help of **data-mage-init** and **x-magento-init** constructions. The widgets allow to initialize inline widget, which leads to:

1. Call of the corresponding RequireJS module
2. RequireJS module defines the widget (jQuery.define)
3. RequireJS module returns the newly created widget
4. Magento core applies the newly created widget

The widget is defined and immediately used in the corresponding selector, which makes it hard to redefine or modify.

When extending our widgets, one must bear in mind that our mixin must not only extend the widget with jQuery.widget, but also return the widget instance so that it could be immediately used by **x-magento-init** and **data-mage-init** constructions.

Section 3: Magento Core JavaScript Library

3.1 Demonstrate understanding of the mage library

Describe different types of Magento JavaScript templates

Magento supports the following templates:

1. Underscore
2. Knockout
3. ES6 template

Magento pseudo ES6 literals and underscore templates in Magento

Magento supports ES6 templates. If the browser does not support them, then rendering is performed by the underscore patterns.

ES6 template literals are enclosed by the back-tick (``) character instead of double quotes. But in the Magento template, it is necessary to transfer the pattern enclosed by double or single quotes. Therefore, Magento will automatically apply back-tick if the browser supports ES6 templates.

Describe how to use storage and cookies for JavaScript modules

jQuery Storage API is a plugin that simplifies access to storages (HTML5), cookies, and namespace storage functionality; apart from this, it also facilitates compatibility for old browsers.

To work with jQuery Storage API, connect the **jquery** and **jquery/jquery-storageapi** with RequireJS.

The library contains 3 storages:

1. \$.localStorage
2. \$.sessionStorage
3. \$.cookieStorage

Each storage includes the following methods:

1. `storage.get` - get an item from a storage
2. `storage.set` - set an item in a storage
3. `storage.keys` - get keys of a storage or an item in a storage
4. `storage.isEmpty` - check if a storage or an item in a storage is empty
5. `storage.isSet` - check if an item exists in a storage (if not null or undefined)
6. `storage.remove` - delete an item from a storage
7. `storage.removeAll` - truncate the storage

`$.cookieStorage` includes the following additional methods:

1. `$.cookieStorage.setExpires` - set expires date in days
2. `$.cookieStorage.setPath` - sets path for cookies
3. `$.cookieStorage.setDomain` - set domain for cookies
4. `$.cookieStorage.setConf` - set cookie configuration with an object
5. `$.cookieStorage.setDefaultConf` - sets default configuration

How do you use cookies in the module?

To use cookies in the module, connect jQuery Storage API and call the following methods:

```
$.cookieStorage.get('test_key');  
$.cookieStorage.set('test_key', 'test_value');
```

How is `localStorage` used in Magento?

In `localStorage`, there are objects with such keys as `mage-cache-storage`, `mage-translation-storage`, `product_data_storage`, `recently_viewed_product`, etc. Magento initializes them the following way:

```
storage = $ .initNamespaceStorage ('mage-cache-storage'). localStorage;
```

Afterward, Magento uses `storage` as `$.localStorage`.

Demonstrate the ability to use the JavaScript translation framework

How are CSV translations exported to be available in JavaScript modules?

There are two JS file translation strategies: embedded (js files are translated while publishing) and dictionary (dictionary is generated for dynamic translation).

Embedded:

At the deploy, the contents of JS files are translated into the current locale.

Dictionary:

1. \$.InitNamespaceStorage ('mage-translation-storage'); is initialized;
2. \$.InitNamespaceStorage ('mage-translation-file-version') is initialized;

If \$.localStorage.get ('mage-translation-file-version') is not equal to the current one, then:

- A. Js-translation.json is loaded via RequireJS
- B. The contents of the file are added to the translation database (\$.mage.translate)
- C. The contents of the file are saved in localStorage under the mage-translation-storage name or key
- D. The current version of mage-translation-file-version is saved in localStorage

Otherwise

- a. The contents of mage-translation-storage in localStorage is added to the translation database (\$.mage.translate)

How is a new translation phrase added using JavaScript methods?

In order to add a new translation phrase with JavaScript methods, use jquery and mage / translate into RequireJS and execute \$.mage.__('My text').

If the line uses an argument, then execute the following line \$.mage.__('Hello %1').replace('%1', yourVariable)

Describe the capabilities of the JavaScript framework utils components

mage/utils (lib/web/mage/utils/main.js) adds methods for Underscore.js.

What are the different components available through the mage/utils module?

The additional methods from mage/utils for Underscore.js:

1. arrays
 - a. toggle - Facade method to remove/add value from/to array without creating a new instance
 - b. remove - Removes the incoming value from array in case without creating a new instance of it
 - c. add - Adds the incoming value to array if it's not already present in there
 - d. insert - Inserts specified item into container at a specified position
 - e. formatOffset
2. compare
 - a. equalArrays - Checks if all of the provided arrays contain equal values
 - b. compare
3. misc
 - a. uniqueid - Generates a unique identifier
 - b. limit - Limits function call
 - c. normalizeDate - Converts mage date format to a moment.js format
 - d. inRange - Puts provided value in range of min and max parameters
 - e. submit - Serializes and sends data via POST request
 - f. ajaxSubmit - Serializes and sends data via AJAX POST request
 - g. prepareFormData - Creates FormData object and append this data
 - h. filterFormData - Filters data object. Finds properties with suffix and sets their values to properties with the same name without suffix
 - i. convertToMomentFormat - Converts PHP IntlFormatter format to moment format
 - j. getUrlParameters - Get Url Parameters
4. objects
 - a. nested - Retrieves or defines objects' property by a composite path
 - b. nestedRemove - Removes nested property from an object

- c. flatten - Flattens objects' nested properties
 - d. unflatten - Opposite operation of the 'flatten' method
 - e. serialize - Same operation as 'flatten' method, but returns objects' keys wrapped in '[]'
 - f. extend - Performs deep extend of specified objects
 - g. copy - Performs a deep clone of a specified object
 - h. hardCopy - Performs a deep clone of a specified object. Doesn't save links to original object
 - i. omit - Removes specified nested properties from the target object
 - j. isObject - Checks if provided value is a plain object
 - k. isPrimitive
 - l. forEachRecursive - Iterates over obj props/array elems recursively, applying action to each one
 - m. mapRecursive - Maps obj props/array elems recursively
 - n. removeEmptyValues - Removes empty(in common sense) obj props/array elems
 - o. isEmptyObj - Checks that argument of any type is empty in common sense: empty string, string with spaces only, object without own props, empty array, null or undefined
5. strings
- a. castString - Attempts to convert string to one of the primitive values, or to parse it as a valid json object
 - b. stringToArray - Splits string by separator if it's possible, otherwise returns the incoming value
 - c. serializeName - Converts the incoming string which consists of a specified delimiters into a format commonly used in form elements
 - d. isEmpty - Checks whether the incoming value is empty, e.g. 'null' or 'undefined'
 - e. fullPath - Adds 'prefix' to the 'part' value if it was provided
 - f. getPart - Splits incoming string and returns its' part specified by offset
 - g. camelCaseToMinus - Converts nameThroughCamelCase to name-through-minus
 - h. minusToCamelCase - Converts name-through-minus to nameThroughCamelCase
6. template
- a. template - Applies provided data to the template

Demonstrate the ability to use and customize the validation module

In order to enable form validation, add the data-mage-init = '{"validation" attribute to it: {}' attribute to it.

Ways to add fields to the validation:

1. Add data-validate attribute
2. Add rule name as an attribute
3. Add rule name as a class
4. Add validation inside the data-mage-init attribute of the form tag

How can a custom validation rule be added?

To add a custom validation rule, create a js file and connect it to the page. It is possible to use mixin for connection.

Example:

```
define([
    'jquery',
    'mage/translate',
    'jquery/validate'
], function ($) {
    $.validator.addMethod('my-test', function(){
        return true;
    }, $.mage.__('My test validation error message'));
});
```

Some forms may use Magento_Ui/js/lib/validation/validator instead of jquery / validate. To add a method into it, call the following method: `validator.addRule (name, testFunction, errorMessage);`

How can an existing rule be customized?

The existing rule is customized the same way as adding a new rule; the only difference is that it is added with the same name as the rule that needs to be rewritten.

3.2 Demonstrate understanding of mage widgets

Which collapsible widgets are available in Magento?
How do you use them?

Accordion widget

The widget is applied to display a single part of the content.

Call:

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

The **#element** element must contain child elements together with **data-role="title"** and **data-role="content"** attributes

The full list of parameters can be found here:

https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_accordion.html

Collapsible widget

Turns the content element into an accordion, in which the content is expanded or collapsed then the header is clicked. Unlike the accordion widget, it is called for a single title/content pair.

Call:

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

The full list of parameters can be found here:

https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_collapsible.html

Tabs widget

Allows to display a single area with several tabs. Based on the collapsible widget.

Call:

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

The full list of parameters can be found here:

https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_tabs.html

How do you create a new popup, dialog, or modal with the Magento components?

Alert widget

This is a modal window with a single confirmation button.

Call:

```
$('#element').alert({
    title: 'Warning',
    content: 'Warning content',
    actions: {
        always: function(){} //callback
    }
});
```

The full list of parameters can be found here:

https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_alert.html

Confirm widget

This is a modal window with confirmation and cancel buttons.

Call:

```
$('#confirm_element').confirm({  
  title: 'Confirmation title',  
  actions: {  
    confirm: function() {}, //callback when confirmation  
    cancel: function() {}, //callback when cancellation  
    always: function() {} //is executed each time regardless of  
    the performed action  
  }  
});
```

The full list of parameters can be found here:

https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_confirm.html

Modal widget

This is a configurable modal window with an overlay.

Call:

```
$('#modal_content').modal({});
```

The full list of parameters can be found here:

https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_modal.html

Prompt widget

This is a modal window with an entry field and confirmation/cancel buttons. This widget is based on the modal widget.

Call:

```
$('#prompt_element').prompt({
  title: 'Prompt title',
  actions: {
    confirm: function() {}, //callback upon confirmation
    cancel: function() {}, //callback upon cancellation
    always: function() {} //performed every time, regardless of
    the action taken
  }
});
```

The full list of parameters can be found here:

https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_prompt.html

Which other widgets are available in the Magento JavaScript library? How do you use them?

Calendar widget

This is a calendar/date selection widget, based on jQuery Datepicker widget. It can be used in both popup window or in-line. In addition to this, the widget takes into account time zones.

The full list of parameters can be found here:

https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_calendar.html

Gallery widget

This is an adaptive picture gallery with a picture preview feature, based on Fotorama widget.

Call:

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

```

        "ELEMENT_SELECTOR": {
            "mage/gallery/gallery": {
                "data": [{
                    "thumb": "<small_image_url>",
                    "img": "<small_image_url>",
                    "full": "<small_image_url>",
                    "caption": "<message>",
                    "position": "<number>",
                    "isMain": "<true/false>"
                }]
            }
        }
    }
</script>

```

The full list of parameters can be found here:

https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_gallery.html

Loader widget

The widget blocks the page, partially or completely, during the ajax queries load.

The full list of parameters can be found here:

https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_loader.html

Menu widget

This is the main menu widget, based on jQuery UI Menu widget.

The full list of parameters can be found here:

https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_menu.html

QuickSearch widget

This is a widget for a quick search popup window.

The full list of parameters can be found here:

https://devdocs.magento.com/guides/v2.3/javascript-dev-guide/widgets/widget_quickSearch.htm

3.3 Demonstrate the ability to use the customer-data module

Demonstrate understanding of the customer-data module concept

The customer-data module (Magento_Customer / js / customer-data) enables developers to receive the latest actual data for the current user (private data) from the server. The data is loaded with one Ajax request for all requested sections at once. In order to create a personal data section, one needs to create a class that implements the interface `Magento \ Customer \ CustomerData \ SectionSourceInterface` and which returns private data for this section, and add the similar code in di.xml:

```
<type name="Magento\Customer\CustomerData\SectionPoolInterface">
    <arguments>
        <argument name="sectionSourceMap" xsi:type="array">
            <item name="SECTION_NAME"
xsi:type="string">SECTION_CLASS</item>
        </argument>
    </arguments>
</type>
```

What is private data?

Private data is the information that applies to a certain user. Under this category fall such data, as messages, cart, customer, etc. Magento allows to create custom data type, called section.

Why do we need to store information in the browser?

Storing information in the browser is a great way to speed up the process of website loading, because the information is not loaded from the server.

What are performance considerations?

The data is loaded from cookie and localStorage; this process is very fast. Server synchronization of cookies and localStorage is executed in the background.

Demonstrate understanding of how to use the customer-data module in customizations

How is the customer-data module structured?

Customer-data module adds event listeners to all requests and executes Invalidate, when the request is post, put or delete type:

```
$(document).on('ajaxComplete', function (event, xhr, settings) {
    if (settings.type.match(/post|put|delete/i)) {
        // Invalidate affected sections
    }
});
$(document).on('submit', function (event) {
    if (event.target.method.match(/post|put|delete/i)) {
        // Invalidate affected sections
    }
});
```

In case the module detects that one or several sections are invalidated, it makes a request to the server and loads the latest data for the corresponding sections.

How is data accessed, invalidated, or set?

GET - `run customerData.get(sectionName)`. Returns knockout observable.

Example: `this.cart = customerData.get('cart');`

INVALIDATE - `run customerData.invalidate(sectionNames)` or make post/`put/delete` request

Example: `customerData.invalidate(['cart']);`

SET - `run customerData.set(sectionName, data)`

Example: `customerData.set('messages', {});`

Describe how to use sections.xml to modify the information available through the customer-data module

Sections.xml is used to make invalidate solely for altered sections after the execution of every post, put or delete requests. This way, you omit the repeated requests in every section.

Below is an example of the sections.xml file. The file is assigned to the action the names of the sections that need to be invalidated if this path is requested via post, put, or delete.

```
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Customer:etc/sections.xsd">
    <action name="wishlist/index/cart">
        <section name="wishlist"/>
        <section name="cart"/>
    </action>
</config>
```


How can a sections.xml file be used to add a new section and to modify the invalidation rules of an existing section?

To add invalidation rules, first create an action tag in sections.xml with the name action that is equal to the path of the server request. If this path is requested via post, put or delete, then the sections connected with this path will automatically invalidate.

The process of changing is similar to adding, only you need to add an action with the same name.

How can customization change the way existing sections work?

If customization changes the server request path (for example, adding the product to cart), then the standard Magento settings in sections.xml will not work. You will need to re-create these standard rules for the new path.

Section 4: UI Components

4.1 Demonstrate understanding of Knockout customizations

Describe Magento modifications to the Knockout template engine

Describe the remote template engine, template syntax, custom tags and attributes, and the rendering mechanism

Remote Template Engine

Knockout Remote Template Engine is a custom Knockout Template Engine modification in Magento. It is used for loading remote templates via knockout template binding. Template loading is performed using RequireJS.

Example:

```
<div template="templateUrl"></div>
```

Template syntax and custom tags and attributes

In HTML templates, Magento allows a developer to use a binding syntax that is easier to read and write than the standard Knockout binding syntax. Magento allows to use bindings as stand-alone tags and attributes.

Example of Knockout binding:

```
<!-- ko if: isVisible-->
    Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    Suspendisse vel malesuada augue.
<!-- /ko -->
and
```

```
<div data-bind="if: isVisible">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    Suspendisse vel malesuada augue.
</div>
```

Example of Magento binding:

```
<if args="isVisible">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    Suspendisse vel malesuada augue.
</if>
```

and

```
<div if="isVisible">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    Suspendisse vel malesuada augue.
</div>
```

Rendering mechanism

The following handlers are used to render custom tags and attributes:

1. `renderer.handlers.node` - Basic node handler. Replaces custom nodes with a corresponding knockout's comment tag.
2. `renderer.handlers.attribute` - Base attribute handler. Replaces custom attributes with a corresponding knockouts' data binding.
3. `renderer.handlers.wrapAttribute` - Wraps the provided node with a knockout' comment tag.

Demonstrate the ability to use the custom Knockout bindings provided by Magento

Where can a full list of custom bindings be found?

Follow this link to access a full list of custom bindings

https://devdocs.magento.com/guides/v2.3/ui_comp_guide/concepts/knockout-bindings.html

What are they used for?

This is the list of custom Magento bindings with the description of their functions:

- afterRender - notifies the subscriber as an associated element is inserted into the DOM
- autoselect - automatically highlights the text in an input element, when it gets focus
- bindHtml - renders the string provided, as a collection of HTML elements, inside of the associated node
- collapsible - provides methods and properties required for implementing collapsible panels. It can automatically collapse panel when clicking outside of the associated node, toggle optional CSS class when node changes its visibility. It has additional helper bindings: toggleCollapsible, openCollapsible and closeCollapsible
- datepicker - an adapter for the mage/calendar.js widget
- fadeVisible - performs the gradual change of the element's visibility (with an animation effect)
- i18n - used to translate a string according to the currently enabled locale. Additionally, it creates the necessary elements for the TranslateInline jQuery widget, if it's enabled on the page
- keyboard - allows setting up listeners for the keypress event of a specific key
- magelinit - an adapter for the [data-mage-init] attribute that is used to initialize jQuery widgets on the associated element
- optgroup - a decorator for the standard Knockout's options binding which adds the support of nested options, and renders them as the <optgroup> element

- `outerClick` - allows to subscribe for the “click” event that happens outside of the boundaries of the associated element
- `range` - an adapter for the jQuery UI Slider widget. It also implements necessary handlers to work with mobile devices
- `resizable` - an adapter for the jQuery UI Resizable widget
- `scope` - allows evaluating descendant nodes in the scope of an object found in the `UiRegistry` by provided string
- `staticChecked` - implements the behavior similar to the standard checked binding. The difference is that `staticChecked` doesn't change the array of the already selected elements if the value of the associated DOM element changes
- `template` - a customization of the existing Knockout template binding. It is used to render a template inside of the associated element. The original Knockout's implementation was overridden to support asynchronous loading of templates by the provided path, instead of searching for them on the page
- `tooltip` - displaying a tooltip

What alternatives are there?

Some bindings can be placed inside the `data-bind` attribute, in a separate attribute or tag. Let us consider a part of a binding `i18n` source code as an example:

```
ko.bindingHandlers.i18n = {
  init: function (element, valueAccessor) {
    execute(element, valueAccessor);
  },
  update: function (element, valueAccessor) {
    execute(element, valueAccessor, true);
  }
};
```

```
ko.virtualElements.allowedBindings.i18n = true;
```

```
renderer
  .addNode('translate', {
    binding: 'i18n'
  })
  .addAttribute('translate', {
```

```
        binding: 'i18n'
    });
```

Here `ko.bindingHandlers.i18n` and `ko.virtualElements.allowedBindings.i18n` create binding `i18n`, that can be used in `data-bind`:

```
<div data-bind="i18n: 'Translate as a standard knockout binding'"></div>
```

`renderer.addAttribute('translate')` enables to use `i18n` binding in a separate `translate` attribute:

```
<div translate="Translate using the attribute"></div>
```

`renderer.addNode('translate')` allows to use `i18n` binding in the separate `translate` tag:

```
<translate args="Translate using the tag"></translate>
```

Describe the Knockout scope binding

`uiRegistry` stores connections between the component name and its information.

Components get into `uiRegistry` the following way:

```
<script type="text/x-magento-init">
{
    "*": {
        "Magento_Ui/js/core/app": {
            "components": {
                "COMPONENT_NAME": {
                    "component":
"Magento_Ui/js/lib/step-wizard",
                    "initData": ...,
                    "stepsNames": ...,
                    "modalClass": ...
                }
            }
        }
    }
}
```

```
</script>
```

Afterward, this component can be searched for in uiRegistry by COMPONENT_NAME.

Scope changes the current context for the component context. Here is the example:

```
<div data-bind="scope: 'COMPONENT_NAME'">  
</div>
```

What is the purpose of the scope binding?

Scope binding provides a connection between the part of a template and a JavaScript component.

What Knockout problem does it solve?

Scope binding provides the possibility to initiate components as required instead of loading them beforehand, even if they are not utilized.

How exactly does this binding work?

1. Scope binding handler loads an object from uiRegistry asynchronously
2. The component is loaded with the help of RequireJS
3. A new object of this component is created
4. The component object is applied as ViewModel of the context

Demonstrate the ability to use the scope binding in customizations

How is the scope binding used?

Let us consider the example of using scope:

```
<scope args="requestChild('listing_paging')" render="totalTmpl"/>
```

Where requestChild - creates 'async' wrapper for the specified child using uiRegistry 'async' method and caches it; totalTmpl - the name of the knockout field where the name of the template is stored.

How do nested scopes work?

Each scope has a context, and nested scopes are connected with the parent ones.

How can data of a parent scope be accessed from a child?

1. \$parent - the view model object in the parent context
2. \$parents - an array representing all of the parent view models
3. \$root - the topmost parent context
4. \$parentContext - refers to the binding context object at the parent level

How can the scope binding be applied to HTML in Ajax responses?

You can execute `mage/apply/main.apply()` or `jQuery('body').trigger('contentUpdated')`.
For example:

```
<div id="testBlock">

</div>

<script type="text/javascript">
    require(['jquery', 'mage/apply/main'], function($, mage){
        $.ajax({
            url: 'page.html',
            success: function(content){
                $('#testBlock').html(content);

                // Use one of this:
                $('body').trigger('contentUpdated');
                // or
                mage.apply();
            }
        });
    });
</script>
```


4.2 Demonstrate understanding of Magento UI components

Describe how uiComponents are executed in Magento

What is the difference in uiComponent execution compared to other JavaScript module types? What does it mean to "execute a uiComponent"? Why do we need the app component to execute uiComponents?

Unlike the common JavaScript modules, ui Component consists of XML configuration, JavaScript components and templates.

UI component call is performed by the uiComponent tag in a layout XML file:

```
<referenceContainer name="content">
    <uiComponent name="ui_component_name"/>
</referenceContainer>
```

This tag creates an output HTML code the following construction:

```
<div class="admin__data-grid-outer-wrap" data-bind="scope:
'cms_block_listing.cms_block_listing'">
    <div data-role="spinner"
data-component="cms_block_listing.cms_block_listing.cms_block_columns
" class="admin__data-grid-loading-mask">
        <div class="spinner">

<span></span><span></span><span></span><span></span><span></span><spa
n></span><span></span><span></span>
        </div>
    </div>
```

```
<!-- ko template: getTemplate() --><!-- /ko -->
<script type="text/x-magento-init">
    {"*": {"Magento_Ui/js/core/app": {...uiComponent
configuration object...}}}
</script>
</div>
```

This piece of code calls `Magento_Ui/js/core/app` RequireJS module and passes it the object that contains the necessary UI component configuration, as well as its child components. The object, in its turn, creates the necessary components and registers them in `uiRegistry`.

What is the role of the layout component?

`uiLayout` component is used to create and configure other components. It can also be applied to create dynamic components when the application is already running. Apart from this, it allows creating child UI components.

Describe the structure of a UiComponent

What is `uiClass`? How does it instantiate `uiComponents`?

`uiClass` is a low-level class, from which all the rest UI components are inherited. `uiClass` allows to inherit UI components (extend method) and initialize the configuration (method `initConfig`). In addition to this, this class makes it possible to specify the list of parameters and components interaction.

How can existing component instances be accessed?

`uiRegistry` is used to access existing component instances. This component provides access to all the available component instances.

```
var registry = require('uiRegistry');
var component = registry.get('%componentName%');
```

How can a uiComponent be modified? How do you extend an existing uiComponent?

An existing uiComponent can be extended with the extend method; this method also allows to override the existing methods.

```
define([], function () {  
    'use strict';  
  
    return function (Form) {  
        return Form.extend({  
            initialize: function () {  
                this._super();  
                //custom code  
            }  
        });  
    }  
});
```

This._super call allows to call the parent method a similar way, as ::parent call in PHP.

What is the role of the uiElement and uiCollection modules?

uiCollection module is aimed at base-class components that should contain UI components collection. In other cases, uiElement is more suitable as a base-class.

Demonstrate the ability to create a uiComponent with its own data, or operate with data of existing uiComponents

How does a uiComponent access the data it needs? What are the requirements for a subcomponent to provide data? How can data be loaded by Ajax? How can a component receive the data when it is loaded?

uiComponent accesses the data it needs with DataSource component. It is configured with <dataSource /> tag in the configurational XML UI of the component and has the following parameters:

```
<argument name="dataProvider" xsi:type="configurableObject">
    <argument name="class"
xsi:type="string">Vendor\Module\Ui\DataProvider\ComponentNameDataProvider</argument>
    <argument name="name"
xsi:type="string">ComponentName_data_source</argument>
    <argument name="primaryFieldName"
xsi:type="string">entity_id</argument>
    <argument name="requestFieldName" xsi:type="string">id</argument>
</argument>
```

Where class argument contains PHP class that is aimed at realizing **\Magento\Framework\View\Element\UiComponent\DataProvider\DataProviderInterface** interface and inheriting **\Magento\Ui\DataProvider\AbstractDataProvider** class.

Describe the process of sharing data between components

How can one uiComponent instance access data of another instance?

For this one needs to use uiRegistry component, which provides access to all the component instances available.

```
var registry = require('uiRegistry');
var component = registry.get('%componentName%');
```

How can components communicate while taking into account their asynchronous nature?

For communication, the following UI component parameters can be applied:

- Exports - copies the local value into an external parameter. If the external parameter is a function, it will be called using local value as an argument.
- Imports - applied to track changes in an external parameter.
- Links - synchronizes local and external values, so when you modify one, another is modified as well.
- Listens - tracks the component parameters changes.

4.3 Demonstrate the ability to use UI components

Describe the uiComponents lifecycle

What are the stages of uiComponent execution?

At the server:

1. Layout loads UI component (for instance, your_ui_component_name).
2. Search of a UI component's xml files in every enabled module (for example, MODULE_DIR/view/adminhtml/ui_component/your_ui_component_name.xml).
3. Merge of all the found UI components into a single configuration object.
4. Configuration merge with definition.xml. The objects from definition.xml have lesser priority than a single configuration object, created in the previous step.
5. Conversion of the received configuration into JSON and embedded into the page with the help of Magento\Ui\TemplateEngine\Xhtml\Result class.

As a result, we receive the following piece of code:

```
<script type="text/x-magento-init">
  {
    "*":{
      "Magento_Ui/js/core/app":{
        "types":{ ... },
        "components":{ ... }
      }
    }
  }
</script>
```

At the client side:

1. RequireJS loads `Magento_Ui/js/core/app` and passes the configuration as a parameter.
2. `Magento_Ui/js/core/app` calls `Magento_Ui/js/core/renderer/layout` and passes it the configuration.
3. `layout.js` creates instances of UI components and applies configurations to them.
4. The HTML template is rendered with `knockout.js` and the component binding is applied to the template

What is the role of the layout module, and how does it load components, children, and data?

Layout module (`Magento_Ui/js/core/renderer/layout`) performs the initialization of UI components and applies configurations to them.

1. Layout receives the configuration from `Magento_Ui/js/core/app`
2. All the UI components are processed with the `process` function
 - a. If the parent is not initialized, then `waitParent` is called that asynchronously gets parents from `uiRegistry` and then executes the `process`.
 - b. If a component uses the template and it is not loaded, then `waitTemplate` is executed that asynchronously loads the template and then calls `process`.
 - c. Otherwise
 - i. The `build` function is executed for the component initialization
 - ii. The component is saved in the registry
 - iii. The `process` function is called recursively for all the child elements.

After the Layout module created the instance of this class, it overwrites the properties from the UI component's `defaults` property using properties from the JSON. Then resulting properties become the first-level properties of the newly created UI component's instance, and the original `defaults` property is deleted.

What are the types of components it supports?

The layout module supports every javascript module that returns functions. Commonly, the following modules are implemented:

1. uiClass (is rarely used directly)
2. uiElement (inherited from uiClass)
3. uiCollection (inherited from uiElement)
4. uiComponent (equal to uiCollection)

If there are child elements, it is recommended to use uiCollection. If it is impossible, then use uiElement.

Demonstrate the ability to use uiComponents configuration to modify existing instances and create new instances

A particular instance of a UI component is defined primarily by the following:

1. definition.xml
2. UI component's XML declaration
(https://devdocs.magento.com/guides/v2.3/ui_comp_guide/concepts/ui_comp_xmldeclaration_concept.html)
3. Backend/PHP modifiers
(https://devdocs.magento.com/guides/v2.3/ui_comp_guide/concepts/ui_comp_modifier_concept.html)
4. Configuration inside the JavaScript classes

Describe the definitons.xml file and uiComponent instance XML files.

Definitons.xml

(vendor/magento/module-ui/view/base/ui_component/etc/definition.xml) by default contains UI components configuration.

UiComponent instance XML files (for example, MODULE_DIR/view/adminhtml/ui_component/your_ui_component_name.xml) contains configurations for a certain UI component.

How can you modify an existing instance of a uiComponent using a configuration file?

Create an XML file of a UI component with the same name in your Magento module. The configuration will be united with the UI component file of the original Magento module.

What is the role of the Magento layout in the uiComponent workflow?

UiComponent is embedded into Magento layout using the **uiComponent** tag:

Example of layout xml file:

```
<?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>
        <referenceContainer name="content">
            <uiComponent name="your_ui_component_name"/>
        </referenceContainer>
    </body>
</page>
```

4.4 Demonstrate understanding of grids and forms

Custom Grid

KnockoutJS library is completely responsible for rendering and controlling UI Grid components in Magento 2, but, in addition to it, you will also need a common controller to display a base page.

Controller: *app/code/Vendor/Module/Controller/Adminhtml/Item/Index.php*

```
<?php
namespace Vendor\Module\Controller\Adminhtml\Item;

class Index extends \Magento\Backend\App\Action
```



```

{
    protected $resultPageFactory = false;

    public function __construct(
        \Magento\Backend\App\Action\Context $context,
        \Magento\Framework\View\Result\PageFactory
$resultPageFactory
    )
    {
        parent::__construct($context);
        $this->resultPageFactory = $resultPageFactory;
    }

    public function execute()
    {
        $resultPage = $this->resultPageFactory->create();

$resultPage->getConfig()->getTitle()->prepend((__('Items')));

        return $resultPage;
    }
}

```

Layout, in this case, has a single directive - UI component call.

Layout:

app/code/Vendor/Module/view/adminhtml/layout/vendor_module_item_index.xml

```

<page xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../lib/internal/Magento/Framework/View/Layout/etc/page_configuration.xsd">
    <body>
        <referenceContainer name="content">
            <uiComponent name="vendor_module_item_listing"/>
        </referenceContainer>
    </body>
</page>

```

XML of the UI component has all the necessary data for UI grid rendering: data source specification, the necessary columns list and their types, as well as additional elements, like buttons and filters.

Component layout file:

app/code/Vendor/Module/view/adminhtml/ui_component/vendor_module_item_listing.xml

```
<listing xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Ui:etc/ui_configuration.xsd">
    <argument name="data" xsi:type="array">
        <item name="js_config" xsi:type="array">
            <item name="provider"
xsi:type="string">vendor_module_item_listing.vendor_module_item_listing_data_source</item>
            <item name="deps"
xsi:type="string">vendor_module_item_listing.vendor_module_item_listing_data_source</item>
        </item>
        <item name="spinner" xsi:type="string">spinner_columns</item>
        <item name="buttons" xsi:type="array">
            <item name="add" xsi:type="array">
                <item name="name" xsi:type="string">add</item>
                <item name="label" xsi:type="string" translate="true">Add
New Item</item>
                <item name="class" xsi:type="string">primary</item>
                <item name="url" xsi:type="string">*/*/new</item>
            </item>
        </item>
    </argument>
    <dataSource name="data_source_name">
        <argument name="dataProvider" xsi:type="configurableObject">
            <argument name="class"
xsi:type="string">Magento\Framework\View\Element\UiComponent\DataProvider\DataProvider</argument>
            <argument name="name"
xsi:type="string">vendor_module_item_listing_data_source</argument>
            <argument name="primaryFieldName"
xsi:type="string">item_id</argument>
            <argument name="requestFieldName"
xsi:type="string">id</argument>
```

```

        <argument name="data" xsi:type="array">
            <item name="config" xsi:type="array">
                <item name="component"
xsi:type="string">Magento_Ui/js/grid/provider</item>
                <item name="update_url" xsi:type="url"
path="mui/index/render"/>
                <item name="storageConfig" xsi:type="array">
                    <item name="indexField"
xsi:type="string">item_id</item>
                </item>
            </item>
        </argument>
    </dataSource>
    <columns name="spinner_columns">
        <selectionsColumn name="ids">
            <argument name="data" xsi:type="array">
                <item name="config" xsi:type="array">
                    <item name="resizeEnabled"
xsi:type="boolean">false</item>
                    <item name="resizeDefaultWidth"
xsi:type="string">55</item>
                    <item name="indexField"
xsi:type="string">item_id</item>
                </item>
            </argument>
        </selectionsColumn>
        <column name="item_id">
            <argument name="data" xsi:type="array">
                <item name="config" xsi:type="array">
                    <item name="filter" xsi:type="string">textRange</item>
                    <item name="sorting" xsi:type="string">asc</item>
                    <item name="label" xsi:type="string"
translate="true">ID</item>
                </item>
            </argument>
        </column>
        <column name="name">
            <argument name="data" xsi:type="array">
                <item name="config" xsi:type="array">
                    <item name="filter" xsi:type="string">text</item>

```

```

        <item name="editor" xsi:type="array">
            <item name="editorType"
xsi:type="string">text</item>
            <!-- VALIDATION EXAMPLE -->
            <item name="validation" xsi:type="array">
                <item name="required-entry"
xsi:type="boolean">true</item>
                <!-- CUSTOM VALIDATION EXAMPLE -->
                <item name="custom-validation"
xsi:type="boolean">true</item>
            </item>
        </item>
        <item name="label" xsi:type="string"
translate="true">Name</item>
    </item>
</argument>
</column>
<column name="created_at"
class="Magento\Ui\Component\Listing\Columns\Date">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="filter" xsi:type="string">dateRange</item>
            <item name="component"
xsi:type="string">Magento_Ui/js/grid/columns/date</item>
            <item name="dataType" xsi:type="string">date</item>
            <item name="label" xsi:type="string"
translate="true">Created</item>
        </item>
    </argument>
</column>
<column name="updated_at"
class="Magento\Ui\Component\Listing\Columns\Date">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="filter" xsi:type="string">dateRange</item>
            <item name="component"
xsi:type="string">Magento_Ui/js/grid/columns/date</item>
            <item name="dataType" xsi:type="string">date</item>
            <item name="label" xsi:type="string"
translate="true">Modified</item>
        </item>
    </argument>
</column>

```

```

        </argument>
    </column>
    <!-- IMAGE EXAMPLE -->
    <column name="image"
class="Vendor\Module\Ui\Component\Listing\Column\Image">
        <argument name="data" xsi:type="array">
            <item name="config" xsi:type="array">
                <item name="component"
xsi:type="string">Magento_Ui/js/grid/columns/thumbnail</item>
                <item name="sortable" xsi:type="boolean">>false</item>
                <item name="altField" xsi:type="string">title</item>
                <item name="has_preview" xsi:type="string">1</item>
                <item name="label" xsi:type="string"
translate="true">Image</item>
            </item>
        </argument>
    </column>
</columns>
</listing>

```

Image column

For image column, we need to create

Vendor\Module\Ui\Component\Listing\Column\Image class.

```

<?php
namespace Vendor\Module\Ui\Component\Listing\Column;

use Magento\Catalog\Helper\Image;
use Magento\Framework\UrlInterface;
use Magento\Framework\View\Element\UiComponentFactory;
use Magento\Framework\View\Element\UiComponent\ContextInterface;
use Magento\Store\Model\StoreManagerInterface;
use Magento\Ui\Component\Listing\Columns\Column;

class Image extends Column
{
    const ALT_FIELD = 'title';
}

```

```

protected $storeManager;

public function __construct(
    ContextInterface $context,
    UiComponentFactory $uiComponentFactory,
    Image $imageHelper,
    UrlInterface $urlBuilder,
    StoreManagerInterface $storeManager,
    array $components = [],
    array $data = []
) {
    $this->storeManager = $storeManager;
    $this->imageHelper = $imageHelper;
    $this->urlBuilder = $urlBuilder;
    parent::__construct($context, $uiComponentFactory, $components,
$data);
}

public function prepareDataSource(array $dataSource)
{
    if(isset($dataSource['data']['items'])) {
        $fieldName = $this->getData('name');
        foreach($dataSource['data']['items'] as & $item) {
            $url = '';
            if($item[$fieldName] != '') {
                $url = $this->storeManager->getStore()->getBaseUrl(
                    \Magento\Framework\UrlInterface::URL_TYPE_MEDIA
                ).'{IMAGE_PATH}/'. $item[$fieldName];
            }
            $item[$fieldName . '_src'] = $url;
            $item[$fieldName . '_alt'] = $this->getAlt($item) ?: '';
            $item[$fieldName . '_link'] = $this->urlBuilder->getUrl(
                '{IMAGE_CLICK_LINK}',
                ['yourentity_id' => $item['yourentity_id']]
            );
            $item[$fieldName . '_orig_src'] = $url;
        }
    }

    return $dataSource;
}

```

```

    }

    protected function getAlt($row)
    {
        $altField = $this->getData('config/altField') ?: self::ALT_FIELD;
        return isset($row[$altField]) ? $row[$altField] : null;
    }
}

```

Custom validation

For custom-validation rule to work, we need to create a RequireJs module
`app/code/Vendor/Module/view/adminhtml/web/js/custom-validation.js`

```

require(
    [
        'Magento_Ui/js/lib/validation/validator',
        'jquery',
        'mage/translate'
    ], function(validator, $){
        validator.addRule(
            'custom-validation',
            function (value) {
                //custom validation logic
            }
            ,$.mage.__('Custom Validation Message.')
        );
    });

```

Then connect it to UI Grid on the page.

Custom Ui Form

Same as with UI grid, it requires a controller to render a base page.

Controller:

`app/code/Vendor/Module/Controller/Adminhtml/Index/NewAction.php`

```

<?php
namespace Vendor\Module\Controller\Adminhtml\Index;

class NewAction extends \Magento\Backend\App\Action
{
    const ADMIN_RESOURCE = 'Index';

    protected $resultPageFactory;

    public function __construct(
        \Magento\Backend\App\Action\Context $context,
        \Magento\Framework\View\Result\PageFactory
$resultPageFactory)
    {
        $this->resultPageFactory = $resultPageFactory;
        parent::__construct($context);
    }

    public function execute()
    {
        return $this->resultPageFactory->create();
    }
}

```

Layout:

app/code/Vendor/Module/view/adminhtml/layout/vendor_module_index_edit.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>
        <referenceContainer name="content">
            <uiComponent name="custom_ui_form" />
        </referenceContainer>
    </body>
</page>

```



```
</body>
</page>
```

Ui form component layout specifies the data source for the form, the necessary buttons, tabs, fields and their types.

Ui component definition:

app/code/Vendor/Module/view/adminhtml/ui_component/custom_ui_form.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<form xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Ui:etc/ui_configuration.xsd">
    <argument name="data" xsi:type="array">
        <item name="js_config" xsi:type="array">
            <item name="provider"
xsi:type="string">vendor_module_form.vendor_module_form_data_source</item>
            <item name="deps"
xsi:type="string">vendor_module_form.vendor_module_form_data_source</item>
        </item>
        <item name="label" xsi:type="string" translate="true">Contact
Form</item>
        <item name="config" xsi:type="array">
            <item name="dataScope" xsi:type="string">data</item>
            <item name="namespace"
xsi:type="string">contact_form</item>
        </item>
        <item name="template"
xsi:type="string">templates/form/collapsible</item>
        <item name="buttons" xsi:type="array">
            <item name="back"
xsi:type="string">Vendor\Module\Block\Adminhtml>Contact\Edit\BackButton</item>
            <item name="delete"
xsi:type="string">Vendor\Module\Block\Adminhtml>Contact\Edit\DeleteButton</item>
```

```

        <item name="reset"
xsi:type="string">Vendor\Module\Block\Adminhtml>Contact\Edit\ResetBut
ton</item>
        <item name="save"
xsi:type="string">Vendor\Module\Block\Adminhtml>Contact\Edit\SaveButt
on</item>
    </item>
</argument>
<dataSource name="vendor_module_form_data_source">
    <argument name="dataProvider" xsi:type="configurableObject">
        <argument name="class"
xsi:type="string">Vendor\Module\Model>Contact\DataProvider</argument>
        <argument name="name"
xsi:type="string">vendor_module_form_data_source</argument>
        <argument name="primaryFieldName"
xsi:type="string">vendor_module_contact_id</argument>
        <argument name="requestFieldName"
xsi:type="string">id</argument>
        <argument name="data" xsi:type="array">
            <item name="config" xsi:type="array">
                <item name="submit_url" xsi:type="url"
path="vendor_module/index/save" />
            </item>
        </argument>
    </argument>
    <argument name="data" xsi:type="array">
        <item name="js_config" xsi:type="array">
            <item name="component"
xsi:type="string">Magento_Ui/js/form/provider</item>
        </item>
    </argument>
</dataSource>

<fieldset name="contact">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="collapsible"

```

```

xsi:type="boolean">false</item>
    <item name="label" xsi:type="string"
translate="true">Contact Fieldset</item>
    </item>
</argument>
<field name="vendor_module_contact_id">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="visible"
xsi:type="boolean">false</item>
            <item name="dataType"
xsi:type="string">text</item>
            <item name="formElement"
xsi:type="string">input</item>
            <item name="source"
xsi:type="string">contact</item>
        </item>
    </argument>
</field>
<field name="name">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="label" xsi:type="string">Name</item>
            <item name="visible"
xsi:type="boolean">true</item>
            <item name="dataType"
xsi:type="string">text</item>
            <item name="formElement"
xsi:type="string">input</item>
            <item name="source"
xsi:type="string">contact</item>
        </item>
    </argument>
</field>
<field name="email">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">

```

```

        <item name="label" xsi:type="string">Email</item>
        <item name="visible"
xsi:type="boolean">true</item>
        <item name="dataType"
xsi:type="string">text</item>
        <item name="formElement"
xsi:type="string">input</item>
        <item name="source"
xsi:type="string">contact</item>
    </item>
</argument>
</field>
<field name="comment">
    <argument name="data" xsi:type="array">
        <item name="config" xsi:type="array">
            <item name="label"
xsi:type="string">Comment</item>
            <item name="visible"
xsi:type="boolean">true</item>
            <item name="dataType"
xsi:type="string">text</item>
            <item name="formElement"
xsi:type="string">input</item>
            <item name="source"
xsi:type="string">contact</item>
        </item>
    </argument>
</field>
</fieldset>
</form>

```

Modifying existing UI components

Since UI components utilize the Magento 2-standard XML layout handles system for configuration, adding and overriding is a simple task.

Let us take the example of a UI component, which configuration is located at the `Magento_Catalog/view/adminhtml/ui_component/product_listing.xml` file. To add a

new column, create

app/code/Vendor/Module/adminhtml/ui_component/product_listing.xml file in your module.

app/code/Vendor/Module/adminhtml/ui_component/product_listing.xml

```
<listing xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Ui:etc/ui_configuration.xsd">
    <columns name="product_columns">
        <column name="custom_column">
            <argument name="data" xsi:type="array">
                <item name="config" xsi:type="array">
                    <item name="filter"
xsi:type="string">text</item>
                    <item name="sortable"
xsi:type="boolean">>false</item>
                    <item name="label" xsi:type="string"
translate="true">Custom Column</item>
                    <item name="sortOrder"
xsi:type="number">75</item>
                </item>
            </argument>
        </column>
    </columns>
</listing>
```

Section 5: Checkout

5.1 Demonstrate understanding of checkout architecture

Describe key classes in checkout JavaScript: Actions, models, and views

What are actions, models, and views used for in checkout?

Actions, models and views in checkout implement the MVC pattern and are located in the vendor/magento/module-checkout/view/frontend/web/js folder. They are used to separate data-related logic (Model) from UI-related logic (View) from business logic (Action).

How does Magento store checkout data?

Checkout data (Magento_Checkout/js/checkout-data) is stored in Customer data and localStorage. At the checkout initialization, the Magento_Checkout/js/model/quote module is initialized from which you can get the latest data about the quote. Quote contains knockout observables that you can subscribe to in order to receive up-to-date information.

What type of classes are used for loading/posting data to the server?

Rest API is used for uploading and sending data to the server. Both actions and models execute fetching.

How does a view file update current information?

You need to change the information in two places:

1. checkoutData (Magento_Checkout / js / checkout-data) so that the data remains after the checkout page reloads.
2. quote (Magento_Checkout / js / model / quote) so that the data changes at the current page without reloading.

Demonstrate the ability to use the checkout steps for debugging and customization

How do you add a new checkout step?

Consider the example of creating a checkout step that contains a text field and a button for going to the next step.

Create files for layout, uiComponent and template:

1. Create the uiComponent file:

<module dir>/view/frontend/web/js/step.js

```
define([
    'ko',
    'uiComponent',
    'underscore',
    'Magento_Checkout/js/model/step-navigator'
],
function (ko, Component, _, stepNavigator) {
    return Component.extend({
        defaults: {
            template: 'Rain_Step/step'
        },

        isVisible: ko.observable(true),
        textField: ko.observable(""),

        initialize: function () {
```

```

        this._super();
        stepNavigator.registerStep('test_step', null, 'Test
Step', this.isVisible, _.bind(this.navigate, this), 15);
        return this;
    },

    navigate: function () {

    },

    navigateToNextStep: function () {
        stepNavigator.next();
    }
    });
}
);

```

2. Create the template file:

<module dir>/view/frontend/web/template/step.html

```

<li id="test_step" data-bind="fadeVisible: isVisible">
    <div class="step-title" data-bind="i18n: 'Test Step'"
data-role="title"></div>
    <div id="checkout-step-title" class="step-content"
data-role="content">

        <form data-bind="submit: navigateToNextStep"
novalidate="novalidate">
            <div>
                <textarea class="input-text" rows="3"
data-bind="{value: textField}"></textarea>
            </div>
            <div class="actions-toolbar">
                <div class="primary">
                    <button data-role="opc-continue" type="submit"
class="button action continue primary">
                        <span><!-- ko i18n: 'Next'--><!-- /ko

```



```

--></span>
        </button>
    </div>
</div>
</form>
</div>
</li>

```

3. Create layout file, that adds the newly created uiComponent into the checkout steps:
<module dir>/view/frontend/layout/checkout_index_index.xml:

```


<?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>
        <referenceBlock name="checkout.root">
            <arguments>
                <argument name="jsLayout" xsi:type="array">
                    <item name="components" xsi:type="array">
                        <item name="checkout" xsi:type="array">
                            <item name="children" xsi:type="array">
                                <item name="steps" xsi:type="array">
                                    <item name="children"
xsi:type="array">
                                        <item name="test_step"
xsi:type="array">
                                            <item name="component"
xsi:type="string">Rain_Step/js/step</item>
                                        </item>
                                    </item>
                                </item>
                            </item>
                        </item>
                    </item>
                </argument>
            </arguments>
        </referenceBlock>
    </body>
</page>


```

```
</arguments>
</referenceBlock>
</body>
</page>
```

We get the following result:

Default welcome msg! Sign In or Create an Account

 **LUMA**



✓

Shipping

2

Test Step

3

Review & Payments

Shipping Address

Email Address *

?

You can create an account after checkout.

First Name *

Order Summary

2 Items in Cart ^



Juno Jacket

Qty: 1

View Details ▾

\$77.00

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



Search entire store here...



Test Step

Next

Order Summary

Cart Subtotal \$99.00

Shipping \$10.00

Flat Rate - Fixed

Order Total \$109.00

2 Items in Cart ^



Juno Jacket \$77.00

Qty: 1

How do you modify the order of steps?

To modify the order of steps, change the sortOrder parameter in the stepNavigator.registerStep method (code, alias, title, isVisible, navigate, sortOrder). If you specify sortOrder <10, then the Test Step will be displayed before the Shipping step.



Test Step

[Next](#)

Order Summary

2 Items in Cart



Juno
Jacket

\$77.00

Qty: 1

[View](#)

If sortOrder is more than 10 and less than 20, then it will be reflected between the Shipping step and Review & Payments step



Search entire store here...



Test Step

Next

Order Summary

Cart Subtotal \$99.00

Shipping \$10.00

Flat Rate - Fixed

Order Total \$109.00

2 Items in Cart ^





Juno \$77.00
Jacket


Qty: 1

If sortOrder is more than 20, then it will be displayed after the Review & Payments step

Default welcome msg! Sign In or Create an Account



Search entire store here... 

 2

✓

Shipping

✓

Review & Payments

3

Test Step

Payment Method

Check / Money order


☒ My billing and shipping address are the same

Igor Rain
45 Ridge View Cir Apt 4
Machias, Maine 04654-1054
United States
8080808080

Place Order

Order Summary

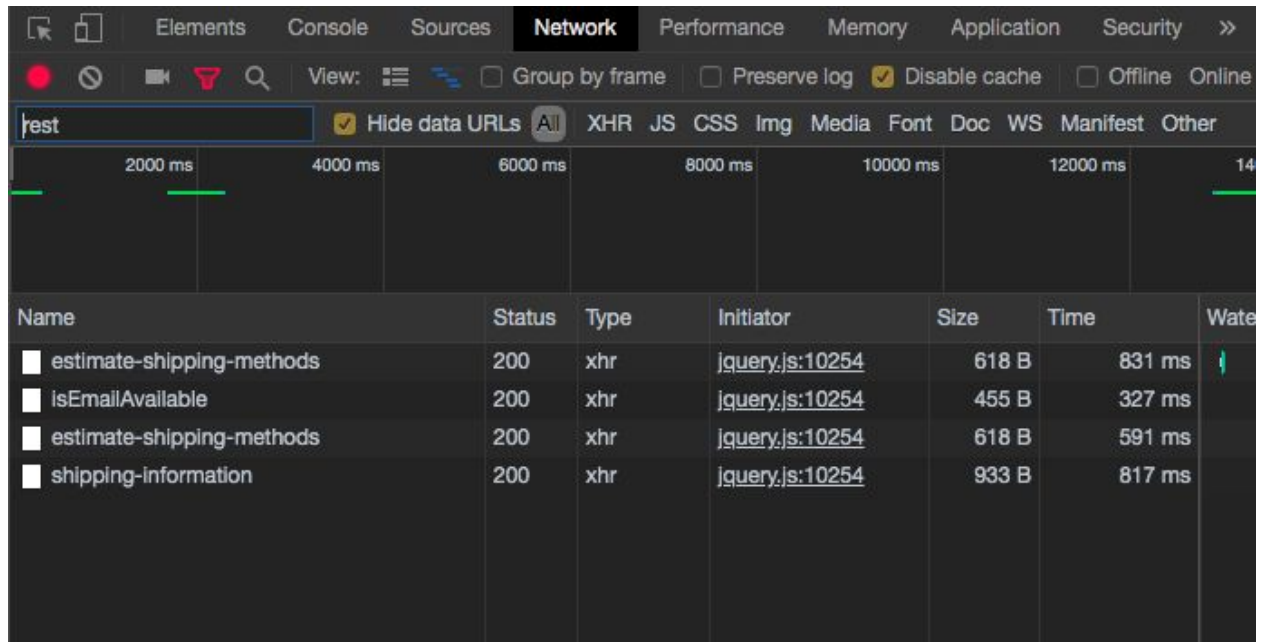
Cart Subtotal	\$99.00
Shipping	\$10.00
Flat Rate - Fixed	
Order Total	\$109.00
2 Items in Cart	▼

Ship To: 

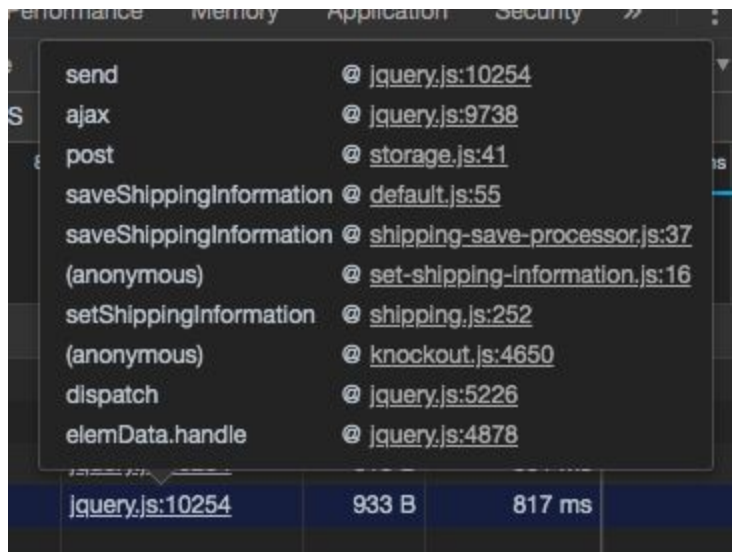
However, the Review & Payments step does not have a Next button, but a Place Order button instead. In order to give the user an opportunity to switch to Test Step from the Review & Payments step, make a modification in the Review & Payments step.

Debug the data flow of each step.

To achieve this, open the Network tab in the Chrome Dev Tools and enter "rest" in the search field.



Then, point the cursor at Initiator and expand the call stack of ajax queries.



How do you customize a step's logic?

You can find the customization instructions at DevDocs.

- [Add a new checkout step](#)
- [Customize the view of an existing step](#)
- [Add a custom payment method to checkout](#)
- [Add custom validations before order placement](#)
- [Add custom shipping carrier validations](#)
- [Add custom input mask for ZIP code](#)

- [Add a custom template for a form field on Checkout page](#)
- [Add a new input form to checkout](#)
- [Add a new field in address form](#)
- [Add custom shipping address renderer](#)

Customize the shipping step rendering and saving

How does Magento save information about the shipping address for different types of checkout (logged in with default address, without default address, not logged in)?

The information is saved in quote after moving from Shipping to Review & Payments step.

In order to save information about the shipping address for different types of checkout, the POST request to REST resource is executed with the `shipping_address`, `billing_address`, `shipping_method_code`, `shipping_carrier_code`, `extension_attributes` information:

- With default address: `/V1/carts/mine/shipping-information`
- Without default address: `/V1/carts/mine/shipping-information`
- Not logged in: `/V1/guest-carts/:cartId/shipping-information`

How does Magento obtain the list of available shipping methods?

POST request to REST resource is executed after shipping address is changed:

- Logged in, for already saved address:
`/V1/carts/mine/estimate-shipping-methods-by-address-id`
- Logged in, for new address: `/V1/carts/mine/estimate-shipping-methods`
- Not logged in: `/V1/guest-carts/:cartId/estimate-shipping-methods`

Which events can trigger this process?

In vanilla Magento 2.3 `estimate-shipping-methods` are called if another shipping address is selected for State/Province, Zip/Postal Code or Country attributes are altered in the current shipping address.

How does Magento save a selected address and shipping method?

As it was mentioned above, Magento calls shipping-information after moving from Shipping to Review & Payments step.

5.2 Demonstrate understanding of payments

Add new payment method and payment methods renderers.

Payment method declaration. Payment.xml file contains method name and type (online or offline) information.

/app/code/Vendor/Module/etc/payment.xml

```
<?xml version="1.0" ?>
<payment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Module:etc/
payment.xsd">
    <groups>
        <group id="offline">
            <label>Custom Payment Method</label>
        </group>
    </groups>
    <methods>
        <method name="custom">
            <allow_multiple_address>1</allow_multiple_address>
        </method>
    </methods>
</payment>
```

Payment method default config. The configuration XML file has the default settings, including the created order status settings, name, active/inactive and PHP class, responsible for back end processing of the order, created by this method.

/app/code/Vendor/Module/etc/config.xml

```
<?xml version="1.0" ?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Store:etc/config.xsd">
    <default>
        <payment>
            <custom>
                <active>1</active>
                <model>Vendor\Payment\Model\Payment\Custom</model>
                <order_status>pending</order_status>
                <title>Custom Payment Method</title>
                <allowspecific>0</allowspecific>
                <group>Offline</group>
            </custom>
        </payment>
    </default>
</config>
```

Backend implementation. PHP class accounts for order backend processing and includes the realization of the basic payment method actions, like authorize, capture and refund.

/app/code/Vendor/Module/Payment/Custom.php

```
<?php
namespace Vendor\Module\Model\Payment;
class Custom extends \Magento\Payment\Model\Method\Cc
{
    public function authorize(\Magento\Payment\Model\InfoInterface
$payment, $amount)
    {
        //authorize logic
    }

    public function capture(\Magento\Payment\Model\InfoInterface
```

```

$payment, $amount)
{
    //capture logic
}

    public function refund(\Magento\Payment\Model\InfoInterface
$payment, $amount)
    {
        //refund logic
    }
}

```

Layout declaration. JS-part of the payment method should be added to the checkout page jsLayout to be processed on the frontend by KnockoutJS. It includes an indication to a new JS module, accountable for the new payment method.

/app/code/Vendor/Module/view/frontend/layout/checkout_index_index.xml

```

<?xml version="1.0" ?>
<page layout="1column"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="urn:magento:framework:View/Layout/etc/page_configuration.xsd">
    <body>
        <referenceBlock name="checkout.root">
            <arguments>
                <argument name="jsLayout" xsi:type="array">
                    <item name="components" xsi:type="array">
                        <item name="checkout" xsi:type="array">
                            <item name="children" xsi:type="array">
                                <item name="steps" xsi:type="array">
                                    <item name="children" xsi:type="array">
                                        <item name="billing-step"
xsi:type="array">
                                            <item name="children"
xsi:type="array">
                                                <item name="payment"
xsi:type="array">
                                                    <item name="children"

```

```
xsi:type="array">
</item name="renders"
xsi:type="array">
    <item
name="children" xsi:type="array">
        <item
name="custom-payment" xsi:type="array">
            <item
name="component"
xsi:type="string">Vendor_Module/js/view/payment/custom</item>
                <item
name="methods" xsi:type="array">
                    <item name="custom" xsi:type="array">
                        <item name="isBillingAddressRequired" xsi:type="boolean">>true</item>
                    </item>
                </item>
            </item>
        </item>
    </item>
</item>
</referenceBlock>
</body>
</page>
```

JS component. The component defines a necessary renderer for payment method display, as well as additional fields processing, if any are available.

```
/app/code/Vendor/Module/view/frontend/web/js/view/payment/custom.js
```

```

define(
    [
        'uiComponent',
        'Magento_Checkout/js/model/payment/renderer-list'
    ],
    function (Component,
        rendererList) {
        'use strict';
        rendererList.push(
            {
                type: 'custom',
                component:
                'Vendor_Module/js/view/payment/method-renderer/custom-method'
            }
        );
        return Component.extend({});
    }
);

```

JS renderer. It is accountable for payment method rendering on the frontend and defines KnockoutJS template, used for this purpose.

/app/code/Vendor/Module/view/frontend/web/js/view/payment/method-renderer/custom-method.js

```

define(
    [
        'Magento_Checkout/js/view/payment/default'
    ],
    function (Component) {
        'use strict';
        return Component.extend({
            defaults: {
                template: 'Vendor_Module/payment/custom'
            },
        });
    }
);

```

```
);
```

KnockoutJs template. It contains basic elements for payment method selection and configuration at the checkout page, as well as its own Place Order button realization.
`/app/code/Vendor/Module/view/frontend/web/template/payment/custom.html`

```
<div class="payment-method" data-bind="css: {'_active': (getCode() ==
isChecked())}">
    <div class="payment-method-title field choice">
        <input type="radio"
            name="payment[method]"
            class="radio"
            data-bind="attr: {'id': getCode()}, value:
getCode(), checked: isChecked, click: selectPaymentMethod, visible:
isRadioButtonVisible()"/>
        <label data-bind="attr: {'for': getCode()}" class="label"><span
data-bind="text: getTitle()"></span></label>
    </div>
    <div class="payment-method-content">
        <!-- ko foreach: getRegion('messages') -->
        <!-- ko template: getTemplate() --><!-- /ko -->
        <!--/ko-->
        <div class="payment-method-billing-address">
            <!-- ko foreach:
$parent.getRegion(getBillingAddressFormName()) -->
            <!-- ko template: getTemplate() --><!-- /ko -->
            <!--/ko-->
        </div>
        <div class="checkout-agreements-block">
            <!-- ko foreach: $parent.getRegion('before-place-order')
-->
            <!-- ko template: getTemplate() --><!-- /ko -->
            <!--/ko-->
        </div>
        <div class="actions-toolbar">
            <div class="primary">
                <button class="action primary checkout"
```

```

        type="submit"
        data-bind="
            click: placeOrder,
            attr: {title: $t('Place Order')},
            css: {disabled: !isPlaceOrderActionAllowed()},
            enable: (getCode() == isChecked())
        "
        disabled>
        <span data-bind="i18n: 'Place Order'"></span>
    </button>
</div>
</div>
</div>
</div>

```

Modify an existing payment method.

You can modify an existing payment method using Magento JS mixins to modifying/adding the existing methods. You can also completely replace the model or renderer by changing the jsLayout parameter through layout XML files, or by using the plug-in to \Magento\Checkout\Block\Checkout\LayoutProcessor::process.

How does a payment method send its data to the server?

Magento 2 API method rest/default/V1/guest-carts/payment-information or rest/default/V1/carts/mine/payment-information is used to send payment method data to the server. The choice between the two depends on whether the current customer a guest or is logged-in to the store.

Example request:

```

{
    "cartId": "CART_ID",
    "billingAddress": {
        "countryId": "CODE",
        "regionCode": null,
        "region": null,
    }
}

```

```

    "customerId": "CUSTOMER_ID",
    "street": [
      "STREET_1",
      "STREET_2"
    ],
    "telephone": "PHONE",
    "postcode": "POSTCODE",
    "city": "CITY",
    "firstname": "FIRSTNAME",
    "lastname": "LASTNAME",
    "saveInAddressBook": null
  },
  "paymentMethod": {
    "method": "PAYMENT_METHOD_CODE",
    "additional_data": {
      "cc_cid": "",
      "cc_ss_start_month": "",
      "cc_ss_start_year": "",
      "cc_ss_issue": "",
      "cc_type": "",
      "cc_exp_year": "",
      "cc_exp_month": "",
      "cc_number": "",
      "is_active_payment_token_enabler": false,
      "alias": ""
    }
  }
}

```

What is the correct approach to deal with sensitive data?

Local storage is unsuitable for storing sensitive data, for user browsers can be easily compromised. Server, in its turn, is also not the best choice of sensitive data storage place. Therefore, the most correct approach is to integrate with a payment provider and further application of tokenization to reference a payment.

Describe the data flow during order placement

Each available payment method renderer has its own Place Order button realization. Click binding at this button leads to the payment method model, where the method is inherited from `Magento_Checkout/js/view/payment/default` module.

/vendor/magento/module-checkout/view/frontend/web/js/view/payment/default.js

```
placeOrder: function (data, event) {
    var self = this;

    if (event) {
        event.preventDefault();
    }

    if (this.validate() && additionalValidators.validate()) {
        this.isPlaceOrderActionAllowed(false);

        this.getPlaceOrderDeferredObject()
            .fail(
                function () {
                    self.isPlaceOrderActionAllowed(true);
                }
            ).done(
                function () {
                    self.afterPlaceOrder();

                    if (self.redirectAfterPlaceOrder) {
                        redirectOnSuccessAction.execute();
                    }
                }
            );

        return true;
    }

    return
```

The method contains several verifications and a `getPlaceOrderDeferredObject` call that returns jQuery ajax request.

/vendor/magento/module-checkout/view/frontend/web/js/action/place-order.js

```
function (quote, urlBuilder, customer, placeOrderService) {
    'use strict';

    return function (paymentData, messageContainer) {
        var serviceUrl, payload;

        payload = {
            cartId: quote.getQuoteId(),
            billingAddress: quote.billingAddress(),
            paymentMethod: paymentData
        };

        if (customer.isLoggedIn()) {
            serviceUrl =
urlBuilder.createUrl('/carts/mine/payment-information', {}));
        } else {
            serviceUrl =
urlBuilder.createUrl('/guest-carts/:quoteId/payment-information', {
                quoteId: quote.getQuoteId()
            });
            payload.email = quote.guestEmail;
        }

        return placeOrderService(serviceUrl, payload,
messageContainer);
    };
}
```

Afterward, `PlaceOrderAction` module creates API Payload, which differs depending on whether the current customer is logged in, and calls `placeOrderService`.

/vendor/magento/module-checkout/view/frontend/web/js/model/place-order.js

```

define(
    [
        'mage/storage',
        'Magento_Checkout/js/model/error-processor',
        'Magento_Checkout/js/model/full-screen-loader'
    ],
    function (storage, errorProcessor, fullScreenLoader) {
        'use strict';

        return function (serviceUrl, payload, messageContainer) {
            fullScreenLoader.startLoader();

            return storage.post(
                serviceUrl, JSON.stringify(payload)
            ).fail(
                function (response) {
                    errorProcessor.process(response,
messageContainer);
                    fullScreenLoader.stopLoader();
                }
            );
        };
    }
);

```

There, Magento 2 API URL is called that either creates an order or returns the error, further processed into various done and fail callbacks.

Which modules are involved?

Magento_Checkout, Magento_Payment, Magento_Sales, as well as payment systems' modules.