OXID eShop developer documentation

Release 1.0.0

OXID eSales AG

Oct 26, 2017

Contents

This is OXID eShop developer documentation. It should provide necessary up to date information for developers.

Currently this repository contains skeleton for documentation which should become useful documentation for developers.

Conventions how to write documentation can be found *here*.

CHAPTER 1

Table of Contents

Getting started

This section describes how to prepare environment to start develop using OXID eShop.

Environment preparation

Documentation which describes how to prepare OXID eShop for development purposes can be found here: https://github.com/OXID-eSales/oxvm_eshop

Install OXID eShop compilation

Please, install the OXID eShop compilation performing the following steps:

Step 1: Deploy source code and install project dependencies

The recommended way to obtain the source code of OXID eShop and to install the project dependencies is to use Composer. You will find details how to install and use Composer here. Please make sure to have a sufficient understanding of how Composer works before proceeding.

If by any reason you are not able to use Composer to install OXID eShop or one of its modules on a specific application server, please skip this step and read *these instructions* to learn how to deploy the source code using an alternative way.

Depending on the edition of OXID eShop you want to install, run one the following commands in the command line interface of your operating system.

If you install OXID eShop for module development, remember we recommend using oxVM for development, but if by any reason you need the OXID eSales development tools to be installed, remove the *–no-dev* option in the commands below.

Note: For OXID eShop Professional Edition or OXID eShop Enterprise Edition, you need to enter the credentials you should have received when purchasing the product.

- For Community Edition:

```
composer create-project --no-dev oxid-esales/oxideshop-project your_project_
↪name dev-b-6.0-ce
```

- For Professional Edition:

```
composer create-project --no-dev oxid-esales/oxideshop-project your_project_
↪name dev-b-6.0-pe
```

- For Enterprise Edition:

```
composer create-project --no-dev oxid-esales/oxideshop-project your_project_
↪name dev-b-6.0-ee
```

When the Composer has finished successfully, a new directory will have been created in your working directory. It is called *your_project_name* in this example and it is referred to as *project root directory*.

The *project root directory* contains all files, which are needed to continue with the installation of OXID eShop.

Watch out for error messages during the installation progress.

See our *troubleshooting* section for solutions.

Note: If you install the compilation **without** the *–no-dev* option, the following development tools will be installed together with OXID eShop:

- OXID eShop Testing Library
- IDE code completion support for OXID eShop
- OXID Coding Standards
- Azure Theme for selenium tests

Technical details

Composer will automatically download the source files of the specified version and edition of OXID eShop.

In a second step it will install fixed versions of the project dependencies as defined in the meta package of the installed edition of OXID eShop.

After Composer installed all dependencies, it executes several tasks. One of them is to generate the classes of the *unified namespace* *\OxidEsales\Eshop*.

Step 2: Configure the HTTP server

Move the *project root directory* to a directory accessible by your HTTP server. Configure the servers public document root to point to the *source* directory of the *project root directory*

Step 3: Adapt file and directory permissions

The following directories and its subdirectories must always be writable by the HTTP server during the run time:

- ./source/export
- ./source/log/
- ./source/out/pictures/
- ./source/out/media/
- ./source/tmp/

For the next step, the graphical setup, the following files and directories must be writable for the HTTP server:

- ./source/Setup
- ./source/config.inc.php
- ./source/.htaccess

Note: In a development environment, the easiest way to adapt permissions, is to run

```
sudo chmod 777 -R source/config.inc.php source/.htaccess source/tmp/ source/
↪log/ source/out/pictures/ source/out/media/ source/export
```

Step 4: Run the graphical setup

Open **http(s)://<your shop URL>/Setup** in your browser and follow the instructions of the graphical setup.

At the end of the installation process, the directory ./source/Setup is deleted.

After the graphical setup, please set the following files to read-only for the HTTP server:

- ./source/config.inc.php
- ./source/.htaccess

Note: As the file ./source/config.inc.php contains database credentials, you should consider to restrict read access to the HTTP server.

Activate pre-installed modules

None of the bundled modules is activated by default during the setup. Please refer to the documentation you find inside the module directory about system requirements and configuration of each module.

Install more modules and module dependencies

After the installation, you may proceed with the installation of some of the many modules the OXID eco system provides. Refer to the installation instructions of each of the modules.

Keep in mind that some OXID eShop modules may have special requirements, which may go beyond the system requirements of a standard installation of OXID eShop. These requirements may either be installable via Composer or may require the installation of certain PHP extensions or even system libraries. In any case, the authors of the modules will have provided you with all necessary information about these requirements and how to install them on your application server.

Known issue on MacOS

If you get the following error in the migrations while installing the OXID eShop on a MAMP [PDOException] SQLSTATE[HY000] [2002] No such file or directory

Look at this blog entry and do the following steps:

```
sudo mkdir /var/mysql
sudo ln -s /Applications/MAMP/tmp/mysql/mysql.sock /var/mysql/mysql.sock
sudo chown _mysql /var/mysql/mysql.sock
sudo chmod 777 /var/mysql/mysql.sock
```

Hints for development

Always use Composers' –no-plugins switch

It is a good practice to run all Composer commands, which update components with the –no-plugins option and to run update action in a separate command. Like this it is ensured, that the latest versions of the plugins are used.

Examples:

```
# Update all components including Composer plugins to their latest version
composer update --no-plugins

# execute plugins in their latest version
composer update
```

```
# Install new component and update dependencies including Composer plugins␣
↪to the required version
composer require --no-plugins monolog/monolog
composer install # execute the plugins in their required version
```

Temporarily add Composer dependencies

In general you should extended the functionality of OXID eShop by writing modules, which provide there own dependency management. See *module section* for details. Nevertheless, for a quick hack or a proof of concept, additional dependencies could be added via the composer.json file in the *project root directory*.

For example, if there is a need to add runtime library like monolog run:

```
composer require --no-plugins monolog/monolog
composer install
```

If there is a need to add a development dependency like the OXID eShop testing library:

```
composer require --dev --no-plugins oxid-esales/testing-library:dev-master
composer update
```

Resolving Composer dependency conflicts

The meta package defines, which exact versions of the components will be installed by Composer. These versions have been tested by OXID eSales to ensure, that OXID eShop works as expected and to avoid security issues. There might be situations, where a 3rd party dependency conflicts with the version defined in the meta package. You may resolve this version conflict by adding an alias in the project composer.json file in the *project root directory* like this:

```
{
    "require": {
        "doctrine/cache":"v1.6.0 as v1.6.1"
    }
}
```

This lowers doctrine cache version to v1.6.0 even while the meta package requires v1.6.1.

See the documentation or this issue in GitHub for details

Building your own compilation

A meta package defines the kind and versions of components of a compilation. You may want build your own compilation for two reasons:

To re-define the components of a compilation:

- Create a new meta package by using the existing one as a template
- Re-define the components and their versions
  - Require different versions of existing components
  - Remove predefined components
  - Require new components

To add new components to the compilation:

- Create a new meta package
- Require new components
- Require the existing meta package in the newly created meta package

Make this new meta package available through Packagist, GitHub, file system or any other supported way.

Edit the composer.json file in the *project root directory* and require the new meta package instead of default one.

Install OXID eShop compilation on servers, where Composer is not available

We strongly recommend to *install OXID eShop via Composer* on the application server! But if Composer is not available e.g. on a "shared hosting account" or in a high-security environment, it is still possible to install or deploy

an OXID eShop. This solution requires more effort and also some knowledge about Composer, as you will run the Composer commands on one machine and then copy the files over to the application server.

The process is roughly:

- *Setup a local environment*
- *Deploy the source code and install project dependencies in the local environment*
- *Prepare the generated files for deployment on the remote server (UNIX-based only)*
- *Copy the files to the application server and continue installation*
- *Managing modules and module dependencies*

Setup a local environment

As a first step setup a local environment. For the sake of simplicity we call this environment *local*, but it can also be a remote machine, a docker container, a virtual machine or any other installation where you have sufficient access rights to install and run executables.

This local environment should be as similar as possible to the server, where OXID eShop should finally be installed or deployed. Especially the PHP stack and the required system libraries should be identical to the stack of the application server. Keep in mind that even differences in patch versions may matter. So it is really important to keep the local environment and the application server in sync in order be able to copy files from one system to another. Failing to do so may lead to hard to detect errors during the runtime of OXID eShop.

Make sure to have a working installation of Composer on this local environment. You will find details how to install and use Composer here. Please make also sure to have a sufficient understanding of how Composer works before proceeding.

Deploy the source code and install project dependencies in the local environment

On your local environment follow the installation instructions, section *Step 1: Deploy source code and install project dependencies*. After this step has been completed without errors, you will find a new directory in your current directory. This new directory is called *your_project_name* in the example, but you may have chosen a different name. This directory will be referred to as *project root directory*.

Prepare the generated files for deployment on the remote server (UNIX-based only)

Users of Windows servers can skip this step, as Composer does not create symbolic links on Windows based systems.

On UNIX based systems, Composers creates symbolic links in the directory *project root directory*/vendor/bin/, which cannot be just copied to a remote system like plain files.

There are at least 2 possible solutions to overcome this issue:

1. In case you have *shell access* to the remote server, you should consider archiving the files using the *tar*-command:

```
# create tar archive in the local environment
tar -cvzf oxid-eshop.tar.gz <project root directory>
```

Like this, also the symbolic links are extracted on the application server. Use this command to extract the tar archive on the application server

```
# extract tar archive on the application server
tar -xvzf oxid-eshop.tar.gz
```

2. In case you have *no shell access* to the remote server, you have to delete the symbolic links and to manually create alternative files, which have to be copied to the application server.

```
cd <project root directory>
rm vendor/bin/*

cat << 'EOF' >> vendor/bin/oe-eshop-db_views_generate
#!/usr/bin/env sh

dir=$(d=${0%[/\\]*}; cd "$d" > /dev/null; cd "../oxid-esales/oxideshop-db-
↪views-generator" && pwd)

dir=$(echo $dir | sed 's/ /\ /g')
"${dir}/oe-eshop-db_views_generate" "$@"
EOF

cat << 'EOF' >> vendor/bin/oe-eshop-demodata_install
#!/usr/bin/env sh

dir=$(d=${0%[/\\]*}; cd "$d" > /dev/null; cd "../oxid-esales/oxideshop-
↪demodata-installer/bin" && pwd)

dir=$(echo $dir | sed 's/ /\ /g')
"${dir}/oe-eshop-demodata_install" "$@"
EOF

cat << 'EOF' >> vendor/bin/oe-eshop-doctrine_migration
#!/usr/bin/env sh

dir=$(d=${0%[/\\]*}; cd "$d" > /dev/null; cd "../oxid-esales/oxideshop-
↪doctrine-migration-wrapper/bin" && pwd)

dir=$(echo $dir | sed 's/ /\ /g')
"${dir}/oe-eshop-doctrine_migration" "$@"
EOF
```

Copy the files to the application server and continue installation

Copy the *project root directory* to your application server and set all files in the vendor/bin directory to be executable. Then finish the installation on the application server starting with Step 2 of the *standard installation instructions*.

Managing modules and module dependencies

Some OXID eShop modules are installable via Composer or may require some 3rd party components (e.g. monolog/monolog) to be installed via Composer.

To install these modules or their dependencies, follow the same strategy: Install them in a local environment following the installation instructions of the module and then copy the newly installed files to the application server.

All files, which are managed by Composer live inside a subdirectory of *project root directory* called *vendor*. The contents of this directory and all its subdirectories may completely change with every execution of **`composer require`** or **`composer update`**, so it is a good practice to always completely replace this directory on the server.

Continue the installation procedure (copy modules files, configure module, etc.) on the application server.

Troubleshooting

I am asked for a github token

By default github has API access limits set for anonymous access. In order to overcome these limits one has to create a github token, which could be done as described in: https://help.github.com/articles/creating-an-access-token-for-command-line-use/

I get a Composer\Downloader\TransportException

During the installation of OXID eShop Professional or Enterprise Edition you get the following error:

```
[Composer\Downloader\TransportException]
Invalid credentials for 'https://enterprise-edition.packages.oxid-esales.com/packages.
↪json', aborting.
```

You may have stored some outdated or wrong credentials. Please review <your home directory>/.composer/auth.json and delete the section, which begins with "professional-edition.packages.oxid-esales.com" resp. "enterprise-edition.packages.oxid-esales.com"

I am asked "Do you want to remove the existing VCS (.git, .svn..) history? [Y,n]"

In general you can say "Yes". It is not normally important to keep VCS history locally. You can always look it up on github.

There was an error during the execution of the unified namespace generator

If the generation of the *unified namespace classes* fails, OXID eShop will not run properly. In this case you should look *here* for possible fixes.

What's next?

Module resources

All the information for backend developer can be found under *module section*. You'll find all the necessary information to understand how modules works and how to write them.

Theming resources

Information for creating or extending OXID eShop theme, can be found in *themes section*.

OXID eShop components

Migrations

OXID eShop uses database migrations for eShop setup and updates. Migration tool can be used for project specific migrations too.

Infrastructure

At the moment OXID eShop uses "Doctrine 2 Migrations" and it's integrated via OXID eShop migration components.

Doctrine Migrations runs migrations with a single configuration. There is a need to run several configurations (suites) of migrations for OXID eShop project. For example one for Community Edition, one for Enterprise Edition and one for a project. For this reason OXID eShop Doctrine Migration Wrapper was created.

Doctrine Migration Wrapper needs some information about the OXID eShop installation like:

- what edition is active
- what are credentials for database

This information is gathered from OXID eShop Facts. Facts has a class which can provide an information about OXID eShop and it's environment. This component is Shop independent and can be used without bootstrap. The only restriction is to have config.inc.php file configured.

Usage

Running migrations - CLI

Script to run migrations is registered to composer bin directory. It accept two parameters:

- Doctrine command
- Edition

```
vendor/bin/oe-eshop-db_migrate migrations:migrate
```

This command will run all the migrations which are in OXID eShop specific directories. For example if you have OXID eShop Enterprise edition, migration tool will run migrations in this order:

- Community Edition migrations
- Professional Edition migrations
- Enterprise Edition migrations
- Project specific migrations

In case you have Community Edition:

- Community Edition migrations
- Project specific migrations

It is also possible to run migrations for specific suite by defining environment variable - **MIGRATION_SUITE**. This variable defines what type of migration it is. There are 4 types:

- **PR** - For project specific migrations. It should be always used for project development.
- **CE** - Generates migration file for OXID eShop Community Edition. **It's used for product development only**.

- **PE** - Generates migration file for OXID eShop Professional Edition. **It's used for product development only**.
- **EE** - Generates migration file for OXID eShop Enterprise Edition. **It's used for product development only**.

Generate migration

```
vendor/bin/oe-eshop-db_migrate migrations:generate
```

This command will create shop views by current eShop version, edition and configuration. It is a good practice to run it right after migrations command.

Generate migration for a single suite

```
vendor/bin/oe-eshop-db_migrate migrations:generate PR
```

This command will generate new migration. Migration class will be generated to specific directory according **MIGRATION_SUITE** variable. In this case it will be generated in *source/migration/project_data/* directory.

Run Doctrine 2 Migrations commands

Sometimes there will be a need to run doctrine specific commands. To do so run Doctrine Migrations command:

```
vendor/bin/oe-eshop-db_migrate DOCTRINE_COMMAND
```

For example, you would like to get the list of doctrine migrations available commands:

```
vendor/bin/oe-eshop-db_migrate
```

More information on how to use Doctrine 2 Migrations can be found in official documentation page: http://docs.doctrine-project.org/projects/doctrine-migrations/en/latest/

Using Doctrine Migrations Wrapper

Doctrine Migration Wrapper is written in PHP and could be used without command line interface. To do so:

- Create `Migrations` object with `MigrationsBuilder->build()`
- Call `execute` method with needed parameters

Unified Namespace Generator

See github-repository

This component is responsible for creating the classes of the namespace `OxidEsales\Eshop` which are called *unified namespace classes*.

When do the unified namespace classes get generated?

The unified namespace generator implements a composer plugin and a standalone script. It generates the unified namespace classes on the fly, e.g. when you install or update the OXID eShop:

The generation of unified namespace classes is triggered by running

- **composer create-project** with the OXID eShop metapackage
- **composer install**
- **composer update**. If you want to be sure, to get no errors because of an old version of the unified-namespace-generator, first run **composer update --no-plugins --no-scripts** and afterwards **composer update**. If you directly execute first **composer update**, you may encounter errors. In this case, run again **composer update** and the errors should go away.
- **composer require**. If you want to be sure, to get no errors because of an old version of the unified-namespace-generator, first run **composer require --no-update** and afterwards **composer update**.
- **reset-shop**
- by manually executing **vendor/bin/oe-eshop-unified_namespace_generator**

Mode Of Operation

Given the example you run the following command:

```
composer create-project --no-dev oxid-esales/oxideshop-project my_oxid_eshop_project␣
↪dev-b-6.0-ce
```

By triggering the generation with other commands the steps 1 and 2 can be different.

1. Download and install all libraries to the folder *vendor*
2. oxideshop-unified-namespace-generator is executed by the composer event POST_INSTALL
3. Collect the files `Core/Autoload/UnifiedNamespaceClassMap.php` from each installed edition. Collect the file Core/Autoload/BackwardsCompatibilityClassMap.php from OXID eShop Community Edition
4. Generate the unified namespace classes and write them to the folder `vendor/oxid-esales/oxideshop-unified-namespace-generator/generated`. There should be one unified namespace class for every class in the OXID eShop edition.

Searching for errors

If you get either errors

- by calling on of the commands of *this section* or
- you get a message that a unified namespace class could not be found like

```
Class OxidEsales\Eshop\Core\ConfigFile not found in bootstrap.php on line 18
```

Then, you should read the following steps in order to find the reason for the error:

1. Have a look at the directory `vendor/oxid-esales/oxideshop-unified-namespace-generator/generated`
2. Are the unified namespace classes inside this directory, have the correct namespace and *extend the correct edition class*?
3. Be sure, the directory has write permissions
4. Execute the command **vendor/bin/oe-eshop-unified_namespace_generator** manually and look for errors
5. Be sure, the requirements as stated in *Mode Of Operation* are fulfilled

Module resources

Module development

Add dependencies and autoload via composer

Glossary:

- `<shop_directory>` - OXID eShop directory of the project.
- `<vendor>` - Vendor name of the module.
- `<module-vendor/module-name> ` - Name of the module which is registered in the composer file.
- `<branch_name>` - Branch name which will be used to develop the module.

Steps how to add

These steps describes how to add module dependency to OXID eShop project.

- Checkout module to the modules directory in the OXID eShop.

```
cd <shop_directory>/source/modules/<vendor>
git clone <git_path_to_module_repository> <module_id>
```

- Add a link from module to the Shop composer file.

```
cd <shop_directory>
composer config repositories.<module-vendor/module-name> path <shop_directory>/source/
↪modules/<vendor><module_id>
```

- Install module through a composer.

```
composer require <module-vendor/module-name>:*
```

Note: Composer will silently take other branch or release if a requirement could not be solved differently.

For example:

- Module has a release without requirements.
- Current code requires dependency in the module composer file.
- System does not meet the requirement.
- After composer install older module without requirements will be taken by composer.

Disable usage of Packagist to avoid this situation.

Why in this way

- Adding module to the modules directory allows to change files of the module and see changes on the fly.
- Installing though the composer will:
  - Add all the dependencies of the module to the project.

- Register module namespace so composer autoloader could be used to load objects.

Namespace

Composer autoloader is used to load classes. In order to load module classes the module needs to register it's namespace to the modules path:

```
"autoload": {
  "psr-4": {
    "<vendor>\\<module-name>\\": "../../../source/modules/<vendor>/<module-name>"
  }
},
```

Note: Shop v6 still supports modules for Shop v5.3. Classes without namespaces might be registered in the module metadata file. Read more in OXID Forge.

Override existing OXID eShop functionality

This page describes how to override default OXID eShop functionality.

Extending 'add to basket' functionality

In this section the existing "loggerdemo" module will be used which logs a product's id when it is added to the basket.

Override functionality

To override functionality there is a need to create a module class. Here, the "loggerdemo" module will be used as an example.

There is a need to create a child class - `OxidEsales\LoggerDemo\Model\Basket` - which should override OXID eShop class `OxidEsales\EshopCommunity\Application\Model\Basket` method `addToBasket`:

```
.
- source
    - modules
        - oe
           -loggerdemo
              - Model
                  - Basket.php
```

Note: Here `oe` - module developer vendor name, `loggerdemo` - module name.

The class `OxidEsales\LoggerDemo\Model\Basket` could have contents like this:

```
namespace OxidEsales\LoggerDemo\Model;
use OxidEsales\EventLoggerDemo\BasketItemLogger;

class Basket extends Basket_parent
```

```
{
    public function addToBasket(
        $productID,
        $amount,
        $sel = null,
        $persParam = null,
        $override = false,
        $bundle = false,
        $oldBasketItemId = null
    ) {
        $basketItemLogger = new BasketItemLogger($this->getConfig()->getLogsDir());
        $basketItemLogger->logItemToBasket($productID);
        return parent::addToBasket($productID, $amount, $sel, $persParam, $override,
↪$bundle, $oldBasketItemId);
    }
}
```

In this example method `addToBasket` is overridden and it adds logging functionality. To override the method one needs to:

- Extend a *Unified Namespace* class - `<className>_parent`, in this case it is `Basket_parent`.
- Call parent method, so the chain would not be broken.

Autoload module classes

The file *composer.json* in module root directory must be created (see "*How to create a module installable via composer?*") and module namespace must be defined (see "*Add dependencies and autoload via composer: Namespace*").

The *composer.json* file in module root directory could look like this:

```
{
  "name": "oxid-esales/logger-demo-module",
  "description": "This package contains demo module for OXID eShop.",
  "type": "oxideshop-module",
  "keywords": ["oxid", "modules", "eShop", "demo"],
  "homepage": "https://www.oxid-esales.com/en/home.html",
  "license": [
    "GPL-3.0",
    "proprietary"
  ],
  "require": {
    "oxid-esales/event_logger_demo": "dev-master"
  },
  "autoload": {
    "psr-4": {
      "OxidEsales\\LoggerDemo\\": "../../../source/modules/oe/loggerdemo"
    }
  },
  "minimum-stability": "dev",
  "prefer-stable": true,
  "extra": {
    "oxideshop": {
      "target-directory": "oe/loggerdemo"
    }
  }
}
```

The project *composer.json* file should have entries looking like this:

```
"repositories": {
    "oxid-esales/logger-demo-module": {
        "type": "path",
        "url": "source/modules/oe/loggerdemo"
    }
},
"require": {
    "oxid-esales/logger-demo-module": "dev-master"
}
```

To register a namespace and download dependencies there is a need to run composer update command in project root directory:

```
composer update
```

Composer will generate the PSR-4 autoload file with included module. So at this point OXID eShop will be able to autoload classes.

Add entry to module metadata file

OXID eShop needs to know which class should be extended, to do this there is a need to add a record in *metadata.php* file:

```
'extend' => [
  \OxidEsales\Eshop\Application\Model\Basket::class =>␣
↪\OxidEsales\LoggerDemo\Model\Basket::class,
],
```

Module Structure

Module structure in OXID eShop

All modules exist in the OXID eShop modules directory.

To separate modules it is:

- **Recommended** to group them by unique **vendor**.
- **Required** to give them unique id.
- **Required** to store module files in a directory with a name equal to **module_id**.

So the final structure of a module should be:

```
.
- source
    - modules
        - <vendor>
            - <module_id>
                - composer.json
                - Controller
                - metadata.php
                - Model
                - README.md
```

```
                - ...
                - tests
```

Module structure in module repository

In the repository it is recommended to keep module files without vendor or module directory. This allows to clone and use module directly in OXID eShop modules directory. Possible structure of the module in the repository:

```
.
- composer.json
- Controller
- metadata.php
- Model
- README.md
- ...
- tests
```

Module transformation

OXID Composer Plugin could be used in order to to create vendor and module_id directories

Language files

Language files are not specified inside the metadata.php but searched by naming conventions inside the module directory.

Example language file:

```
<?php

  $sLangName = 'English';

  $aLang = array(
      'charset'                      => 'UTF-8',

      'VENDORMYMODULEIDLANGUAGEKEY' => 'my translation of VENDORMYMODULEIDLANGUAGEKEY
↪',
  );
```

UTF-8 is the only possible charset for language files as the OXID eShop runs by default with UTF-8 itself and does not convert charsets. If you use any other charset for your language files, you have to use html codes for special characters.

Frontend

Translation files can be placed in the folders

- `Application/translations`
- `application/translations`
- `translations`

inside your module directory. If you have a folder `Application` or `application` inside your module, translation files are searched inside this directory. Otherwise, they are searched inside the folder `translations`. Inside these directory, you have to create a directory for the specific language, e.g. `de` or `en`. Inside the language specific, directory, the filename has to be _lang.php.

Example:

```
.
- source
    - modules
        - <vendor>
            - <module_id>
                - translations
                    - de
                        - myvendormymodule_de_lang.php
                    - en
                        - myvendormymodule_en_lang.php
```

Admin

Translation files can be placed in

- `Application/views/admin/`

Example:

```
.
- source
    - modules
        - <vendor>
            - <module_id>
                - Application
                    - views
                        - admin
                            - de
                                - myvendormymodule_admin_de_lang.php
                            - en
                                - myvendormymodule_admin_en_lang.php
```

For translations of module settings, have a look at the section *settings of the metadata file*.

Note: In order to use translation files in your module, you have to specify at least one class inside the section `extend` in your metadata.php.

Custom JavaScript / CSS / Images

Create out/src/js/, out/src/img/ and out/src/css/ directories so it fit Shop structure and would be easier to debug for other people. You can use something like this to include your scripts in to templates:

```
[{oxscript include=$oViewConf->getModuleUrl("{moduleID}", "out/src/js/{js_fle_name}.js
↪")}]
```

Module testing

It is recommended to write tests by using OXID Testing Library.

OXID Testing Library helps to test single module by:

- Adding helpers to write tests.
- Adding communication with OXID eShop layer.
- Ensuring that tests do not affect each other due to database usage.
- Stabilizing Selenium tests.
- **Allows to test compilation intercompatibility:** OXID eShop allows several modules to work at the same time and they might interact with each other. Testing Library allows to easily run tests for each module to check intercompatibility.

Module tests structure

Default Testing Library behavior is to run all tests which are defined in one of the test classes:

- AllTestsUnit
- AllTestsSelenium

These classes define default directories to store tests for a module:

- Unit
- Integration
- Acceptance

Possible structure of module tests:

```
<module_id>/tests/Acceptance/testData/fileNeededToBeCopiedToShop
<module_id>/tests/Acceptance/testSql/demodata.sql
<module_id>/tests/Acceptance/testSql/demodata_PE_CE.sql
<module_id>/tests/Acceptance/testSql/demodata_EE.sql
<module_id>/tests/Acceptance/testSql/demodata_EE_mall.sql
<module_id>/tests/Acceptance/moduleAcceptanceTest.php
<module_id>/tests/Integration/moduleIntegrationTest.php
<module_id>/tests/Unit/moduleUnitTest.php
<module_id>/tests/additional.inc.php
<module_id>/tests/phpunit.xml
```

Possible example in PayPal GitHub repository.

Testing library and it's documentation in GitHub.

Users predefined in demo data

If you are running tests or using `reset-shop` functionality of testing library, it's possible to use these credentials in OXID eShop:

```
Rights: admin
User name: admin
Password: admin
```

```
Rights: buyer
User name: user@oxid-esales.com
Password: user
```

Note: The status of this document is in progress. More information will be added later.

Steps for creating a module

- Initiate the repository
- *Create composer.json file*
- Create metadata file (Information is available in blog post and in *documentation metadata page*)
- *Override existing OXID eShop functionality*
- *Create module structure*
- *Add dependencies and autoload via composer*
- *Test module*

More information how to write a module for older OXID eShop versions could be found in this tutorial.

Demo Logger module could be used as a simple example from GitHub repository.

PayPal module could be used as an advanced example from GitHub repository.

Metadata

Since OXID eShop version 4.9.0 / 5.2.0 (Release notes) each module has to have metadata set. This has to be done with a file metadata.php in the module directory.

Note: There is already a blog post about Module Metadata but this blog post is partly outdated with release of OXID eShop 6.0.

Helpers

Here are some links to little helpers/tools for developers of modules for the OXID eShop:

- OXID module internals
- Metadata Generator

Version 1.0

The same like version 1.1 but without module events.

Version 1.1

Changes compared to version 1.0

- *Module events*

id

The extension id must be unique. It is recommended to use vendor prefix + module root directory name. Module ID is used for getting all needed information about extension. If this module has defined config variables in `oxconfig` and `oxconfigdisplay` tables (e.g. `module:efifactfinder`), the extension id used in these tables should match extension id defined in metadata file. Also same id (`efifactfinder`) must be used when defining extension templates blocks in `oxtplblocks` table.

Note: The extension id for modules written for OXID eShop versions >= 4.7.0 mustn't be > 25 characters. The extension id for modules written for OXID eShop versions >= 4.9.0 mustn't be > 93 characters. Please also see https://bugs.oxid-esales.com/view.php?id=5549.

---

title

Used to display extension title in the extensions list and detail information.

description

Used to display extension description in the extension detail information page. This field is *multilang capable*

lang

Default extension language. Displaying extension title or description there will be checked if these fields have a selected language. If not, the selected language defined in the lang field will be selected. E.g. if admin is opened in German and extension is available in English, the English title and description value will be shown as there is translation into German.

thumbnail

Extension thumbnail filename. Thumbnail should be in root folder and it is displayed in admin under extension details page.

version

The version number of this extension.

author

The author/developer of this extension.

url

Link to module writer web page.

email

Module vendor email.

extend

On this place shall be defined which shop classes are extended by this module. Here is an example:

```
'extend'       => array(
      'order'         => 'oe/oepaypal/controllers/oepaypalorder',
      'payment'       => 'oe/oepaypal/controllers/oepaypalpayment',
      'wrapping'      => 'oe/oepaypal/controllers/oepaypalwrapping',
      'oxviewconfig' => 'oe/oepaypal/controllers/oepaypaloxviewconfig',
      'oxaddress'     => 'oe/oepaypal/models/oepaypaloxaddress',
      'oxuser'        => 'oe/oepaypal/models/oepaypaloxuser',
      'oxorder'       => 'oe/oepaypal/models/oepaypaloxorder',
      'oxbasket'      => 'oe/oepaypal/models/oepaypaloxbasket',
      'oxbasketitem' => 'oe/oepaypal/models/oepaypaloxbasketitem',
      'oxarticle'     => 'oe/oepaypal/models/oepaypaloxarticle',
      'oxcountry'     => 'oe/oepaypal/models/oepaypaloxcountry',
      'oxstate'       => 'oe/oepaypal/models/oepaypaloxstate',
  ),
```

This information is used for activating/deactivating extension. Take care you declare the keys (e.g. oxorder) always in lower case! Take care you declare the file names case sensitive! It is suggested to use lower case for file names, to avoid difficulties.

files

All module php files that do not extend any shop class. On request shop autoloader checks this array and if class name is registered in this array, loads class. So now no need to copy module classes to shop `core` or `view` folder and all module files can be in module folder.

```
'files' => array(
      'oePayPalException'                => 'oe/oepaypal/core/exception/
↪oepaypalexception.php',
      'oePayPalCheckoutService'          => 'oe/oepaypal/core/
↪oepaypalcheckoutservice.php',
      'oePayPalLogger'                   => 'oe/oepaypal/core/oepaypallogger.php',
      'oePayPalPortlet'                  => 'oe/oepaypal/core/oepaypalportlet.php',
      'oePayPalDispatcher'               => 'oe/oepaypal/controllers/
↪oepaypaldispatcher.php',
      'oePayPalExpressCheckoutDispatcher' => 'oe/oepaypal/controllers/
↪oepaypalexpresscheckoutdispatcher.php',
      'oePayPalStandardDispatcher'       => 'oe/oepaypal/controllers/
↪oepaypalstandarddispatcher.php',
      'oePaypal_EblLogger'               => 'oe/oepaypal/core/oeebl/oepaypal_
↪eblogger.php',
      'oePaypal_EblPortlet'              => 'oe/oepaypal/core/oeebl/oepaypal_
↪eblportlet.php',
```

```
      'oePaypal_EblSoapClient'            => 'oe/oepaypal/core/oeebl/oepaypal_
↪eblsoapclient.php',
      'oepaypalevents'                    => 'oe/oepaypal/core/oepaypalevents.php',
  ),
```

blocks

In this array are registered all module templates blocks. On module activation they are automaticly inserted into database. On activating/deactivating module, all module blocks also are activated/deactivated

```
'blocks' => array(
      array(
          'template' => 'widget/sidebar/partners.tpl',
          'block'=>'partner_logos',
          'file'=>'/views/blocks/oepaypalpartnerbox.tpl'
          'position' => '2'
      ),
      array(
          'template' => 'page/checkout/basket.tpl',
          'block'=>'basket_btn_next_top',
          'file'=>'/views/blocks/oepaypalexpresscheckout.tpl'
          'position' => '1'
      ),
      array(
          'template' => 'page/checkout/basket.tpl',
          'block'=>'basket_btn_next_bottom',
          'file'=>'/views/blocks/oepaypalexpresscheckout.tpl'
      ),
  ),
  )
```

Differences in block file definition per shop/metadata version.

In OXID eShop >= 4.6 with metadata version 1.0 template block `file` value was relative to `out/blocks` directory inside module root.

In OXID eShop 4.7 / 5.0 with metadata version 1.1 template block `file` value has to be specified directly from module root.

To maintain compatibility with older shop versions, template block files will work using both notations.

Template block `file` value holding path to your customized block should be defined using full path from module directory, earlier it was a sub path from modules `out/blocks` directory.

You can define a position of a block if a template block is extended multiple (by different modules). So you can sort the block extensions. This is done via the optional template block `position` value.

settings

There are registered all module configuration options. On activation they are inserted in config table and then in backend you can configure module according these options. Lets have a look at the code to become a clearer view.

```
'settings' => array(
      array('group' => 'main', 'name' => 'dMaxPayPalDeliveryAmount', 'type' => 'str',␣
↪      'value' => '30'),
      array('group' => 'main', 'name' => 'blPayPalLoggerEnabled',    'type' => 'bool',
↪      'value' => 'false'),
```

```
      array('group' => 'main', 'name' => 'aAlwaysOpenCats',          'type' => 'arr',␣
↪      'value' => array('Preis','Hersteller')),
      array('group' => 'main', 'name' => 'aFactfinderChannels',      'type' => 'aarr',
↪      'value' => array('1' => 'de', '2' => 'en')),
      array('group' => 'main', 'name' => 'sConfigTest',              'type' => 'select
↪',    'value' => '0', 'constraints' => '0|1|2|3', 'position' => 3 ),
      array('group' => 'main', 'name' => 'sPassword',                'type' =>
↪'password', 'value' => 'changeMe')
  )

/* Entries in lang.php for constraints example:
'SHOP_MODULE_sConfigTest'        => 'Field Label',
'SHOP_MODULE_sConfigTest_0'      => '',
'SHOP_MODULE_sConfigTest_1'      => 'Value x',
'SHOP_MODULE_sConfigTest_2'      => 'Value y',
'SHOP_MODULE_sConfigTest_3'      => 'Value z'
*/
```

Each setting belongs to a group. In this case its called `main`. Then follows the name of the setting which is the variable name in oxconfig/oxconfigdisplay table. It is best practice to prefix it with your moduleid to avoid name collisions with other modules. Next part is the type of the parameter and last part is the default value.

In order to get correct translations of your settings names in admin one should create *views/admin/module_options.php'* where is the language with 2 letters for example `en` for english. There should be placed the language constants according to the following scheme:

```
// Entries in module_options.php for above code examples first entry:
'SHOP_MODULE_GROUP_main'                    => 'Paypal settings',
'SHOP_MODULE_dMaxPayPalDeliveryAmount'      => 'Maximal delivery amount',
'HELP_SHOP_MODULE_dMaxPayPalDeliveryAmount' => 'A help text for this setting',
```

So the shop looks in the file for a language constant like `SHOP_MODULE_GROUP_` and for the single setting for a language constant like `SHOP_MODULE_`. In php classes you can query your module settings by using the `function getParameter()` of `oxConfig` class:

```
$myconfig = $this->getConfig();
$myconfig->getConfigParam("dMaxPayPalDeliveryAmount");
```

or since OXID 4.7 you can also use

```
$myconfig = oxRegistry::get("oxConfig");
$myconfig->getConfigParam("dMaxPayPalDeliveryAmount");
```

templates

Module templates array. All module templates should be registered here, so on requiring template shop will search template path in this array.

'templates' => array('order_dhl.tpl' => 'oe/efi_dhl/out/admin/tpl/order_dhl.tpl')

events

Module events were introduced in metadata version 1.1. Currently there are only 2 of them (onActivate and onDeactivate), more events will be added in future releases. Event handler class shoul'd be registered in medatata files array.

```
'events'       => array(
      'onActivate'   => 'oepaypalevents::onActivate',
      'onDeactivate' => 'oepaypalevents::onDeactivate'
  ),
```

custom JavaScript / CSS / Images

Create out/src/js/, out/src/img/ and out/src/css/ directories so it fit Shop structure and would be easier to debug for other people. You can use something like this to include your scripts in to templates:

```
[{oxscript include=$oViewConf->getModuleUrl("{moduleID}", "out/src/js/{js_fle_name}.js
↪")}]
```

Metadata file version

```
$sMetadataVersion = '1.1';
```

Here is an example of PayPal module metadata file:

```
/**
 * Metadata version
 */
$sMetadataVersion = '1.1';

/**
 * Module information
 */
$aModule = array(
  'id'           => 'oepaypal',
  'title'        => 'PayPal',
  'description'  => array(
      'de' => 'Modul fuer die Zahlung mit PayPal. Erfordert einen OXID eFire Account␣
↪und die abgeschlossene Aktivierung des Portlets "PayPal".',
      'en' => 'Module for PayPal payment. An OXID eFire account is required as well␣
↪as the finalized activation of the portlet "PayPal".',
  ),
  'thumbnail'    => 'logo.jpg',
  'version'      => '2.0.3',
  'author'       => 'OXID eSales AG',
  'url'          => 'http://www.oxid-esales.com',
  'email'        => 'info@oxid-esales.com',
  'extend'       => array(
      'order'         => 'oe/oepaypal/controllers/oepaypalorder',
      'payment'       => 'oe/oepaypal/controllers/oepaypalpayment',
      'wrapping'      => 'oe/oepaypal/controllers/oepaypalwrapping',
      'oxviewconfig' => 'oe/oepaypal/controllers/oepaypaloxviewconfig',
      'oxaddress'     => 'oe/oepaypal/models/oepaypaloxaddress',
      'oxuser'        => 'oe/oepaypal/models/oepaypaloxuser',
      'oxorder'       => 'oe/oepaypal/models/oepaypaloxorder',
      'oxbasket'      => 'oe/oepaypal/models/oepaypaloxbasket',
      'oxbasketitem' => 'oe/oepaypal/models/oepaypaloxbasketitem',
      'oxarticle'     => 'oe/oepaypal/models/oepaypaloxarticle',
      'oxcountry'     => 'oe/oepaypal/models/oepaypaloxcountry',
      'oxstate'       => 'oe/oepaypal/models/oepaypaloxstate',
```

```
  ),
  'files' => array(
      'oePayPalException'                => 'oe/oepaypal/core/exception/
↪oepaypalexception.php',
      'oePayPalCheckoutService'          => 'oe/oepaypal/core/
↪oepaypalcheckoutservice.php',
      'oePayPalLogger'                   => 'oe/oepaypal/core/oepaypallogger.php',
      'oePayPalPortlet'                  => 'oe/oepaypal/core/oepaypalportlet.php',
      'oePayPalDispatcher'               => 'oe/oepaypal/controllers/
↪oepaypaldispatcher.php',
      'oePayPalExpressCheckoutDispatcher' => 'oe/oepaypal/controllers/
↪oepaypalexpresscheckoutdispatcher.php',
      'oePayPalStandardDispatcher'       => 'oe/oepaypal/controllers/
↪oepaypalstandarddispatcher.php',
      'oePaypal_EblLogger'               => 'oe/oepaypal/core/oeebl/oepaypal_
↪eblogger.php',
      'oePaypal_EblPortlet'              => 'oe/oepaypal/core/oeebl/oepaypal_
↪eblportlet.php',
      'oePaypal_EblSoapClient'           => 'oe/oepaypal/core/oeebl/oepaypal_
↪eblsoapclient.php',
      'oepaypalevents'                   => 'oe/oepaypal/core/oepaypalevents.php',
  ),
  'events'       => array(
      'onActivate'   => 'oepaypalevents::onActivate',
      'onDeactivate' => 'oepaypalevents::onDeactivate'
  ),
  'blocks' => array(
      array('template' => 'widget/sidebar/partners.tpl', 'block'=>'partner_logos',␣
↪                'file'=>'/views/blocks/oepaypalpartnerbox.tpl'),
      array('template' => 'page/checkout/basket.tpl',    'block'=>'basket_btn_next_top
↪',               'file'=>'/views/blocks/oepaypalexpresscheckout.tpl'),
      array('template' => 'page/checkout/basket.tpl',    'block'=>'basket_btn_next_
↪bottom',            'file'=>'/views/blocks/oepaypalexpresscheckout.tpl'),
      array('template' => 'page/checkout/payment.tpl',   'block'=>'select_payment',␣
↪                'file'=>'/views/blocks/oepaypalpaymentselector.tpl'),
  ),
 'settings' => array(
      array('group' => 'main', 'name' => 'dMaxPayPalDeliveryAmount', 'type' => 'str',␣
↪ 'value' => '30'),
      array('group' => 'main', 'name' => 'blPayPalLoggerEnabled',    'type' => 'bool',
↪ 'value' => 'false'),
  )
);
```

Multilanguage fields

Note: This section is about multilanguage fields of strings introduced in the metadata.php file itself. If you want to use translations in your module for frontend or backend, you should place them in your module according the *module structure conventions*

Extension description is a multilanguage field. This should be an array with a defined key as language abbervation and the value of it's translation.

```
'description'  => array(
  'de'=>'Intelligente Produktsuche und Navigation.',
  'en'=>'Intelligent product search and navigation.',
)
```

The field value also can be a simple string. If this field value is not an array but simple text, this text string will be displayed in all languages.

Vendor directory support

All modules can be placed not directly in shop modules directory, but also in vendor directory. In this case the `vendormetadata.php` file must be placed in the vendor directory root. If the modules handler finds this file on scanning the shop modules directory, it knows that this is vendor directory and all subdirectories in this directory should be scanned also. Currently the `vendormetadata.php` file can be empty, in future here will be added some additional information about the module vendor. Vendor directory structure example:

```
modules
  oxid
    module1
      module1 files
    module2
      module2 files
    module3
      module3 files
```

In case of using a vendor directory you still need to describe file paths relatively to the modules directory:

```
'extend' => array(
      'some_class' => 'oxid/module1/my_class'
),
'templates' => array(
      'my_template.tpl' => 'oxid/module1/my_template.tpl'
)
```

Version 2.0

Changes compared to version 1.1

- New Section Controllers: To be able to use namespaces for module controllers, we introduce module's metadata.php version 2.0 with a new section `controllers`. The support for `files` was dropped in Module's metadata version 2.0. Classes in a namespace will be found by the autoloader. If you use your own namespace, *register it in the module's composer.json file*.

Important: You can use metadata version 2.0 with controllers only for modules using namespaces. When using modules without a namespace you will have to use metadata version 1.0 with the 'files' section to register your module controllers.

- Templates and blocks for different Shop themes. It also allows to define templates and blocks for all themes (define in the same way as in old metadata).

id

The extension id must be unique. It is recommended to use vendor prefix + module root directory name. Module ID is used for getting all needed information about extension. If this module has defined config variables in `oxconfig` and `oxconfigdisplay` tables (e.g. `module:efifactfinder`), the extension id used in these tables should match extension id defined in metadata file. Also same id (`efifactfinder`) must be used when defining extension templates blocks in `oxtplblocks` table.

Note: the extension id for modules written for OXID eShop versions >= 4.9.0 mustn't be > 93 characters. Please also see https://bugs.oxid-esales.com/view.php?id=5549.

---

title

Used to display extension title in the extensions list and detail information.

description

Used to display extension description in the extension detail information page. This field is *multilang capable*

lang

Default extension language. Displaying extension title or description there will be checked if these fields have a selected language. If not, the selected language defined in the lang field will be selected. E.g. if admin is opened in German and extension is available in English, the English title and description value will be shown as there is translation into German.

thumbnail

Extension thumbnail filename. Thumbnail should be in root folder and it is displayed in admin under extension details page.

version

The version number of this extension.

author

The author/developer of this extension.

url

Link to module writer web page.

email

Module vendor email.

extend

On this place shall be defined which shop classes are extended by this module. You can use metadata version 2.0 with *controllers* only for modules using namespaces.

```
'extend'       => array(
      \OxidEsales\Eshop\Application\Model\Payment::class =>␣
↪MyVendor\MyModuleNamespace\Application\Model\MyModulePayment::class,
      \OxidEsales\Eshop\Application\Model\Article::class =>␣
↪MyVendor\MyModuleNamespace\Application\Model\MyModuleArticle::class
  ),
```

You should extend only OXID eShop classes within the *Unified Namespace* (`\OxidEsales\Eshop`). If you try to extend e.g a class of the namespace `\OxidEsales\EshopCommunity`, you are not able to activate the module and get a warning message in the OXID eShop admin.

controllers

At this place, you can define, which controllers should be able to be called directly, e.g. from templates. You can define a routing of `controller keys` to module classes.

The key of this array

- is a identifier (`controller key`) which should be unique over all OXID eShop modules. Use vendor id and module id for prefixing.
- Take care you declare the keys always in lower case!

The value is the assigned class which should also be unique.

```
'controllers'  => [
    'myvendor_mytestmodule_mymodulecontroller' =>␣
↪MyVendor\mytestmodule\MyModuleController::class,
    'myvendor_mytestmodule_myothermodulecontroller' =>␣
↪MyVendor\mytestmodule\MyOtherModuleController::class,
],
```

Now you can route requests to the module controller e.g. in a template:

```
<form action="[{$oViewConf->getSelfActionLink()}]" name="MyModuleControllerAction"␣
↪method="post" role="form">
    <div>
        [{$oViewConf->getHiddenSid()}]
        <input type="hidden" name="cl" value="myvendor_mytestmodule_mymodulecontroller
↪">
        <input type="hidden" name="fnc" value="displayMessage">
        <input type="text" size="10" maxlength="200" name="mymodule_message" value="[{
↪$the_module_message}]">
        <button type="submit" id="MyModuleControllerActionButton" class="submitButton
↪">[{oxmultilang ident="SUBMIT"}]</button>
    </div>
</form>
```

If the controller key is not found within the shop or modules, it is assumed that the controller key is a class with this name. If there is no class with this name present, the OXID eShop will redirect to the shop front page.

blocks

In this array are registered all module templates blocks. On module activation they are automaticly inserted into database. On activating/deactivating module, all module blocks also are activated/deactivated.

```
'blocks' => array(
      array(
          'template' => 'widget/sidebar/partners.tpl',
          'block'=>'partner_logos',
          'file'=>'/views/blocks/oepaypalpartnerbox.tpl'
          'position' => '2'
      ),
      array(
          'template' => 'page/checkout/basket.tpl',
          'block'=>'basket_btn_next_top',
          'file'=>'/views/blocks/oepaypalexpresscheckout.tpl'
          'position' => '1'
      ),
      array(
          'template' => 'page/checkout/basket.tpl',
          'block'=>'basket_btn_next_bottom',
          'file'=>'/views/blocks/oepaypalexpresscheckout.tpl'
      ),
  ),
  )
```

The template block `file` value has to be specified directly from module root. You can define a position of a block if a template block is extended multiple (by different modules). So you can sort the block extensions. This is done via the optional template block `position` value.

To describe block or overwrite default block template for specific theme, use theme attribute in block description.

```
'blocks' => array(
    array(
        'theme' => 'shop_theme_id'
        'template' => 'name_off_shop_template_which_contains_block',
        'block'=>'name_off_shop_block',
        'file'=>'path_to_module_block_file'
    ),
```

Note:

- To override default block use same template and block values.
- Specific block will override all files for specific block.
- It is not allowed to use *admin* as a theme id.

Example

```
'blocks' => array(
    array(
        'template' => 'deliveryset_main.tpl',
```

```
        'block'=>'admin_deliveryset_main_form',
        'file'=>'/views/blocks/deliveryset_main.tpl',
    ),
    array(
        'template' => 'widget/sidebar/partners.tpl',
        'block'=>'partner_logos',
        'file'=>'/views/blocks/widget/sidebar/oepaypalpartnerbox1.tpl',
    ),
    array(
        'template' => 'widget/sidebar/partners.tpl',
        'block'=>'partner_logos',
        'file'=>'/views/blocks/widget/sidebar/oepaypalpartnerbox2.tpl',
    ),
    array(
        'theme' => 'flow_theme',
        'template' => 'widget/sidebar/partners.tpl',
        'block'=>'partner_logos',
        'file'=>'/views/blocks/widget/sidebar/oepaypalpartnerboxForFlow.tpl',
    ),
)
```

In this particular example:

- If *flow_theme* theme is active, the contents of *oepaypalpartnerboxForFlow.tpl* file would be loaded in *partners.tpl* partner_logos block.
- For other then *flow_theme* theme, the *oepaypalpartnerbox1.tpl* and *oepaypalpartnerbox2.tpl* files contents would be shown in *partners.tpl partner_logos block*.

Custom blocks

It is possible to reuse template blocks for parent theme when child theme extends parent theme.

```
'blocks' => array(
    array(
        'template' => 'widget/minibasket/minibasket.tpl',
        'block'=>'widget_minibasket_total',
        'file'=> '/views/blocks/widget/minibasket/oepaypalexpresscheckoutminibasket.
↪tpl',
    ),
    array(
        'template' => 'widget/sidebar/partners.tpl',
        'block'=> 'partner_logos',
        'file'=>'/views/blocks/widget/sidebar/oepaypalpartnerbox.tpl',
    ),
    array(
        'theme' => 'flow_theme',
        'template' => 'widget/minibasket/minibasket.tpl',
        'block'=> 'widget_minibasket_total',
        'file'=> '/views/blocks/widget/minibasket/
↪oepaypalexpresscheckoutminibasketFlow.tpl',
    ),
    array(
        'theme' => 'flow_theme',
        'template' => 'widget/sidebar/partners.tpl',
        'block'=> 'partner_logos',
        'file'=> '/views/blocks/widget/sidebar/oepaypalpartnerboxForFlow.tpl',
```

```
    ),
    array(
        'theme' => 'flow_theme_child',
        'template' => 'widget/sidebar/partners.tpl',
        'block'=> 'partner_logos',
        'file'=> '/views/blocks/widget/sidebar/oepaypalpartnerboxForMyCustomFlow.tpl',
    ),
)
```

In this particular example *flow_theme_child* extends *flow_theme*. If *flow_theme_child* theme would be active:

- *oepaypalpartnerboxForMyCustomFlow.tpl* template block would be used instead of *partner_logos*.
- *oepaypalexpresscheckoutminibasketFlow.tpl* template would be used instead of *widget_minibasket_total*.

settings

There are registered all module configuration options. On activation they are inserted in config table and then in backend you can configure module according these options. Lets have a look at the code to become a clearer view.

```
'settings' => array(
      array('group' => 'main', 'name' => 'dMaxPayPalDeliveryAmount', 'type' => 'str',␣
↪      'value' => '30'),
      array('group' => 'main', 'name' => 'blPayPalLoggerEnabled',    'type' => 'bool',
↪      'value' => 'false'),
      array('group' => 'main', 'name' => 'aAlwaysOpenCats',          'type' => 'arr',␣
↪      'value' => array('Preis','Hersteller')),
      array('group' => 'main', 'name' => 'aFactfinderChannels',      'type' => 'aarr',
↪      'value' => array('1' => 'de', '2' => 'en')),
      array('group' => 'main', 'name' => 'sConfigTest',              'type' => 'select
↪',    'value' => '0', 'constraints' => '0|1|2|3', 'position' => 3 ),
      array('group' => 'main', 'name' => 'sPassword',                'type' =>
↪'password', 'value' => 'changeMe')
  )
```

Each setting belongs to a group. In this case its called `main`. Then follows the name of the setting which is the variable name in oxconfig/oxconfigdisplay table. It is best practice to prefix it with your moduleid to avoid name collisions with other modules. Next part is the type of the parameter and last part is the default value.

In order to get correct translations of your settings names in admin one should create `views/admin/module_options.php` where is the language with 2 letters for example `en` for english. There should be placed the language constants according to the following scheme:

```
// Entries in module_options.php for above code examples first entry:
'SHOP_MODULE_GROUP_main'                    => 'Paypal settings',
'SHOP_MODULE_dMaxPayPalDeliveryAmount'      => 'Maximal delivery amount',
'HELP_SHOP_MODULE_dMaxPayPalDeliveryAmount' => 'A help text for this setting',
```

So the shop looks in the file for a language constant like `SHOP_MODULE_GROUP_` and for the single setting for a language constant like `SHOP_MODULE_`. In php classes you can query your module settings by using the `function getConfigParam()` of `Config` class:

```
$myconfig = Registry::getConfig();
$myconfig->getConfigParam("dMaxPayPalDeliveryAmount");
```

templates

All module templates should be registered here, so on requiring template shop will search template path in this array. Default template (for all themes) are described in same way as in metadata v1.*

```
'templates' => array(
    'module_template_name'    => 'path_to_module_template',
)
```

To have template for specific theme, define it in an array with the key equal to theme id.

```
'templates' => array(
    'theme_id' => array(
        'module_template_name'    => 'path_to_module_template',
    )
)
```

Note:

- Its possible to use any theme id, even default one, if you want to specify some template for the theme.
- It is not allowed to use *admin* as a theme id.

Example

```
'templates' => array(
    'order_paypal.tpl' => 'oe/oepaypal/views/admin/tpl/order_paypal.tpl',
    'ipnhandler.tpl'   => 'oe/oepaypal/views/tpl/ipnhandler.tpl',
    'more.tpl'         => 'oe/oepaypal/views/tpl/moreDefault.tpl',

    'flow_theme' => array(
        'more.tpl' => 'oe/oepaypal/views/tpl/moreFlow.tpl',
    )
)
```

Templates for child theme

It is possible to reuse templates for parent theme when child theme extends parent theme. This mechanism is especially useful in project scope when needs to customize an already existing theme.

```
'templates' => array(
    'order_paypal.tpl' => 'oe/oepaypal/views/admin/tpl/order_paypal.tpl',
    'ipnhandler.tpl'   => 'oe/oepaypal/views/tpl/ipnhandler.tpl',
    'more.tpl'         => 'oe/oepaypal/views/tpl/moreDefault.tpl',

    'flow_theme' => array(
        'ipnhandler.tpl' => 'oe/oepaypal/views/tpl/ipnhandlerFlow.tpl',
        'more.tpl'       => 'oe/oepaypal/views/tpl/moreFlow.tpl',
    ),

    'flow_theme_child' => array(
        'more.tpl'   => 'oe/oepaypal/views/tpl/moreMyCustomFlow.tpl',
    )
)
```

In this particular example *flow_theme_child* extends *flow_theme*. If *flow_theme_child* theme would be active:

- *moreMyCustomFlow.tpl* template would be used instead of *more.tpl*.
- *ipnhandlerFlow.tpl* template would be used instead of *ipnhandler.tpl*.

events

Module events were introduced in metadata version 1.1. There are 2 events: onActivate and onDeactivate.

```
'events'       => array(
      'onActivate'   => '\OxidEsales\PayPalModule\Core\Events::onActivate',
      'onDeactivate' => '\OxidEsales\PayPalModule\Core\Events::onDeactivate'
  ),
```

Metadata file version

```
$sMetadataVersion = '2.0';
```

Multilanguage fields

Note: This section is about multilanguage fields of strings introduced in the metadata.php file itself. If you want to use translations in your module for frontend or backend, you should place them in your module according the *module structure conventions*.

Extension description is a multilanguage field. This should be an array with a defined key as language abbervation and the value of it's translation.

```
'description'  => array(
  'de'=>'Intelligente Produktsuche und Navigation.',
  'en'=>'Intelligent product search and navigation.',
)
```

The field value also can be a simple string. If this field value is not an array but simple text, this text string will be displayed in all languages.

Vendor directory support

All modules can be placed not directly in shop modules directory, but also in vendor directory. Vendor directory structure example:

```
modules
  oxid
    module1
      module1 files
    module2
      module2 files
    module3
      module3 files
```

Example of metadata.php

Here is an example of a module metadata file:

```
Example for module using namespaces

<?php
/**
 * Metadata version
 */
$sMetadataVersion = '2.0';
/**
 * Module information
 */
$aModule = array(
    'id'           => 'myvendor_mytestmodule',
    'title'        => 'Test metadata controllers feature',
    'description'  => '',
    'thumbnail'    => 'picture.png',
    'version'      => '2.0',
    'author'       => 'OXID eSales AG',
    'controllers'  => [
        'myvendor_mytestmodule_MyModuleController' =>␣
↪MyVendor\mytestmodule\MyModuleController::class,
        'myvendor_mytestmodule_MyOtherModuleController' =>␣
↪MyVendor\mytestmodule\MyOtherModuleController::class,
    ],
    'templates' => [
        'mytestmodule.tpl' => 'mytestmodule/mytestmodule.tpl',
        'mytestmodule_other.tpl' => 'mytestmodule/test_module_controller_routing_
↪other.tpl'
    ]
);
```

Compatibility between different metadata versions

This table shows, which versions of versions of OXID eShop are compatible with which version of metadata.php. The metadata version is not checked before OXID eShop version 6. Only features of the metadata versions are checked: e.g. the feature `events` is checked in OXID eShop > 4.9.

OXID eShop version	Metadata version
< 4.6	no metadata.php, 1.0, 1.1 with reduced feature set of 1.0
>= 4.6 and < 4.9/5.2	no metadata.php, 1.0, 1.1 with reduced feature set of 1.0
>= 4.9/5.2 and < 6.0	1.0, 1.1
>= 6	1.0, 1.1, 2.0

How to extend frontend user form?

There is a possibility to add additional form input fields in frontend without adding additional logic how to save the field data. This page will describe how to achieve this by using extend user module example.

Preparation

For having additional input field in user form first of all there will be a need to create new column in user table. This can be achieved by using module *events* which would create a column. In this page an example of database table column called `EXTENDUSER_ADDITIONALCONTACTINFO` will be used.

Template

The block which will have to be extend is located in template file *form/fieldset/user_billing.tpl*. To extend it there will be a need to create a template file and describe it in *metadata.php* file:

```
'blocks' => array(
    array('template' => 'form/fieldset/user_billing.tpl', 'block'=>'form_user_billing_
↪country', 'file'=>'/views/user.tpl'),
),
```

/views/user.tpl contents could look like this:

```
[{$smarty.block.parent}]

<div class="form-group">
    <label class="control-label col-lg-3">Additional contact info</label>
    <div class="col-lg-9">
        <input class="form-control" type="text" maxlength="128"
               name="invadr[oxuser__extenduser_additionalcontactinfo]"
               value="[{$oxcmp_user->oxuser__extenduser_additionalcontactinfo->value}]
↪"
               required=""
        >
    </div>
</div>
```

Most important thing here is input field with name attribute `name="invadr[oxuser__extenduser_additionalcontactini` which says for OXID eShop to try write into table `oxuser` column `EXTENDUSER_ADDITIONALCONTACTINFO` provided value.

Modify white listed fields

For security reasons there is an array of "white listed" fields. Only those table columns which has equivalent field in "white list" array can be updated by submitting form and passing parameters via POST request.

There are two classes which contains white listed fields:

- For table `oxusers` - `OxidEsales\EshopCommunity\Application\Model\User\UserUpdatableFields`.
- For table `oxaddress` - `OxidEsales\EshopCommunity\Application\Model\User\UserShippingAddressUp`

So to add additional field to the white list it's needed to extend one of those classes. In `oxuser` table case - `OxidEsales\EshopCommunity\Application\Model\User\UserUpdatableFields`. Entry in module metadata file would look like this:

```
'extend' => [
    \OxidEsales\Eshop\Application\Model\User\UserUpdatableFields::class =>␣
↪\OxidEsales\ExtendUser\UserUpdatableFields::class
],
```

And the contents of file could look like this:

```php
namespace OxidEsales\ExtendUser;
/**
 * @see \OxidEsales\Eshop\Application\Model\User\UserUpdatableFields
 */
class UserUpdatableFields extends UserUpdatableFields_parent
{
    public function getUpdatableFields()
    {
        $updatableFields = parent::getUpdatableFields();
        return array_merge($updatableFields, ['EXTENDUSER_ADDITIONALCONTACTINFO']);
    }
}
```

In this way into updatable fields array would be added new field - `EXTENDUSER_ADDITIONALCONTACTINFO`.

So after module activation new form functioning field will appear in the user form.

How to make an OXID eShop module installable via composer?

OXID eShop modules are installed via Composer by using the OXID eShop Composer Plugin.

In order to install a module correctly, this plugin requires four fields to be described in module `composer.json` file:

- *name*
- *type*
- *extra*
- *require*
- *autoload*

PayPal module example:

```json
{
    "name": "oxid-esales/paypal-module",
    "description": "This is the PayPal module for the OXID eShop.",
    "type": "oxideshop-module",
    "keywords": ["oxid", "modules", "eShop"],
    "homepage": "https://www.oxid-esales.com/en/home.html",
    "license": [
        "GPL-3.0",
        "proprietary"
    ],
    "extra": {
        "oxideshop": {
            "blacklist-filter": [
                "documentation/**/*.*"
            ],
            "target-directory": "oe/oepaypal"
        }
    },
    "require": {
        "php": ">=5.6",
        "lib-curl": ">=7.26.0",
        "lib-openssl": ">=1.0.1",
        "ext-curl": "*",
```

```
        "ext-openssl": "*"
    },
    "autoload": {
        "psr-4": {
            "OxidEsales\\PayPalModule\\": "../../../source/modules/oe/oepaypal"
        }
    }
}
```

name

This is the name the OXID eShop module will be publicly known and installable. E.g. in our example you could type

```
composer require oxid-esales/paypal-module
```

type

Module must have `oxideshop-module` value defined as a type. This defines how the repository should be treated by the installer.

extra: {oxideshop}

target-directory

`target-directory` value will be used to create a folder inside the Shop `modules` directory. This folder will be used to place all files of the module.

Important: It is strongly recommended to set the target directory value to `<vendor of the module>` + `<module ID>`, e.g. `oe/oepaypal`.

---

source-directory

If `source-directory` is given, the value defines which directory will be used to define where the files and directories will be picked from. When the parameter is not given, the root directory of the module is used instead.

Note: Usually this parameter should not be used if all files are placed in the module's root directory.

---

blacklist-filter

If `blacklist-filter` is given, it will be used to filter out unwanted files and directories while the copy from `source-directory` to `target-directory` takes place. The value of `blacklist-filter` must be a list of strings where each item represents a glob filter entry and is described as a relative path (relative to `source-directory`).

Below is a list of **valid** entries:

- `README.md` - will filter one specific file `README.md`;

- `*.pdf` - will filter all PDF documents from the source root directory;
- `**/*.pdf` - will filter all PDF documents from the source root directory and all of it's child directories;
- `example/path/**/*` - will filter all files and directories from the directory `example/path`, including the given directory itself.

Below is a list of **non-valid** entries:

- `/an/absolute/path/to/file` - absolute paths are not allowed, only relative paths are accepted;
- `some/path/` - ambigious description of directory to filter, it's not clear if only the files are needed to be filtered or directories have to be included as well.

For the most up-to-date definition of what can be accepted as an argument, please follow the tests which covers the behaviour.

require

Here you must define all dependencies your module has. You must define:

- a minimum PHP version. In the example PHP >=5.6 is required
- the required system libraries and their versions, if applicable. In the example lib-curl >=7.26.0 and lib-openssl >=1.0.1 are required
- the required PHP extension and their versions, if applicable. In the example the PHP extensions curl and openssl must be activated
- the required composer components, if applicable. In the example the are no requirements defined

autoload

It is necessary to define a PSR-4 compatible auto loading mechanism. For an easier development, we recommend to use "../../../source/modules/vendorname/moduleid". You will find more detailed development related information *here*

Keep in mind, that the *target-directory* in the section extra: {oxideshop} has to fit the autoload path you define here. In our PayPal example the PSR-4 autoload path points to a path inside the OXID eShop source/modules directory. This path must match the path of the *target-directory* as defined in the extra: {oxideshop} section, as the files will be copied there.

Multiple themes

The contents of this page moved to *Module Metadata Version 2.0*

Interacting with database

Active records and magic getters

Oxid uses active record to work with database. The OXID eShop architecture is based on MVC patterns. To implement models, active record pattern is used. So in general, each model class is linked with a database table. For example, the `Article` model is linked with the `oxarticles` table, Order with the `oxorders` table etc. All models are stored in the directory Application/Models. Let's take one of them, for example the `Article` model, and try to fetch the product (with the ID `demoId`) data from database:

```
$product = oxNew(\OxidEsales\Eshop\Application\Model\Article::class); // creating␣
↪model's object
$product->load( 'demoId' ); // loading data
//getting some informations
echo $product->oxarticles__oxtitle->value;
echo $product->oxarticles__oxshortdesc->value;
```

Magic getters are used to get models attributes; they are constructed in this approach:

```
$model->tablename__columnname->value;
'tablename' is the name of the database table where the model data is stored
'columnname' is the name of the column of this table containing the data you want to␣
↪fetch
```

To set data to a model and store it, database magic setters (with the same approach as magic getters) are used:

```
$product = oxNew(\OxidEsales\Eshop\Application\Model\Article::class);
$product->oxarticles__oxtitle = new \OxidEsales\Eshop\Core\Field ( 'productTitle' );
$product->oxarticles__oxshortdesc = new \OxidEsales\Eshop\Core\Field(
↪'shortdescription' );
$product->save();
```

In this example the new record will be inserted into the table. To update an information, we have to load the model, set the new data and call the save()-method:

```
$product = oxNew(\OxidEsales\Eshop\Application\Model\Article::class);
$product->load( 'demoId' );
$product->oxarticles__oxtitle = new \OxidEsales\Eshop\Core\Field ( 'productTitle' );
$product->oxarticles__oxshortdesc = new \OxidEsales\Eshop\Core\Field(
↪'shortdescription' );
$product->save();
```

There are other ways to do the same - without loading the data - just simply setting the ID with the setId()-method:

```
$product = oxNew(\OxidEsales\Eshop\Application\Model\Article::class);
$product->setId( 'demoId' );
$product->oxarticles__oxtitle = new \OxidEsales\Eshop\Core\Field( 'productTitle' );
$product->oxarticles__oxshortdesc = new \OxidEsales\Eshop\Core\Field(
↪'shortdescription' );
$product->save();
```

In this example, it will be checked if this ID exists and if so, the record in the database will be updated with the new record.

Making a query

Using the `ResultsetInterface`:

```
$resultSet = \OxidEsales\Eshop\Core\DatabaseProvider::getDb()->select($query);
//Fetch the results row by row
if ($resultSet != false && $resultSet->count() > 0) {
    while (!$resultSet->EOF) {
        $row = $resultSet->getFields();
        //do something
        $resultSet->fetchRow();
    }
}
```

Using the method `ResultsetInterface::fetchAll()`:

```
$resultSet = \OxidEsales\Eshop\Core\DatabaseProvider::getDb()->select($query);
//Fetch all at once (beware of big arrays)
$allResults = $resultSet->fetchAll()
foreach($allResults as $row) {
   //do something
};
```

Important: do not try something like this, you will lose the first result row:

```
 $resultSet = \OxidEsales\Eshop\Core\DatabaseProvider::getDb()->select($query);
 while ($row = $resultSet->fetchRow()) {
        //do something
 };

The point is: the ResultSet immediately executes the first call to␣
↪ResultSet::fetchRow() in its constructor, and
each following call to ResultSet::fetchRow() advances the content of␣
↪ResultSet::fields to the next row.
Do always access ResultSet::fields before calling ResultSet::fetchRow() again.
```

Transactions

If one transaction fails, the whole chain of nested transactions is rolled back completely. In some cases it might not be evident that your transaction is already running within an other transaction.

An example how to catch exceptions inside a database transaction:

```
// Start transaction outside try/catch block
$database->startTransaction();
try {
    $database->commitTransaction();
} catch (\Exception $exception) {
    $database->rollbackTransaction();
    if (!$exception instanceof DatabaseException) {
    throw $exception;
    }
}
```

MySQL master slave

Doctrine DBAL handles the master slave replication for the OXID eShop on each request. OXID eShop 6 follows these rules:

- once the request was routed to the master, it stays on the master.
- writes and transactions go to master.

If you are not careful in using the OXID eShop database API, this can lead .e.g to execute more requests than necessary on the MySQL master sever and underutilize the MySQL slave server.

Different API methods for read and write

There is a difference between the methods `DatabaseInterface::select()` and `DatabaseInterface::execute()` The method `DatabaseInterface::select()` can only be used for read alike methods (SELECT, SHOW) that return a kind of result set. The method `DatabaseInterface::execute()` must be used for write alike methods (INSERT, UPDATE, DELETE) in OXID eShop 6.

Namespaces with OXID eShop and modules

Topics to be covered

- **The backwards compatibility layer**
  - the *Unified Namespace*
  - find the *Unified Namespace* equivalents for the old bc classes (like oxarticle)
  - how we marked classes that are not intended to be extended by a module
- **Module installation**
  - old style (copy & paste)
  - new style (via composer)
- **How to extend the OXID eShop's namespaced classes**
  - in case your module does not yet use a namespace
  - in case your module does use it's own namespace
- **Use your own namespaces in a module with OXID eShop**
  - Install the module via composer or alternatively how to register your namespace in the main composer.json
  - Use own module classes
  - Use module controllers that do not simply extend existing shop functionality

Introduction

The following part of the documentation will cover the namespaces and what this means for a module developer. In short: we introduced namespaces in all the OXID eShop's core classes so that composer autoloader can be used.

You are able to extend the oxSomething classes (like oxarticle) in your module but we do not recommend this for new code. When we moved the OXID eShop's oxSomething classes under namespace we not only removed the 'ox' Prefix from the class name but gave some classes better suited names. (e.g. the former `sysreq` class now is named `OxidEsales\Eshop\Application\Controller\Admin\SystemRequirements`, all controller classes now have the postfix 'Controller' in their name). We will tell you how to find the new class names a bit later in this documentation.

NOTE: We now did physically remove the deprecated oxSomething bc classes (by that we mean all the old OXID eShop classes from before namespace era) while still offering backwards compatibility in case your module still relies on the old style class names. This BC layer is planned to be removed at some future time but you will have more than enough time to port your modules before that will happen.

NOTE: In order to use composer autoload, folder structure and class files needs to match the namespace (`UpperCamelCase`).

The Unified Namespace (`OxidEsales\Eshop`)

The *Unified Namespace* (`OxidEsales\Eshop`) provides an edition independent namespace for module and core developers.

Important: Please do not use the shop classes from the edition namespaces in your code! (More info)

NOTE: If you want to refer to a class name, always use the '::class' notation instead of using a plain string.

```
Example:

$articleFromUnifiedNamespace =␣
↪oxNew(\OxidEsales\Eshop\Application\Model\Article::class);
//which is equivalent to the old style
$articleFromBcClass = oxNew('oxarticle');
```

Equivalents for the old bc classes

See CE file `CoreAutoloadBackwardsCompatibilityClassMap.php`, which is an array mapping the *Unified Namespace* class names to the pre OXID eShop namespace class names (what we call the bc class names here). If you write a new module, please use the *Unified Namespace* class names as the bc class names are deprecated and should not be used for new code.

The OXID eShop itself still uses the old bc class names in some places but this will change in the near future.

Classes that are not to be extended by a module

We mark all classes that are not to be overwritten by a module with **@internal** but apart from that there is currently no mechanism that prevents a module developer from trying to extend such a shop class. We do not guarantee that the shop will work as expected if you try to do that though. What can definitely not be extended by a module is the `OxidEsales\Eshop\Core\UtilsObject` class.

Module installation

Installing a module can be done as before by copying the module sources into the shop's module directory (old style) and then activating the module in the shop admin backend. With namespaces in OXID eShop we have the possibility to let composer handle retrieving and copying the module sources to the correct location for you. You still have to activate the module in the shop admin either way.

Just create a composer.json in the module's root directory

```
{
    "name": "myvendor/mymodule",
    "extra": {
        "oxideshop": {
            "target-directory": "myvendor/mymodule"
        }
    }
}
```

Go to the shop's root directory and configure/require the module in the shop's composer.json.

```
composer config repositories.myvendor/mymodule vcs https://github.com/myvendor/
↪mymodule
composer require myvendor/mymodule:dev-master
```

The module sources now are located in the directory modules/myvendor/mymodule. Keep in mind that any changes made in the module directory itself will be overwritten with the next call to composer update (composer prompts for confirm though).

Extend an OXID eShop class with a module

If you want to adjust a standard OXID eShop class with a module (let's chose `OxidEsales\Eshop\Application\Model\Article` formerly known as `oxarticle` for example), you need to extend the module class (let's say `MyVendorMyModuleArticle`) from a *Unified Namespace* parent class (`MyVendorMyModuleArticle_parent`). The shop creates the class chain in such a way that once your module is activated, all methods from the `OxidEsales\Eshop\Application\Model\Article` are available in `MyVendorMyModuleArticle` and can be overwritten with module functionality.

IMPORTANT: It is only possible to extend shop BC and *Unified Namespace* classes. Directly extending classes from the shop edition namespaces is not allowed and such a module can not be activated. Trying to activate it gives an error in the admin backend.

No own module namespace

Create a module class that extends `OxidEsales\Eshop\Application\Model\Article`, for example

```
<?php
 # Example for a module without own namespace
 class MyVendorMyModuleArticle extends MyVendorMyModuleArticle_parent {

     public function getSize()
     {
         $originalSize = parent::getSize();

         //double the size
         $newSize = 2 * $originalSize;

         return $newSize;
     }
 }
```

Backwards compatible way, not recommended when writing new code:

```
# Register the extend class in the module's metadata.php
# Here we extend the shop's OxidEsales\Eshop\Application\Model\Article via the bc␣
↪class name
//.....
'extend'      => array(
    'oxarticle' => 'myvendor/mymodule/Application/Model/MyVendorMyModuleArticle'
)
//.....
```

The **recommended way to extend a shop core class with a module** in OXID eShop when the module does not support namespaces yet is as follows:

```
# Register the extend class in the module's metadata.php
//.....
'extend'      => array(
     \OxidEsales\Eshop\Application\Model\Article::class =>
             'myvendor/mymodule/Application/Model/MyVendorMyModuleArticle'
)
//.....
```

Use your own namespaces with OXID eShop

Now create a class like before to extend a shop class but this time give it a namespace:

```
<?php
 # Example for module with own namespace

 namespace MyVendor\MyModuleNamespace\Application\Model;

 class MyModuleArticle extends MyModuleArticle_parent
 {
     public function getSize()
     {
         $originalSize = parent::getSize();

         //double the size
         $newSize = 2 * $originalSize;

         return $newSize;
     }
 }
```

Register the class in the module's metadata,php:

```
# Register the extend class in the module's metadata.php
//.....
'extend'      => array(
     \OxidEsales\Eshop\Application\Model\Article::class =>
          MyVendor\MyModuleNamespace\Application\Model\MyModuleArticle::class
)
//.....
```

Install and register your module with composer

To have the composer autoloader find your module file via namespace, create a composer.json file in the module's root directory.

```
{
    "name": "myvendor/mymodule",
    "autoload": {
        "psr-4": {
            "MyVendor\\MyModuleNamespace\\": "./"
        }
    },
    "extra": {
        "oxideshop": {
```

```
            "target-directory": "myvendor/mymodule"
        }
    }
}
```

Then in the shop's root directory do

```
composer config repositories.myvendor/mymodule vcs https://github.com/myvendor/
↪mymodule
composer require myvendor/mymodule:dev-master
```

and run composer update.

In case you do not want to handle module installation with composer but copy & paste it old style into the shop's module directory, register your module namespace directly in the shop's main composer.json:

```
"autoload": {
     "psr-4": {
         "OxidEsales\\EshopCommunity\\": "./source",
         ....
         "MyVendor\\MyModuleNamespace\\": "./source/modules/myvendor/mymodule"
     }
 }
```

And then run composer update so composer can update it's autoload file.

Using namespaces in module classes that do not extend OXID eShop classes

Add for example a model class to your module:

```
<?php
namespace MyVendor\MyModuleNamespace\Application\Model;

class MyModuleModel
{
    public function doSomething()
    {
        //.....
        // do something
        //......
        return $someResult;
    }
}
```

There is no need to register this class in the metadata.php as the composer autoloader will do the trick.

```
<?php
 namespace MyVendor\MyModuleNamespace\Application\Controller;

 use MyVendor\MyModuleNamespace\Application\Model\MyModuleModel;

 class MyModulePaymentController extends MyModulePaymentController_parent
 {
     public function render()
     {
         $template = parent::render();
         //.....
```

```
        $model = new MyModuleModel;
        $someResult = $model->doSomething();
        // do something else
        //......
        return $template;
    }
```

or with oxNew instead of new

```
<?php
 namespace MyModuleNamespace/Application/Controller;

 class MyModulePaymentController extends MyModulePaymentController_parent
 {
     public function render()
     {
         $template = parent::render();
         //.....
         $model =␣
↪oxNew(\MyVendor\MyModuleNamespace\Application\Model\MyModuleModel::class);
         $someResult = $model->doSomething();
         // do something else
         //......
         return $template;
     }
```

In the module's metadata you only need to register the class extending the shop's payment controller but not your module's new model class.

```
# Register the extend class in the module's metadata.php
//.....
'extend'      => array(
    \OxidEsales\Eshop\Application\Controller\PaymentController::class
        =>␣
↪MyVendor\MyModuleNamespace\Application\Controller\MyModulePaymentController::class
)
//.....
```

Use module controllers that do not simply extend existing shop functionality

In case you want to not only extend shop functionality in a module but for example want to introduce a new controller that handles own form data we recommend you have a look into what changed with module metadata version 2.0. In short: in case you want introduce controllers in your module that support namespaces and that do not simply extend shop functionality, you need to use metadata version 2.0 and register these controller classes in the module's metadata.php file.

More information regarding this topic can be found here.

Theme resources

Theme Configuration

Possibility to configure theme was added with the pull request #381

Example how to change settings for already existing theme as it was done in the Flow Theme

How to create a theme installable via composer?

Themes are installed via Composer by using OXID eShop Composer Plugin.

In order to install theme correctly this plugin requires two fields to be described in theme `composer.json` file:

- *type*
- *extra*

Flow theme example:

```
{
   "name": "oxid-esales/flow-theme",
   "description": "This is Flow theme for OXID eShop.",
   "type": "oxideshop-theme",
   "keywords": ["oxid", "themes", "eShop"],
   "homepage": "https://www.oxid-esales.com/en/home.html",
   "license": [
       "GPL-3.0",
       "proprietary"
   ],
   "extra": {
     "oxideshop": {
       "target-directory": "flow",
       "assets-directory": "out/flow"
     }
}
```

type

Theme must have `oxideshop-theme` value defined as a type. This defines how the repository should be treated by the installer.

extra: {oxideshop}

target-directory

`target-director` value will be used to create a folder inside the Shop `Application/views` directory. This folder will be used to place all files of the module.

assets-directory

Defines where public resources like `css, js, images` are placed inside the theme. The plugin will copy those files to the Shop `out` directory.

Note: It is recommended to keep assets in out directory at a root level of the repository.

System Architecture

Autoloading Of Classes

Currently shop has three autoloaders registered: Composer autoloader, Backwards Compatibility Autoloader and Module Autoloader. They are registered in exactly this order in the file `bootstrap.php`.

General workflow

If you request a class, then first the Composer autoloader is asked, after that the Backwards Compatibility Autoloader and in the end the Module Autoloader:

Composer Autoloader

It is the first autoloader in line and tries to to autoload all namespaced classes, which are configured in the root `composer.json` file or child `composer.json` files. An example of a class which would be resolved by this autoloader is `OxidEsales\Eshop\Application\Model\Article`.

Backwards Compatibility Autoloader

Its purpose is to autoload all deprecated shop classes which are defined in the file `Core/Autoload/BackwardsCompatibilityAutoload.php`. This is not a real autoloader: If a backwards compatibility class from `Core/Autoload/BackwardsCompatibilityAutoload.php` is requested, this autoloader searches the Unified Namespace equivalent of the backwards compatible class and hands the request over to the Composer autoloader. If you request e.g. the backwards compatiblity class `oxArticle`, this autoloader would resolve the class to its unified namespace equivalent `OxidEsales\Eshop\Application\Model\Article` and trigger the composer autoloader.

Module Autoloader

This autoloader is responsible for loading module classes (defined in metadata as module files and extensions). It first checks if given class exists in any of active modules module file. If so - this class is included and it stops here. If not - it tries to check whether it is an extension of any active module, as modules can extend other module classes. This is also the case when extension is created via `new ExtendedClass` instead of `oxNew`, and as ExtendedClass_parent class does not exist, it has to be created at this point.

Unified Namespace Classes

The *Unified Namespace* (`OxidEsales\Eshop`) provides an edition independent namespace for module and core developers. So disregarding if the shop edition is CE/PE/EE, the *Unified Namespace* class name is to be used in code (core and modules).

Generation of unified namespace classes

The component *unified-namespace-generator* generates the unified namespace classes on the fly, e.g. when you install or update the OXID eShop.

Inheritance chain of unified namespace classes

Example OXID eShop Professional Edition

Example OXID eShop Enterprise Edition

Example OXID eShop Enterprise Edition with 2 modules activated

Warning: Do NOT use the PHP method `get_class` as its return value is dependent on the modules which are currently activated in the shop:

```
// returns Vendor1\Module2\Application\Model\Article in this example
get_class(oxNew(OxidEsales\Eshop\Application\Model\Article::class));
```

Update

Update OXID eShop from version 4.10 / 5.3 to version 6.0.0

This guide acts on the assumption you have an already running OXID eShop 4.10 / 5.3 and want to update it to OXID eShop 6.0.0. If you do not have the newest patch release of 4.10 / 5.3, you should first update to this patch release. A helping approach regarding the update is to deactivate all modules and then successively activate them after the update. Read the documentation about all changes carefully. In some cases you have to take actions, in some cases not. This depends on your OXID eShop, the modules, user data or the theme you are using. Please also have a look at the source code documentation for deprecated classes and methods. The update manual is divided into several chapters:

Database

Tables and fields

Before starting with the changes described in the following sections, you should make sure that your OXID eShop 4.10 / 5.3 is running on utf-8 database tables. See here for migration instructions. You should also take care that your own tables use UTF-8. There are also exceptions from utf-8 in the OXID eShop database tables (e.g. the column `OXID` which is latin1 in most tables). If you refer to those columns from your own tables, you also have to use latin1.

In order to do update the database the update, you have to

1. Execute the `migrate_XXX.sql` files described in the following
2. Run *database migrations* in OXID eShop 6 via the command

```
vendor/bin/oe-eshop-db_migrate migrations:migrate
```

For step 1, we provide update SQL scripts for each OXID eShop edition. We divided them into two files:

1. queries, where you can not lose data while the execution and
2. queries, where you will lose data while the execution.

So we expect, that you read the second file especially carefully!

You will recognize the second file on its postfix '_cleanup'.

OXID eShop Community Edition:

1. `migrate_ce_5_3_to_6_0.sql`
2. `migrate_ce_5_3_to_6_0_cleanup.sql`

OXID eShop Professional Edition:

1. `migrate_pe_5_3_to_6_0.sql`
2. `migrate_pe_5_3_to_6_0_cleanup.sql`

OXID eShop Enterprise Edition

1. `migrate_ee_5_3_to_6_0.sql`
2. `migrate_ee_5_3_to_6_0_cleanup.sql`

InnoDb: Change of database engine

The database engine in OXID eShop 4.10 / 5.3 is mostly MyISAM. In OXID eShop 6, the database engine is InnoDB for all database tables.

- Migrating the database with the scripts (see the previous section) from MyISAM to InnoDb may need some time, additional disk space and RAM. Be sure to plan a maintenance window in your production shop, provide enough disk space and RAM on your MySQL server.
- If you implemented your own queries to OXID eShop database tables, be sure to sort the results explicitely (e.g. using the MySQL `ORDER BY`). Otherwise the order of the results may change with the migration from MyISAM to InnoDB.

Database API

Read these changes carefully if you implemented own database queries. Otherwise you can skip this section.

New interfaces

OXID eShop 4.10 / 5.3 introduced new interfaces: the `\OxidEsales\Eshop\Core\Database\Adapter\DatabaseInterfa` and the `\OxidEsales\Eshop\Core\Database\Adapter\ResultSetInterface`. Be aware that there are already deprecated methods in the interfaces in OXID eShop 4.10 / 5.3 which were removed in OXID eShop 6. Hints for replacing those methods in your code will be shown in the following sections.

DatabaseInterface

- the function parameter `$executeOnSlave` for some functions is deprecated in OXID eShop 5.3. You could additionally call `DatabaseInterface::forceMasterConnection()` before or encapsulate your logic in a transaction. Both mechanisms will force SQL queries to be read from the master server from this point on. This was done due to the changed MySQL master slave handling in OXID eShop 6. See the section *Master slave* for details.

- the constant `DatabaseInterface::FETCH_MODE_DEFAULT` shouldn't be used any more. Doctrine uses `FETCH_MODE_BOTH` by default.
- The database transaction isolation level is set on session scope, not globally any more. Have a look at the comments of the method `DatabaseInterface::setTransactionIsolationLevel()`.

ResultSetInterface

- there is no way any more to move the pointer inside the resultSet any more in OXID eShop 6. The related methods will be removed completely. Do not use them, there is no elegant replacement.
  - `ResultSetInterface::move()`
  - `ResultSetInterface::moveNext()`
  - `ResultSetInterface::moveFirst()`
  - `ResultSetInterface::moveLast()`
  - `ResultSetInterface::_seek()`
  - `ResultSetInterface::EOF()`

    Deprecated (5.3) logic, does not work in 6.0 and higher any more:

```
$rs = oxDb::getDb()->select($sQuery);
if ($rs != false && $rs->recordCount() > 0) {
    while (!$rs->EOF) {
        //do something
        $rs->moveNext();
    }
}
```

    Example: new (since 6.0) logic

```
$resultSet = \OxidEsales\Eshop\Core\DatabaseProvider::getDb()->select($query);
 //Fetch the results row by row
 if ($resultSet != false && $resultSet->count() > 0) {
     while (!$resultSet->EOF) {
         $row = $resultSet->getFields();
         //do something
         $resultSet->fetchRow();
     }
 }
```

- the following methods can be replaced with `ResultSetInterface::fetchAll()` in OXID eShop 6 to retrieve all rows or ResultSetInterface::fetchRow() to retrieve a single row:
  - `ResultSetInterface::getAll()`
  - `ResultSetInterface::getArray()`
  - `ResultSetInterface::getRows()`
- The methods, which are related to the `ADODB lite ResultSet *fields*` property meta data were completely removed in OXID eShop 6.
  - `ResultSetInterface::fetchField()` Do not use any more.
  - `ResultSetInterface::fields($field)` Do not use any more.

- `ResultSetInterface::recordCount()` will be removed completely. Do not retrieve the affected row in the `RecordSet`, but in the `DatabaseInterface`.
- The methods `DatabaseInterface::select()` and `DatabaseInterface::selectLimit()` now return an object of the type `ResultSetInterface`.

More examples how to use the database, *can be found here.*

Difference between read and write methods

In OXID eShop 4.10 / 5.3 you can use the methods `execute` and `select` synonymously. In OXID eShop 6, the method `DatabaseInterface::select()` can only be used for read alike methods (`SELECT`, `SHOW`) that return a kind of result set. The method `DatabaseInterface::execute()` must be used for write alike methods (`INSERT`, `UPDATE`, `DELETE`) in OXID eShop 6. See the section *Master slave* for details.

Transactions

If you use transactions in your database queries, please read this section. The transaction handling has changed substantially in OXID eShop 6:

- nested transactions are possible now. If one transaction fails, the whole chain of nested transactions is rolled back completely. In some cases it might not be evident that your transaction is already running within an other transaction.
- as all OXID eShop tables now support InnoDb, transactions are possible on all OXID eShop tables.

For details have a look on the *transactions documentation*

ADOdb Lite

The library for the database abstraction layer (DBAL) changed from ADOdb Lite in OXID eShop 4.10 / 5.3 to Doctrine DBAL in OXID eShop 6.

As using the library *ADOdb Lite* directly was not recommended at any time, you should not have to take care for this change.

Log MySQL queries

The possibility to log MySQL queries was removed. There is no explicit recommendation on how to replace this feature in your OXID eShop.

Session storage

The possibility to save sessions to the eShop application database was removed. A blog post about the impact and alternatives in OXID eShop 6 and can be found on oxidforge.

Master slave

The implementation and usage of MySQL master slave replication changed in OXID eShop 6. This results in the following changes:

- the parameter `executeOnSlave` was deprecated in OXID eShop 4.10 / 5.3. Have a look at the section *Database API* on how to avoid `executeOnSlave`.
- the configuration parameter `iMasterSlaveBalance` was used in OXID eShop 4.10 / 5.3 to balance the amount of read accesses between master and slave(s). Due to differences in now letting Doctrine DBAL handle Master/Slave connections the balance feature cannot be supported anymore.
- as the ratio between master and slave utilisation can vary between an OXID eShop 4.10 / 5.3 and an OXID eShop 6, you have to review your master slave concept with OXID eShop 6.
- for database queries in modules please have a look at the *database documentation*.

Files

This section describes the steps to update the file structure from a OXID eShop version 4.10 / 5.3 to version 6. As there are many changes in the file structure, the approach for the update is:

1. setup an OXID eShop 6 in parallel to your existing OXID eShop 4.10 / 5.3
2. copy the files described in the following sections from the OXID eShop 4.10 / 5.3 to the OXID eShop 6

Please always pay attention to upper and lower case letters in file and directory names.

Own Scripts And / Or Configuration

- use UTF-8 encoding for all your scripts
- if you made changes to `.htaccess` files in OXID eShop 4.10 / 5.3, port them to the equivalent `.htaccess` files in OXID eShop 6. Pay attention to the fact that the `.htaccess` files in OXID eShop 6 are compatible with Apache 2.2 and 2.4 where OXID eShop 4.10 / 5.3 `.htaccess` file were only compatible with Apache 2.2.
- if you made changes to the file `config.inc.php` in OXID eShop 4.10 / 5.3, port them to the file `config.inc.php` in OXID eShop 6

Languages

If you added a new language (additionally to the languages `de` and `en`) in OXID eShop 4.10 / 5.3, you have to port this language to OXID eShop 6 because many language constants changed. In order to port the language, you have to either

- replace the language files by downloading an OXID eShop 6 compatible language pack. E.g. from a 3rd party vendor or via translate.oxidforge.org.
- or copy and update the language files manually.

Language related files reside in the following directories (also see OXIDprojects/languages for a language pack example):

- `application/translations` in OXID eShop 4.10 / 5.3 respectively `Application/translations` in OXID eShop 6
- `application/views/admin` in OXID eShop 4.10 / 5.3 respectively *Application/views/admin* in OXID eShop 6
- `application/views/yourThemeName` in OXID eShop 4.10 / 5.3 respectively `Application/views/yourThemeName` in OXID eShop 6
- `out/yourThemeName` in OXID eShop 4.10 / 5.3 and also `out/yourThemeName` in in OXID eShop 6

- `setup` in OXID eShop 4.10 / 5.3 respectively `Setup` in OXID eShop 6

Smarty Plugins

If you created own Smarty plugins in OXID eShop 4.10 / 5.3 and installed them by copying them to the folder `core/smarty/plugins`, move them to the folder `Core/Smarty/Plugins` in OXID eShop 6.

Folder out

Copy the files from the folders

- `out/downloads`
- `out/media`
- `out/pictures` (except `out/pictures/wysiwygpro` and `out/pictures/generated`)

to the equivalent folders in OXID eShop 6. For updating the images used in WYSIWYG Pro, *see this section.*

Log-Files

Copy all log files from the directory `log`. Do not copy the standard `.htaccess` files. If you made changes to `.htaccess` files in OXID eShop 4.10 / 5.3, port them to the equivalent `.htaccess` files in OXID eShop 6.

Folders bin / export / log / export

Copy the files from these directories. Do not copy the standard `.htaccess` files. If you made changes to `.htaccess` files in OXID eShop 4.10 / 5.3, port them to the equivalent `.htaccess` files in OXID eShop 6.

Modules

- if you made changes to the file `modules/composer.json` in OXID eShop 4.10 / 5.3, port those changes into the root `composer.json` file in OXID eShop 6 or into a modules `composer.json` file
- if you made changes to the file `modules/functions.php` in OXID eShop 4.10 / 5.3, port those changes into the equivalent file `modules/functions.php` file in OXID eShop 6

For updating a module itself, have a look at the *Guideline for porting modules to OXID eShop version 6.0*

Modules

For updating existing modules from OXID eShop 5.4 to OXID eShop 6, either

- get an OXID eShop 6 compatible version of your modules or
- update the modules by yourself. Please have a look at the following sections on how to update by yourself.

Overview about the steps to port a module to the OXID eShop version 6.0

In the table below you can find an overview what steps you can, and at least have to do, to port your module to the OXID eShop version 6.0. Every line of the table represents a step or an adaption. As you see, there are two columns named "Minimal" and "Full". Your absolute to-dos for now are marked as "Minimal" with a "". They tell you, that you have to do them in order to end up with a module which works with the OXID eShop version 6.0. All to-dos are marked as "Full". This tells you, that you are not done after the "Minimal" porting of your module. There are more steps to make to be fully aligned with the version 6.0. We strongly recommend you to do the "Full" steps now, or as soon as possible. We do so, cause

- you will fit better in OXIDs long term stable investment strategy and
- with the next (major) versions there will be more changes, which wll add up to a bigger amount of open to-dos.

Topic	Minimal	Full
Assure test coverage for your code		
Convert all files to UTF-8		
Adjust PHP version		
Adjust removed functionality		
Adjust your database code to the new DB Layer		
Adjust the code style of your modules code		
Exchange BC Layer classes		
Remove deprecated code		
*Installable via composer**		
Introduce a namespace in your module		

(*) If you are maintaining a module which is part of the *OXID eShop Compilation* the installation has to work via composer!

Minimal steps

This section describes the minimum changes, which are necessary to make an existing module compatible with OXID eShop version 6.0.

Cover your code with tests

Make sure that you have all logic covered by tests - Unit, Integration, Acceptance. Let them run once after every step in this guide. We will not go into detail why this is important for you and the health of your module, cause this question was discussed millions of times. You can easily find stuff about this topic.

UTF-8 only

Starting with the 6.0 the OXID eShop is UTF-8 only. This means all your modules

- Translation files,
- SQL files,
- Code files,
- Test files,
- and all other files

have to be UTF-8 encoded.

Required PHP version

The code must work with PHP 5.6 and higher. Check the official PHP migration documentation on php.net what you have to do.

Removed functionality in OXID eShop

Make sure your module does not use any of the functionality that was deprecated in 5.3 and has been removed in OXID eShop 6.0. You can find a list of changes in OXID Forge.

Stick to database interfaces

Especially have an eye on the changes in database layer. ADOdb Lite (OXID eShop 5.x) was exchanged in favour of Doctrine/DBAL which leads to some slightly different behaviour in some cases. We had to introduce some backwards compatibility breaks there.

Check 5.3 code for what will be deprecated:

- OXID eShop 5.3 ResultSetInterface
- OXID eShop 5.3 DatabaseInterface

New equivalents:

- OXID eShop 6.0 ResultSetInterface
- OXID eShop 6.0 DatabaseInterface

In ADOdb Lite there was not such a thing as a ResultSetInterface, it was introduced in v5.3.0 to be able to have an upgrade path to the version 6.0.

IMPORTANT: Return values of e.g. oxDb::getDb()->select() and oxDb::getDb()->selectLimit() have changed, now an instance of ResultSet (implementing ResultSetInterface) is returned.

Deprecated (5.3) logic, does not work in 6.0 and higher any more:

```
$rs = oxDb::getDb()->select($sQuery);
if ($rs != false && $rs->recordCount() > 0) {
    while (!$rs->EOF) {
        //do something
        $rs->moveNext();
    }
}
```

Example: new logic (since 6.0)

```
$resultSet = \OxidEsales\Eshop\Core\DatabaseProvider::getDb()->select($query);
//Fetch the results row by row
if ($resultSet != false && $resultSet->count() > 0) {
    while (!$resultSet->EOF) {
        $row = $resultSet->getFields();
        //do something
        $resultSet->fetchRow();
    }
}
```

Another example: new logic (since 6.0)

```
$resultSet = \OxidEsales\Eshop\Core\DatabaseProvider::getDb()->select($query);
//Fetch all at once (beware of big arrays)
$allResults = $resultSet->fetchAll()
foreach($allResults as $row) {
   //do something
};
```

IMPORTANT NOTE: do not try something like this, you will lose the first result row:

```
$resultSet = \OxidEsales\Eshop\Core\DatabaseProvider::getDb()->select($query);
while ($row = $resultSet->fetchRow()) {
       //do something
};
```

What will happen: the ResultSet immediately executes the first call to ResultSet::fetchRow() in its constructor and each following call to ResultSet::fetchRow() advances the content of ResultSet::fields to the next row. Always access ResultSet::fields before calling ResultSet::fetchRow() again.

Full steps

On top of the *minimal steps* we recommend you to take the following steps to completely move your module to the version 6.0 of the OXID eShop.

Code style

From OXID eShop version 6.0 on PSR-0 and PSR-4 standards will be used in OXID eShop core code. Our Codesniffer can help you achieving this goal.

Backwards compatibility layer and Unified Namespace

Mind that from version 6.0 on the OXID eShop is using namespaces. Therefore nearly all classes known from 5.3 (e.g. `oxArticle`) and previous versions are deprecated now. They exist only as aliases in which we call the Backwards Compatibility Layer (from now on abbreviated with *BC Layer*).

As long as the *BC Layer* is in place, you can use the backwards compatibility classes (e.g. `oxArticle`) equivalent to the actual classes from the *Unified Namespace* (e.g. `\OxidEsales\Eshop\Application\Model\Article`). The *Unified Namespace* is an abstraction for classes which exist in several Editions of the OXID eShop. As soon as the *BC Layer* is dropped in a future release of OXID eShop, you will have to fully port your module to the new Unified Namespaced classes (see *Unified Namespace*).

Replace all OXID eShop backwards compatibility classes (e.g. `oxArticle`) in your module by the equivalent fully qualified *Unified Namespace* classes.

- **check usages in oxNew and new**

```
// Old style (using BC Layer)
$article = oxNew('oxarticle');
$field = new oxField();

// New style:
$article = oxNew(\OxidEsales\Eshop\Application\Model\Article::class);
$field   = new \OxidEsales\Eshop\Core\Field();
```

- **Use the *Unified Namespace* class names for calls to Registry::set() and Registry::get().**

```
// Old style:
oxRegistry::get('oxSeoEncoderVendor');

// New style:
\OxidEsales\Eshop\Core\Registry::get(\OxidEsales\Eshop\Application\Model\SeoEncoderVendor::c
↪
```

Remove deprecated code

Besides the usage of *backwards compatibility classes* there might exist more usages of deprecated code in your modules. Choose your favourite IDE (integrated development environment) and do a code analysis on deprecations. Additionally you can have a look to a list of all deprecations in the *source code documentation <http://docu.oxid-esales.com/CE/sourcecodedocumentation>*.

Make module installable via composer

We recommend that the module is made installable via composer. Modules that will go to the (*OXID eShop Compilation*) **MUST** be installable via composer. Information what needs to be done (the keyword is composer.json) can be found *here*. Verify that composer correctly installs it.

Important: if you made changes to the file `modules/composer.json` in OXID eShop 4.10 / 5.3, port those changes into the root `composer.json` file in OXID eShop 6 or into a modules `composer.json` file

---

Move the module under a module namespace

- Introduce the module namespace in the module's composer.json file's autoload section.

```
"autoload": {
    "psr-4": {
        "MyVendor\\MyModuleNamespace\\": "../../../source/modules/
↪myvendor/mymoduleid"
    }
}
```

NOTE: we recommend to point the namespace to the module's installation path in the shop's module directory. See for example OXID eShop Extension PayPal.

```
"autoload": {
    "psr-4": {
        "OxidEsales\\PayPalModule\\": "../../../source/modules/oe/oepaypal
↪"
    }
}
```

Use the following pattern for your module namespace: `<vendor of the module>`\<module ID> (e.g. `OxidEsales\PayPalModule`)

You can find more about the *Vendor Id* in the Glossary.

- Move all the module classes under namespace.

```
//before:
class oePayPalIPNHandler extends oePayPalController
{
    //...
}

$handler = oxNew('oepaypalipnhandler');
```

```
//after:
namespace OxidEsales\PayPalModule\Controller;
class IPNHandler extends␣
↪\OxidEsales\PayPalModule\Controller\FrontendController
{
     //...
}

$handler = oxNew(\OxidEsales\PayPalModule\Controller\IPNHandler::class);
```

While this step you should exchange all occurrences of the files name. Especially in the metadata.php the 'extends' section should not be forgotten! Remove the entry from the 'files' section, after you moved the class into the namespace. It is not longer needed, cause the namespaces get autoloaded via composer.

- Update metadata.php to version 2.0, see *here*. In case the module uses it's own controllers that do not simply chain extend shop controllers, you need to register a controller key in the metadata.php 'controller' section like described *here*.

```
'controllers' => array(
    ...
    'oepaypalipnhandler' => \OxidEsales\PayPalModule\Controller\IPNHandler::class,
    ...
),
```

Your Controller Keys have to be lowercase and have to follow this pattern: `<vendor of the module><module ID><controller name>` (e.g. `oepaypalipnhandler`)

Theme

Depending on if you use the old deprecated theme azure or the new standard theme flow in OXID eShop 4.10 / 5.3, you have to take different actions.

Theme azure

If you use or extend the deprecated theme azure in OXID eShop 4.10 / 5.3, we recommend to use or extend the new standard theme flow instead.

If you want to use still the theme azure, you have to include azure first in OXID eShop 6 like described here as it is not delivered by default any more. There is an version of the flow theme compatible to OXID eShop 4.10 / 5.3 and a version compatible to OXID eShop 6 like described here.

If you extended the azure theme with a custom theme, you have to update your custom theme as described in the section *Updating a custom theme*. Please also update your modules accordingly.

Theme flow

If you already use the theme `flow` in OXID eShop 4.10 / 5.3, you don't have to do anything. The flow theme is delivered by default with OXID eShop 6.

There is an OXID eShop 6 compatible version of the flow theme which has some differences to the version delivered in OXID eShop 4.10 / 5.3 like described here.

If you extended the flow theme in OXID eShop 4.10 / 5.3 you have to check the differences between the OXID eShop 4.10 / 5.3 compatible flow version and the OXID eShop 6 compatible flow version. Afterwards, update your custom theme as described in the section *Updating a custom theme*. Please also update your modules accordingly.

Updating a custom theme

In order to use your custom theme (name `yourThemeName` in this example) from OXID eShop 4.10 / 5.3 in OXID eShop 6, copy the folders

- `application/views/yourThemeName` from OXID eShop 4.10 / 5.3 to `Application/views/yourThemeName` in OXID eShop 6
- `out/yourThemeName` from OXID eShop 4.10 / 5.3 to the equivalent directory in OXID eShop 6

Afterwards you have to adapt your theme to the new version of its parent theme. Also copy the file `favicon.ico` from the shops root folder if you modified it.

Removed features and new features

Extracted features

Introduction to extracted features

Some features in the core of OXID eShop 4.10 / 5.3 were extracted into OXID eShop 6 compatible modules. If you used or extended one of those features in OXId eShop 4.10 / 5.3, you should read this document carefully. The mentioned OXID eShop modules are available on Github. If you want to contribute to the development of one of those modules, this is possible via a pull request.

In the following sections the affected features and the steps for migrating existing data are described. If you extended one of those features in OXID eShop 4.10 / 5.3, you have to extend the corresponding contribution module in OXID eShop 6. If you did not use or extend one of those features in OXID eShop 4.10 / 5.3, there is nothing to to.

Tags

The feature to tag products was was extracted to the module Tags.

Migration

Possible places for data migrations:

- In OXID eShop 4.10 / 5.3, the database table `oxartextends` had the columns `OXTAGS_*`. In order to migrate your existing tags, simply rename these columns in OXID eShop 6 to to `OETAGS_*`
- related functionality like the search might also be affected as it relies on the tags feature in OXID eShop 4.10 / 5.3

- the tag categories (`http://myoxideshop.com/tags/*`) are not available any more in OXID eShop 6
- regeneration of seo links (table `oxseo`) might be necessary
- the config variable `sTagSeparator` in OXID eShop 4.10 / 5.3 is called `oetagsSeparator` in the contribution module
- the config variable `blShowTags` (Display tags in eShop) is called in OXID eShop 4.10 / 5.3 and located in *Core Settings* → *Settings* → *Shop Frontend* in the OXID eShop admin. In the contribution module this setting is called `oetagsShowTags` and located in the modules setting tab.
- the config variable `aSearchCols` (fields to be considered in the Search) needs to be updated as it contains `oxtags` in OXID eShop 4.10 / 5.3.
- Tags related css classes were removed/renamed in the module.
- the Tags javascript widget was removed/renamed in the module.

Important: The performance of the Tags module might suffer as the module does not use the FULLTEXT feature of the database engine MyISAM any more.

Important: EE needs the EE_addon tags module in addition to work with varnish. (see installation instructions)

Lexware export / (XML export of orders)

The export of orders into XML documents (Lexware export) in OXID eShop 4.10 / 5.3 was extracted to the module Lexware Export.

Migration

There is a config option for VAT settings for the XML export. In OXID eShop 4.10 / 5.3, this option was called `aLexwareVAT` and located in the OXID eShop admin in *Core Settings* → *Settings* → *Other settings*. In OXID eShop 6, this option is called `aOELexwareExportVAT` and you will find this setting in the settings tab of the Lexware export module. Be sure to migrate your settings from this config option.

If you extended or modified this functionality or translations in OXID eShop 4.10 / 5.3, you have to port your changes.

Extended Order administration (Order Summary And Pick Lists)

The extended order administration feature of OXID eShop 4.10 / 5.3 was extracted to the module Extended Order Administration.

If you extended or modified this functionality or translations in OXID eShop 4.10 / 5.3, you have to port your changes.

Statistics

The statistics feature of OXID eShop 4.10 / 5.3 (e.g statistics about conversion rate, number of visitors) was extracted to the module Statistics.

Migration

In OXID eShop 4.10 / 5.3, the statistics were stored in the tables `oxlogs` and `oxstatistics`. In OXID eShop 6, they are stored in the tables `oestatisticslog` and `oestatistics`. In order to migrate your existing entries, simple copy and rename the tables `oxlogs` and `oxstatistics`.

If you extended or modified this functionality, translations or database tables in OXID eShop 4.10 / 5.3, you have to port your changes.

Facebook

The Facebook feature of OXID eShop 4.10 / 5.3 was extracted to the module Facebook Social Plugins. If you extended or modified this functionality or translations in OXID eShop 4.10 / 5.3, you have to port your changes.

Important: The Facebook functionality in OXID eShop 4.10 / 5.3 used an old version of the Facebook API and therefor partly did not work. Our recommendation is to use a third party module for facebook integration.

Captcha

The captcha feature of OXID eShop 4.10 / 5.3 was extracted to the module Captcha. If you extended or modified the captcha functionality, the database table `oxcaptcha` or translations in OXID eShop 4.10 / 5.3, you have to port your changes.

Important: Our recommendation is to use a third party module for captcha functionality as there are more advanced approaches.

Guestbook

The guestbook feature of OXID eShop 4.10 / 5.3 was replaced by the module Guestbook module.

Important: Currently it's not possible to use this feature in the Enterprise Edition, the module is for Community and Professional Edition only at the moment.

Migration

- In OXID eShop 5.3, the guestbook entries were stored in the table `oxgbentries`. In OXID eShop 6, they are stored in the table `oeguestbookentry`. In order to migrate your existing guestbook entries, simple copy and rename the table `oxgbentries`.
- There are config options for the maximum guestbook entries a user can write per day and if you want to moderate the guestbook. In OXID eShop 4.10 / 5.3. these config options were called `iMaxGBEntriesPerDay` and `blGBModerate` (database table `oxconfig`). In the OXID eShop they were located in *Core Settings* → *Settings* → *Other settings*. In OXID eShop 6, you will find these settings in the settings tab of the guestbook module. They are called `oeGuestBookMaxGuestBookEntriesPerDay` and `oeGuestBookModerate`. Be sure to migrate your settings from these config options.
- seo links have to be regenerated

If you extended or modified the guestbook functionality, translations or seo settings in OXID eShop 4.10 / 5.3, you have to port your changes.

InvoicePDF left overs

In the version 4.10 / 5.3 of the OXID eShop PDF invoice generation was included. In OXID eShop 6 it is removed from the OXID eShop code and added as an own repository.

Dropped libraries

We dropped several libraries from the OXID eShop. If you used one of those libraries directly (not via OXID eShop API, which is not recommended by OXID eSales), you have to find a workaround or include the library via your projects root `composer.json` file. Several other libraries were moved to the `vendor` folder.

ADODb Lite

See further information about the therefore made changes.

JpGraph

JpGraph is a graph drawing library. In OXID eShop 4.10 / 5.3 the JpGraph library with the version 2.5 was included in the directory `core/jpgraph>`. If you somehow used the functionality of the JpGraph library, we recommend to require it via composer. There is a public available package which points to the JpGraph github repository.

facebook

As stated in *this section*, the facebook functionality was moved into a module.

smarty

We exchanged the smarty library from our code base with a composer required package. In the version 4.10 / 5.3 of the OXID eShop was used the smarty version 2.6.25. In the OXID eShop version 6.0 the smarty version 2.6.30 is used. It should not add much effort to update your code. But if something stops working, we recommend to look through the smarty documentation.

PHPMailer

We exchanged the PHPMailer library from our code base with a composer required package. Cause we sticked to the version of this library, there will be nothing to do left for you.

Miscelaneous changes

The following changes could, but don't have to be relevant for the update of your OXID eShop. Read them carefully and decide if you have to take actions.

Exception handling

The exception handler was refactored in a way to catch more exceptions than before. Therefor you should have a look at the file `log/EXCEPTION_LOG.txt` after you completed the whole update to OXID eShop 6. Goal should be to have no exceptions in this file.

If you configured exception handling by overwriting the method `oxShopControl::_setDefaultExceptionHandler()`, you can do this from now on by calling the PHP method `set_exception_handler()` in the file `modules/functions.php`.

The format of the file `log/EXCEPTION_LOG.txt` changed a little bit, e.g. a data is included now:

```
[10 Oct 16:44:44.625024 2017] [exception] [type Exception] [code 0] [file /var/www/
↪oxideshop/source/Application/Controller/StartController.php] [line 128] [message␣
↪Argument not valid]
[10 Oct 16:44:44.625024 2017] [exception] [stacktrace] #0 /var/www/oxideshop/source/
↪Core/ShopControl.php(466):␣
↪OxidEsales\EshopCommunity\Application\Controller\StartController->render()
[10 Oct 16:44:44.625024 2017] [exception] [stacktrace] #1 /var/www/oxideshop/source/
↪Core/ShopControl.php(357): OxidEsales\EshopCommunity\Core\ShopControl->_
↪render(Object(OxidEsales\Eshop\Application\Controller\StartController))
[10 Oct 16:44:44.625024 2017] [exception] [stacktrace] #2 /var/www/oxideshop/source/
↪Core/ShopControl.php(289): OxidEsales\EshopCommunity\Core\ShopControl->
↪formOutput(Object(OxidEsales\Eshop\Application\Controller\StartController))
[10 Oct 16:44:44.625024 2017] [exception] [stacktrace] #3 /var/www/oxideshop/source/
↪Core/ShopControl.php(150): OxidEsales\EshopCommunity\Core\ShopControl->_process(
↪'OxidEsales\\Esho...', NULL, NULL, NULL)
[10 Oct 16:44:44.625024 2017] [exception] [stacktrace] #4 /var/www/oxideshop/source/
↪Core/Oxid.php(42): OxidEsales\EshopCommunity\Core\ShopControl->start()
[10 Oct 16:44:44.625024 2017] [exception] [stacktrace] #5 /var/www/oxideshop/source/
↪index.php(31): OxidEsales\EshopCommunity\Core\Oxid::run()
[10 Oct 16:44:44.625024 2017] [exception] [stacktrace] #6 {main}
```

Generic import and erp

If you rely on one of the following old classes, e.g. in a module, you should take care to use the equivalent classes as described. In OXID eShop 4.10 / 5.3, the code of the Generic Import was duplicated in the OXID eShop and the OXID ERP Interface. With OXID eShop 6, we cleaned up this thing: the code of the Generic Import is now only in the OXID eShop.

Changed

The files from `core/objects` are now in the directory `Core/GenericImport/ImportObject`. For some of them the inheritance chain changed (we describe here only the changes on class level):

- the main `base` class changed from `oxERPType` to `ImportObject`, which is now abstract
- `oxERPType_Accessoire` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\Accessories`
- `oxERPType_Artextends` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\ArticleExte`
- `oxERPType_Article` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\Article`
- `oxERPType_Article2Action` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\Article`
- `oxERPType_Article2Attribute` is no longer available

- `oxERPType_Article2Category` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\Artic`
- `oxERPType_Attribute` is no longer available
- `oxERPType_Category` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\Category`
- `oxERPType_Content` is no longer available
- `oxERPType_Country` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\Country`
- `oxERPType_Crossselling` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\CrossSelli`
- `oxERPType_Order` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\Order`
- `oxERPType_OrderArticle` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\OrderArti`
- `oxERPType_ScalePrice` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\ScalePrice`
- `oxERPType_User` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\User`
- `oxERPType_Vendor` is now `\OxidEsales\Eshop\Core\GenericImport\ImportObject\Vendor`

Removed

In former OXID eShop versions the files oxerpbase.php, oxerpcsv.php and oxerpgenimport.php were there for handling the ERP requests. In the version 6.0 all this functionality is bundled in the class OxidEsalesEshopCoreGenericImport-GenericImport. This class lives in the directory SHOP_ROOTsourceCoreGenericImport.

DynPages

The DynPages are not available for OXID eShop 6 anymore. If you extended it, search for a different solution.

WYSIWYG Pro

This section will be added later.

Update OXID eShop from a 6.x version to a 6.y version

1. For updating your project from any 6.0 version to another 6.x version, please edit the metapackage version requirement in your root `composer.json` file to the desired version, e.g. `^v6.0.0-rc.2`
2. Execute **`composer update`** first without executing scripts or plugins

```
composer update --no-plugins --no-scripts
```

3. Execute composer update and execute installation tasks

```
composer update
```

4. Execute the OXID eShop migrations

```
vendor/bin/oe-eshop-db_migrate migrations:migrate
```

Change dependencies

This is an example if you want to include the libary *monolog/monolog* into your project. The steps are similar for changing existing requirements.

1. **cd** to the directory where your root `composer.json` is located
2. Either edit the `composer.json` directly or use the command

```
composer require --no-update monolog/monolog
```

3. Run composer update two times:

```
composer update --no-plugins --no-scripts
composer update
```

Glossary

Introduction

In this glossary we collect terms typical for the OXID world. We collect them in alphabetical order and always try to describe them as easy and abstract as possible.

Edition

An edition is child of the OXID eShop family. Editions are differentiated mainly by their feature sets. Currently there are the editions Community, Professional, Enterprise and B2B.

Meta Package

A *meta package* defines the kind and the exact version of components of a *OXID Compilation* See the composer.json file of the OXID eShop Community Edition meta package for an example.

OXID Compilation

The OXID eShop compilation consists of a certain edition of OXID eShop, which is bundled with the following modules/themes:

- Flow theme
- Paymorrow Module
- PayPal Module
- PayOne Module
- Summernote WYSIWYG Editor

The components of a OXID Compilation are defined in a *Meta Package* To ensure the best stability and interoperability, in a compilation, the versions of all components are pinned to a specific patch release.

Vendor ID

For module developers it is necessary to use unique names for namespaces or classes in their OXID eShop extensions. One way to achieve this is using an unique ID for your company, which you can register by making a pull request to here. This is ID called a *Vendor ID*. More information regarding the *Vendor ID* can be found on https://oxidforge.org/en/extension-acronyms

Conventions for writing developer documentation

Sections

- Each page MUST have one page title as the only first level heading, separated by ===. Otherwise last one would be as document name in Sphinx menu.
- Subsequent headers should be marked with ---, ^^^, """, ~~~ etc.

Good examples:

```
Title
=====

First level
-----------

Second level
^^^^^^^^^^^^

First level
-----------

Second level
^^^^^^^^^^^^

Third level
"""""""""""

Forth level
~~~~~~~~~~~
```

Bad examples:

- Inconsistent headers:

```
First level
-----------

Second level
"""""""""""""

Third level
^^^^^^^^^^^
```

- Two titles in a page:

```
Title
=====
```

```
First level
-----------

Title
=====

First level
-----------
```

External links

To be done...

Use Ref or Doc for links

Use *Ref* or *Doc* to create a link to the page of current developer documentation project.

Using Doc

Use *Doc* when need to link to another file in same catalog.

Example:

- **Code:**

```
:doc:`Modules <modules/index>`
```

- **Rendered result:** *Modules*

Using Ref

Use Ref when need to link to specific file part. References in Sphinx are global, so use unique section name per document and time to form reference. Ref anchor schema: `section_name_with_underscores-YYYYMMDD`

Good examples:

- **Code for Anchor inside page:**

```
.. _conventions_for_using_ref-20160419:

Using Ref
---------
```

- **Code for link which can be in same or other page:**

```
:ref:`Using Ref <conventions_for_using_ref-20160419>`
```

- **Rendered link result** *Using Ref*

Bad examples:

Prefixed with directory name:

```
.. _common_agreements-general-conventions_for_development_wiki_rst_document-20160120:
```

Not suffixed with date:

```
.. _conventions_for_development_wiki_rst_document:
```

Tables

```
+------------------+--------------------+
| Column 1 Heading | Column 2 Heading   |
+==================+====================+
| Column 1 Cell 1  | Column 2 Cell1     |
+------------------+--------------------+
| Column 1 Cell 2  | Column 2 Cell 2    |
+------------------+--------------------+
```

results in

Column 1 Heading	Column 2 Heading
Column 1 Cell 1	Column 2 Cell1
Column 1 Cell 2	Column 2 Cell 2

Code

See http://docutils.sourceforge.net/docs/ref/rst/directives.html#code. Be sure to indent the code with spaces.

Example:

```
.. code:: php

  namespace \OxidEsales\Eshop\Community;

  class Example {}
```

results in

```php
namespace \OxidEsales\Eshop\Community;

class Example {}
```

Highlight Text

Inline markup for menu navigation

```
:menuselection:`Artikel verwalten -->  Artikel`
```

results in: *Artikel verwalten → Artikel*

Inline markup for file names

```
:file:`/usr/lib/python2.{x}/site-packages`
```

results in: /usr/lib/python2.*x*/site-packages

Inline markup for controls

```
:guilabel:`Cancel`
```

results in: *Cancel*

Inline markup for code

```
``exclude_patterns = ['_build', 'Thumbs.db', '.DS_Store']``
```

results in: `exclude_patterns = ['_build', 'Thumbs.db', '.DS_Store']`

Inline markup for commands

```
:command:`cd ..\\GitHub\\Dokumentation-und-Hilfe`
```

results in: **cd ..\GitHub\Dokumentation-und-Hilfe**

Inline markup for downloads

```
:download:`/downloads/varnish/6.0.0/default.vcl`
```

Images

- Do not commit big files or images. Use a link to an external source inside repository. This will help to keep repository small.

```
.. raw:: html

   <p>
      <img width="100" src="https://www.google.co.uk/images/branding/googlelogo/1x/
↪googlelogo_color_272x92dp.png"/>
   </p>
```

UML diagrams

Please do not commit big files or images. Use .svg images and include them like described in the section *Images*. If you created the UML diagram with PlantUml, its good to also add the .puml file to the repository into a separate directory `resources`.

- *Conventions for writing developer documentation*
- Other resources