Magento module overview

> A module is a logical group – that is, a directory containing blocks, controllers, helpers, models – that are related to a specific business feature. In keeping with Magento's commitment to optimal modularity, a module encapsulates one feature and has minimal dependencies on other modules. - Magento DevDocs - Module overview

> Modules and themes are the units of customization in Magento. Modules provide business features, with supporting logic, while themes strongly influence user experience and storefront appearance. Both components have a life cycle that allows them to be installed, deleted, and disabled. From the perspective of both merchants and extension developers, modules are the central unit of Magento organization. - Magento DevDocs - Module overview

> The Magento Framework provides a set of core logic: PHP code, libraries, and the basic functions that are inherited by the modules and other components. - Magento DevDocs - Module overview

Modules can be installed with Composer allowing their version management. All modules installed in such way are placed in the `vendor/` folder and have next base structure `vendor/<vendor>/<type>-<module-name>`. In this case `<type>` can be:

1. `module` - Magento module
2. `theme` - admin or frontend themes
3. `language` - language packs

In a case when you have a very specific functionality or customisation which is related to a specific project and there is no need to share it with other projects it should be created in the `app/code/<vendor>/<type>-<module-name>` directory and the required directories within it.

Module registration

> Magento components, including modules, themes, and language packages, must be registered in the Magento system through the Magento ComponentRegistrar class. - Magento DevDocs - Register your component

> Each component must have a file called registration.php in its root directory. For example, here is the registration.php file for Magento's AdminNotification module. Depending on the type of component, registration is performed through registration.php by adding to it as follows: - Magento DevDocs - Register your component

- How to register modules - registration.php [1]

```
<?php

use \Magento\Framework\Component\ComponentRegistrar;

ComponentRegistrar::register(ComponentRegistrar::MODULE,
'Magento_AdminNotification', __DIR__);
```

- how to register language packages - registration.php [2]

```
<?php

use \Magento\Framework\Component\ComponentRegistrar;

 ComponentRegistrar::register(ComponentRegistrar::LANGUAGE,
'<VendorName>_<packageName>', __DIR__);
```

- how to register themes - registration.php [3]

```
<?php

use \Magento\Framework\Component\ComponentRegistrar;

 ComponentRegistrar::register(ComponentRegistrar::THEME, '<area>/<vendor>/<theme
name>', __DIR__);
```

- composer.json autoload/files[] = "registration.php" [4]

```
 {
"name": "Acme-vendor/bar-component",
"autoload": {
    "psr-4": { "AcmeVendor\\BarComponent\\": ""
},
    "files": [ "registration.php" ]
} }
```

- at what stage - when including vendor/autoload.php [5]

```
  <?php
/**
 * Copyright © Magento, Inc. All rights reserved.
 * See COPYING.txt for license details.
 */

\Magento\Framework\Component\ComponentRegistrar::register(
    \Magento\Framework\Component\ComponentRegistrar::MODULE,
    'Magento_Backend',
    __DIR__
);
```

- how registered when not in composer - project composer.json autoload/files[] = app/etc/NonComposerComponentRegistration.php [6]

```
    "autoload": {
        "files": [

"app/etc/NonComposerComponentRegistration.php"
        ],
    }
```

Examples [1],[2],[3],[4],[5],[6] taken from Magento DevDocs - Register your component

registration.php search paths

```
app/code/*/*/cli_commands.php, registration.php
app/design/*/*/*/registration.php
app/i18n/*/*/registration.php
lib/internal/*/*/registration.php
lib/internal/*/*/*/registration.php
```

registration.php load flow

- pub/index.php
- app/bootstrap.php
- app/autoload.php
- vendor/autoload.php
- vendor/module[]/registration.php – last step in Composer init
- `Magento\Framework\Component\ComponentRegistrar::register(type='module', name='Prince_Productattach', path=`

Describe module limitations.

Minimization of software dependencies is a cornerstone of Magento architecture. Based on this principle there are several logical limitations:

1. One module is responsible only for one feature.
2. Module dependencies on other modules must be declared explicitly.
3. Excluding or disabling a module should not disabling another module.

How do different modules interact with each other?

Magento 2 is PSR-4 compliant. As a main principle of module interaction Magento 2 declares a dependency injection pattern and service contracts.
One of the important parts of module interaction is a Magento area. All components operate with the system and other components in the scope of default areas.

Module dependencies

Modules can contain dependencies upon these software components:

1. Other Magento modules.

2. PHP extensions.
3. Third party libraries.

Module dependencies can be managed by:

> 1. Name and declare the module in the module.xml file.
> 2. Declare any dependencies that the module has (whether on other modules or on a different component) in the module's composer.json file.
> 3. (Optional) Define the desired load order of config files and .css files in the module.xml file.
>
> – Magento DevDocs - Module dependencies

> Example: Module A declares a dependency upon Module B. Thus, in Module A's module.xml file, Module B is listed in the list, so that B's files are loaded before A's. Additionally, you must declare a dependency upon Module B in A's composer.json file. Furthermore, in the deployment configuration, Modules A and B must both be defined as enabled. - Magento DevDocs - Module dependencies

Dependency types

Hard dependency

> Modules with a hard dependency on another module cannot function without the module it depends on. Specifically:
>
> 1. The module contains code that directly uses logic from another module (for example, the latter module's instances, class constants, static methods, public class properties, interfaces, and traits).
> 2. The module contains strings that include class names, method names, class constants, class properties, interfaces, and traits from another module.
> 3. The module deserializes an object declared in another module.
> 4. The module uses or modifies the database tables used by another module.
>
> – Magento DevDocs - Module dependency types

Soft dependency

> Modules with a soft dependency on another module can function properly without the other module, even if it has a dependency upon it. Specifically: The module directly checks another module's availability. The module extends another module's configuration. The module extends another module's layout.
>
> – Magento DevDocs - Module dependency types

Magento module install order flow:

> 1. The module serving as a dependency for another module
> 2. The module dependent on it

Following dependencies should not be created:

> 1. Circular (both direct and indirect)
> 2. Undeclared
> 3. Incorrect
>
> – Magento DevDocs - Module dependency types

> You can build dependencies between classes in the application layer, but these classes must belong to the same module. Dependencies between the modules of the application layer should be built only by the service contract or the service provider interface (SPI). - Magento DevDocs - Module dependency types

Magento areas

A Magento area organizes code for optimized request processing by loading components parts which are related only to the specific area. Areas are registered in the `di.xml` file.

> Modules define which resources are visible and accessible in an area, as well as an area's behavior. The same module can influence several areas. For instance, the RMA module is represented partly in the adminhtml area and partly in the frontend area. If your extension works in several different areas, ensure it has separate behavior and view components for

> each area. Each area declares itself within a module. All resources specific for an area are located within the same module as well. You can enable or disable an area within a module. If this module is enabled, it injects an area's routers into the general application's routing process. If this module is disabled, Magento will not load an area's routers and, as a result, an area's resources and specific functionality are not available.
>
> – Magento DevDocs - Modules and areas

Magento has 5 areas types:

1. Magento Admin (`adminhtml`): entry point for this area is index.php or pub/index.php. The Admin panel area includes the code needed for store management. The /app/design/adminhtml directory contains all the code for components you'll see while working in the Admin panel.
2. Storefront (`frontend`): entry point for this area is index.php or pub/index.php. The storefront (or frontend) contains template and layout files that define the appearance of your storefront.
3. Basic (`base`): used as a fallback for files absent in adminhtml and frontend areas.
4. Cron (`crontab`): In cron.php, the `\Magento\Framework\App\Cron` class always loads the 'crontab' area.

> You can also send requests to Magento using the SOAP and REST APIs. These two areas:
>
> 1. Web API REST (`webapi_rest`): entry point for this area is index.php or pub/index.php. The REST area has a front controller that understands how to do URL lookups for REST-based URLs.
> 2. Web API SOAP (`webapi_soap`): entry point for this area is index.php or pub/index.php.
>
> – Magento DevDocs - Modules and areas

Not documented but used in code Magento/Framework/App/Area.php:

1. Documentation (doc). Deprecated.
2. Admin (admin). Deprecated.

What side effects can come from this interaction?

- error when module is missing or disabled
- error when injecting missing class
- (?) null or error when using object manager for missing class ReflectionException - Class MissingClass does not exist objectManager->create() = new $type() or new $type(…args) --> PHP Warning: Uncaught Error: Class 'MissingClass' not found