Design Patterns In Magento

Model View Controller Pattern (MVC)

Design pattern where business, presen tation and coupling logic are separated

Front Controller Pattern

Makes sure that there is one and only one point of entry. All requests are investigated, routed to the designated controller and then processed accord ingly to the specification

Factory Pattern

Responsible of factorizing (instantiating) classes

Singleton Pattern

Checks the whether this class has already been instantiated before, this results in a shared instance.

Registry Pattern

Internal registry: a global scoped container for storing data

Prototype Pattern

It defines that instances of classes can retrieve a specific other class instance depending on its parent class (the prototype)

Object Pool Pattern

A box with objects so that they do not have to be allocated and destroyed over and over again

Iterator Pattern

The iterator pattern defines that there is a shared way to iterate over a container with objects

Lazy Loading Pattern

Lazy loading ensures that loading data is delayed until the point when it is actually needed

Service Locator Pattern

Design Patterns In Magento (cont)

The service locator pattern abstracts away the retrieval of a certain service. This allows for changing the service without breaking anything

Module Pattern

An implementation of the module pattern would make sure that each element can be removed or swapped

Observer Pattern

By defining observers (or listeners), extra code can be hooked which will be called upon as the observed event fires

Proxy Pattern

- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.

MODULE.XML File

A component declares itself (that is, defines its name and existence) in the **module.xml** file, located in the Magento install directory at **<ComponentName>/etc/**

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:Module/etc/module.xsd">
<module name="Vendor_ComponentName" setup_version="2.0.0"/>
<sequence><module name="Magento_Sales"/></sequence>
</config>
```

CONFIG.XML File

Used to configure module. The responsibilities of the **config.xml** configuration file used in earlier versions of Magento is now divided between several files, located in various module directories.

Example

```
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Store:etc/config.xsd">
<default>
<customsetting><general><enable>1</enable><title>Module Title</title></general>
</customsetting></default></config>
```

DI.XML File

What case we use di.xml ?	ϟ We can use di.xml for (rewrite) preference of a particular class.
	ϟ We can send a new or replace the existing class arguments.
	ϟ Use plugins to do some stuff before, after and around a function
	ϟ By using virtualTypes creating a sub-class of another class.

Example for Preference

```
<preference for="Magento\Customer\Api
\AddressRe positoryInterface" type="Mage
nto\Customer\Model\Resour ceModel\Addres
sRepository" />
```

Above code, When someone asks to instantiate a Magento\Customer\Api\AddressRepositoryInterface it will instantiate a Magento\Customer\Model\Resourc eModel\Addre ssRepository object (the type attribute).

Example for Arguments

```
<type name="Magento\Customer\Model\Resour
ceModel\Group" shared="false"> <argum
ents> <argument name="groupManagement"
xsi:type="object">Magento\Customer\ Api
\GroupManagementInterface\Proxy< /argume
nt> </arguments> </type>
```

In the above code, We are sending object as an argument, we are saying system to insert **"Proxy"** class as an object with the name of groupMana gement. Also we can use Arguments for replacing the existing argument too.

Example for Plugin

```
<type name="Magento\Customer\Model\Resour
ceModel\Visitor"> <plugin name="catalog
Log" type="Magento\Catalog\Model\Plugin\
Log" /> </type>
```

In the above code , public function clean($object) in visitor class is called after public function afterClean(V isitor $subject, $logResourceModel) which is in Log class.

Example for Virtual Types

```
<virtualType name="ourVirtualTypeName"
type="Pulsestorm\TutorialVirtualType
\Model\Argu ment1"> </virtualType>
```

DI.XML File (cont)

Creating a virtual type is sort of like creating a sub-class for an existing class. With virtual types, the only behavior you can change in your virtual sub-class is which dependencies are injected.

EVENTS.XML File

Custom events can be dispatched by simply passing in a unique event name to the event manager when you call the dispatch function. The unique event name is referenced in your module's **events.xml** file where you specify which observers will react to that event.

The observer xml element has the following properties:

name (required)	The name of the observer for the event definition
instance (required)	The fully qualified class name of the observer
disabled	Determines whether this observer is active or not. Default value is false
shared	Etermines the lifestyle of the class. Default is false

Example

```
<event name="my_module_event_before">
<observer name="myObserverName" instance="MyCompany\MyModule\Observ er
\MyObserver" />
</event>
```

ACL.XML File

file is where we define our module access control list resources. Access control list resources are visible under the Magento admin System | Permissions | User Roles area, when we click on the Add New Role button.

Example

```
<?xml version="1.0"?>
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:framework:Acl/etc/acl.xsd">
<acl><resources><resource id="Magento_Backend::admin">
<resource id="Maxime_Jobs::job_head" title="Jobs" sortOrder="100" >
<resource id="Maxime_Jobs::department" title="Departments"
sortOrder="10"><resource id="Maxime_Jobs::department_save"
title="Save Department" sortOrder="10" /><resource
id="Maxime_Jobs::department_delete" title="Delete Department"
sortOrder="20" /></resource></resource>
</resource></resources></acl>
</config>
```

SYSTEM.XML File

The **system.xml** is a configuration file which is used to create configuration fields in Magento 2 System Configuration. You will need this if your module has some settings which the admin needs to set

MENU.XML File

A **menu.xml** file is a collection of <add/> nodes. Each of these nodes adds a Menu Item to Magento's backend

Example

```
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Backend:etc/menu.xsd">
<menu><add id="Example_MenuTutorial::my_menu_example" title="Title Example"
module="Example_MenuTutorial" sortOrder="9999"
resource="Magento_Backend::content" /></menu>
</config>
```

id	Attribute defines a unique identifier for this node. By convention, (but not required) this should the the name of the module
title	Attribute controls the text an end users sees for the Menu Item
module	Attribute should match the current module
sortOrder	Attribute controls how this Menu Item is sorted with the other Menu Items in the system
resource	attribute defines the ACL rule a user must have in order to access this Menu Item

WIDGET.XML File

The file that is responsible for widgets in Magento is placed in **[module_dir]/etc/widget.xml**

WEBAPI.XML File

The **webapi.xml** is used to configure access rights and API Interface which specify method that will be used.

Example

```
<?xml version="1.0"?>
<routes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="urn:magento:module:Magento_Webapi:etc/webapi.xsd">
<route url="/V1/hello/name/:name" method="GET">
<service class="Inchoo\Hello\Api\HelloInterface" method="name"/><resources>
<resource ref="anonymous"/>
</resources></route></routes>
```

WEBAPI.XML File (cont)

Resource tag defines what resources user needs to have to be able to access this api call. Possible options are self, anonymous o Magento resource like **Magento_Catalog::products** or **Magento_Customer::group**

Magento 2 Modes

Command line

Cache Management	Commands
Clear all caches:	php magento cache:clean
Clearing a single cache:	php magento cache:clean config
Flush all caches:	php magento cache:flush
Flushing a single cache:	php magento cache:flush config
Index Management	**Commands**
Runs all available indexes:	php magento indexer:reindex
Cron	**Commands**
Runs all cron jobs by schedule:	php magento cron:run
Help & Knowledge	**Commands**
Display help/infos about a command	php magento help indexer:reindex
Lists all available CLI commands	php magento list

>_ Command line (cont)

Developement & Production	Commands
Set to the production mode to harden your setup	php magento deploy:mode:setproduction
(exceptions are not displayed to the user, only logged to log files)	
Developer mode is less secure than Production mode	php magento deploy:mode:setdeveloper
provides verbose logging (errors displayed in browser) enhanced debuggingand disables static view file caching	

Recompile Store	Commands
Recompile the Magento 2 codebase	php magento setup:di:compile

♻ Useful Code Snippets and Tips

How to get product collection in custom template block

```
$objectManager =
\Magento\Framework\App\ObjectManager::getInstance();
/** @var
\Magento\Catalog\Model\ResourceModel\Product\Collectio
n $productCollection */
$productCollection =
$objectManager->create('Magento\Catalog\Model\Resourc
eModel\Product\Collection');
/** Apply filters here */
$productCollection->load();
```

How to use Mage::log method in Magento 2

```
protected $logger;
public function __construct(\Psr\Log\LoggerInterface
$logger) {$this->logger = $logger; }
```

You use debug, exception, system for psr logger for example

```
$this->logger->info($message);
$this->logger->debug($message);
```

How to retrieve product information in Magento 2

In Magento 2 proposed to use service layer for this. Try use `\Magento\Catalog\Model\ProductRepository::getById` method to get product by id

How to get a store in Magento 2 programmatically?

Magento 1.x

♻ Useful Code Snippets and Tips (cont)

```
Mage::app()->getStore()
```

```
function rm_store($id = null) {
/** @var \Magento\Framework\ObjectManagerInterface $om
*/
$om =
\Magento\Framework\App\ObjectManager::getInstance();
/** @var \Magento\Store\Model\StoreManagerInterface
$manager */
$manager = $om-
>get('Magento\Store\Model\StoreManagerInterface');
return $manager->getStore($id); }
```

How to make JavaScript strings translatable (localizable)

To make JavaScript strings translatable or localizable in Magento 2, use : `$.mage.__`

Example : $.mage.__('Select type of option.')

Some Magento 2 core modules use another way to translate JavaScript strings: they define/require the mage/translate library

How to call theme images from a static block

You call theme images from static block as follows: **{{view url="images/demo.jpg"}}**

How to instantiate a model in Magento 2

Magento strictly discourages the use of ObjectManager directly. It provides service classes that abstract it away for all scenarios.
For all non-injectables (models) you have to use factory:

```
protected $pageFactory;
public function
__construct(\Magento\Cms\Model\PageFactory
$pageFactory) {
$this->pageFactory = $pageFactory;
}
public function someFunc() {
... $page = $this->pageFactory->create(); ...
}
```

How to get the collection of custom Magento 2 modules

```
protected $mymodulemodelFactory;
public function __construct( .... \[Namespace]\
[Module]\Model\ResourceModel\
[Entity]\CollectionFactory $mymodulemodelFactory, ...
) {
... $this->mymodulemodelFactory =
$mymodulemodelFactory; ...
}
```

♻ Useful Code Snippets and Tips (cont)

and you can use in any one of the class methods:

```
$collection = $this->mymodulemodelFactory->create();
```

How to get POST and GET requests in Magento 2

In a case of a controller that extends **Magento\Framework\App\Action\Action**, it is possible to get the request with the aid of `$this->getRequest()->getPost()`.

For a custom class, inject the request in the constructor:

```
namespace Namespace\Module\Something;
class ClassName {
protected $request;
public function __construct(
\Magento\Framework\App\Request\Http $request,
....//rest of parameters here ) {
$this->request = $request;
...//rest of constructor here
}
public function getPost() {
return $this->request->getPost();
}
}
```