Demonstrate ability to use plugins

There are three types of plugins in magento: around, before and after.

Important: Classes, abstract classes and interfaces that are implementations of or inherit from classes that have plugins will also inherit plugins from the parent class. For example, if you create a plugin for \Magento\Catalog\Block\Product\AbstractProduct, plugin methods will be called for all child classes, such as \Magento\Catalog\Block\Product\View, \Magento\Catalog\Block\Product\ProductList\Upsell etc.

Limitations

Plugins only work on **public methods** for any **classes** or **interfaces**. So plugins can not be used on following:

1. Final methods
2. Final classes
3. Non-public methods
4. Class methods (such as static methods)
5. __construct
6. Virtual types
7. Objects that are instantiated before Magento\Framework\Interception is bootstrapped

@since 2.2 *after* plugin has access to *arguments*!

Prioritizing plugins

The `sortOrder` property controls how your plugin interacts with other plugins on the same class.

> The prioritization rules for ordering plugins:
>
> - Before the execution of the observed method, Magento will execute plugins from lowest to greatest `sortOrder`.
>
> - During each plugin execution, Magento executes the current plugin's before method.
> - After the before plugin completes execution, the current plugin's around method will wrap and execute the next plugin or observed method.
> - Following the execution of the observed method, Magento will execute plugins from greatest to lowest `sortOrder`.
>
> - During each plugin execution, the current plugin will first finish executing its around method.
> - When the around method completes, the plugin executes its after method before moving on to the next plugin.
>
> – Magento DevDocs - Plugins

Plugin sortOrder:

- before sortOrder=10, before sortOrder=20, before sortOrder=30 …
- before and around (first half) called together for same plugin!
- around (second half) and after called together for same plugin!

Example:

- pluginA.beforeMethod, pluginA.aroundMethod first half
- pluginB.beforeMethod, pluginB.aroundMethod first half
- pluginC.beforeMethod, `_____
- _____, pluginD.aroundMethod first half
- method()
- _____, pluginD.aroundMethod second half
- pluginC.afterMethod , _____
- pluginB.aroundMethod second half, pluginB.afterMethod
- pluginA.aroundMethod second half, pluginA.afterMethod

Identify strengths and weaknesses of plugins.

> The greatest weakness is exploited in the hands of a developer who is either not experienced or not willing to take the time to evaluate the fallout. For example, used improperly, an around plugin can prevent the system from functioning. They can also make understanding what is going on by reading source code hard (spooky action at a distance).
>
> – Swiftotter Developer Study Guide

Using around plugins:

> Avoid using around method plugins when they are not required because they increase stack traces and affect performance. The only use case for around method plugins is when you need to terminate the execution of all further plugins and original methods.
>
> – Magento DevDocs - Programming Best Practices

In which cases should plugins be avoided?

> Plugins are useful to modify the input, output, or execution of an existing method. Plugins are also best to be avoided in situations where an event observer will work. Events work well when the flow of data does not have to be modified.
>
> – Swiftotter Developer Study Guide

Info from Magento technical guidelines:

> `4. Interception`
>
> 4.1. Around-plugins SHOULD only be used when behavior of an original method is supposed to be substituted in certain scenarios.
>
> 4.2. Plugins SHOULD NOT be used within own module .
>
> 4.3. Plugins SHOULD NOT be added to data objects.
>
> 4.4. Plugins MUST be stateless.
>
> …
>
> `14. Events`

> 14.1. All values (including objects) passed to an event MUST NOT be modified in the event observer. Instead, plugins SHOULD BE used for modifying the input or output of a function.
>
> – Magento DevDocs - Technical guidelines

Also check:

- Yireo - Magent 2 observer or Plugin?
- Magento DevDocs - Observers Best Practices

how it works?

Magento automatically generates `Interceptor` class for the plugin target and store it in the `generated\code` directory.

```
    namespace \Notes;

    class Interceptor extends \Notes\MyBeautifulClass implements
\Magento\Framework\Interception\InterceptorInterface
    {
        use \Magento\Framework\Interception\Interceptor;

        public function __construct($specificArguments1, $someArg2 = null)
        {
            $this->___init();
            parent::__construct($specificArguments1, $someArg2);
        }

        public function sayHello()
        {
            pluginInfo = pluginList->getNext('MyBeautifulClass', 'sayHello')
            ___callPlugins('sayHello', [args], pluginInfo)
        }
    }
```

\Magento\Framework\Interception\Interceptor:

- $pluginList = \Magento\Framework\Interception\PluginListInterface
- $subjectType = 'MyBeautifulClass'
- `___init` - called in in constructor, pluginList = get from object manager, subjectType = class name
- pluginList->getNext
- `___callPlugins`
- `___callParent`

how generated?

```
\Magento\Framework\App\Bootstrap::create
\Magento\Framework\App\Bootstrap::__construct
\Magento\Framework\App\ObjectManagerFactory::create
\Magento\Framework\ObjectManager\DefinitionFactory::createClassDefinition
    \Magento\Framework\ObjectManager\DefinitionFactory::getCodeGenerator
    \Magento\Framework\Code\Generator\Io::__construct
    \Magento\Framework\Code\Generator::__construct
    spl_autoload_register([new \Magento\Framework\Code\Generator\Autoloader,
'load']);

\Magento\Framework\App\ObjectManagerFactory::create
\Magento\Framework\Code\Generator::setGeneratedEntities
\Magento\Framework\App\ObjectManager\Environment\Developer::configureObjectManager

\Magento\Framework\Code\Generator\Autoloader::load
\Magento\Framework\Code\Generator::generateClass
```

Decide how to generate based on file suffix - generator \Magento\Framework\Code\Generator\EntityAbstract

```
array (
  'extensionInterfaceFactory' =>
'\\Magento\\Framework\\Api\\Code\\Generator\\ExtensionAttributesInterfaceFactoryGenerator',
  'factory' => '\\Magento\\Framework\\ObjectManager\\Code\\Generator\\Factory',
  'proxy' => '\\Magento\\Framework\\ObjectManager\\Code\\Generator\\Proxy',
  'interceptor' => '\\Magento\\Framework\\Interception\\Code\\Generator\\Interceptor',
  'logger' => '\\Magento\\Framework\\ObjectManager\\Profiler\\Code\\Generator\\Logger',
    - logs all public methods call
    - Magento\Framework\ObjectManager\Factory\Log -- missing?
  'mapper' => '\\Magento\\Framework\\Api\\Code\\Generator\\Mapper',
    - extractDto() = $this->{$name}Builder->populateWithArray()->create
  'persistor' => '\\Magento\\Framework\\ObjectManager\\Code\\Generator\\Persistor',
    - getConnection, loadEntity, registerDelete, registerNew, registerFromArray, doPersist,
doPersistEntity
  'repository' => '\\Magento\\Framework\\ObjectManager\\Code\\Generator\\Repository', --
deprecated
  'convertor' => '\\Magento\\Framework\\ObjectManager\\Code\\Generator\\Converter',
    - Extract data object from model
    - getModel(AbstractExtensibleObject $dataObject) = getProductFactory()->create()-
>setData($dataObject)->__toArray()
  'searchResults' => '\\Magento\\Framework\\Api\\Code\\Generator\\SearchResults',
    - extends \Magento\Framework\Api\SearchResults
  'extensionInterface' =>
'\\Magento\\Framework\\Api\\Code\\Generator\\ExtensionAttributesInterfaceGenerator',
  'extension' =>
'\\Magento\\Framework\\Api\\Code\\Generator\\ExtensionAttributesGenerator',
    - extension_attributes.xml
    - extends \Magento\Framework\Api\AbstractSimpleObject
    - implements {name}\ExtensionInterface
    - for every custom attribute, getters and setters
  'remote' =>
'\\Magento\\Framework\\MessageQueue\\Code\\Generator\\RemoteServiceGenerator',
)
```

```
Magento\Framework\App\ResourceConnection\Proxy -> type Proxy, name
Magento\Framework\App\ResourceConnection
Magento\Framework\Code\Generator::shouldSkipGeneration - type not detected, or
class exists
\Magento\Framework\Code\Generator::createGeneratorInstance -- new for every file
```

\Magento\Framework\Code\Generator\EntityAbstract - code generation

```
\Magento\Framework\Code\Generator\EntityAbstract::generate
\Magento\Framework\Code\Generator\EntityAbstract::_validateData - class not
existing etc.
\Magento\Framework\Code\Generator\EntityAbstract::_generateCode
  - \Magento\Framework\Code\Generator\ClassGenerator extends
\Zend\Code\Generator\ClassGenerator
-
\Magento\Framework\Code\Generator\EntityAbstract::_getDefaultConstructorDefinition
- \Magento\Framework\Code\Generator\EntityAbstract::_getClassProperties
- \Magento\Framework\Code\Generator\EntityAbstract::_getClassMethods
```

Links

- Magento DevDocs - Plugins (Interceptors)
- Alan Storm - Magento 2 Object Manager Plugin System