Backward Compatibility
Why Backward Compatibility?
Where we are...

- 2.0.0 – November 16, 2015
- 2.0.1 – January 19, 2016
- ..... 
- 2.0.8 – July 18, 2016
- 2.0.9 – August 3, 2016
- 2.0.10 – October 7, 2016
- 2.0.11 – December 13, 2016
- 2.0.12 – January 27, 2017
- 2.0.13 – February 16, 2017

- 2.1.0 – June 23, 2016
- 2.1.1 – August 25, 2016
- 2.1.2 – October 10, 2016
- 2.1.3 – December 8, 2016
- 2.1.4 – January 23, 2017
- 2.1.5 – February 15, 2017
- 2.1.6 – March 27, 2017
- ..... 
- 2.2.0 - TBD

20 patch releases, 1 minor release
Why does Backward Compatibility matter?

For merchants

cost-effective process

For developers –

forward-compatible extensions
Does Magento have a lot of bugs?
Are these bugs annoying for Magento developers?
Keep Magento backwards compatible vs. fixing its flaws?
We MUST do BOTH
Backward Compatible Fix

*it works (most of the time), but code quality is far from good enough
Backward compatibility (BC) policy for Magento code
Semantic Versioning

MAJOR . MINOR . PATCH

Product Version

Module Version

MAJOR . MINOR . PATCH

MAJOR . 100 . 1

MINOR . 1 . 4

PATCH . 4 . 4

MAJOR . 2 . 1 . 4
Public vs Private code

**Public** – has `@api` annotation

**Private** – no `@api` annotation
What examples of Public code Magento has?

- Class/Interfaces (PHP, Javascript)
- XML configuration
- Events
- ...

Full list is on the DevDocs: Versioning and compatibility

API vs SPI (Extension Points)

A PHP Interface usages:

• As an API

• As a Service Provider Interface (SPI)

• As both
API vs SPI (Extension Points)

Who decides whether interface/class belong to API or SPI?

YOU
Dependency Rules
## Dependency Rules

### Rule #1

<table>
<thead>
<tr>
<th>Usage</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calls API</td>
<td>MAJOR</td>
</tr>
</tbody>
</table>

```json
"require": {
  "magento/module-customer": "100.*.*"
}
```
**Dependency Rules**

Rule #2

<table>
<thead>
<tr>
<th>Usage</th>
<th>Dependency</th>
</tr>
</thead>
</table>

**Uses SPI**

**MAJOR+MINOR**

```
"require": {
    "magento/module-customer": "100.5.*"
}
```
## Dependency Rules

### Rule #3

<table>
<thead>
<tr>
<th>Usage</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses private code</td>
<td>MAJOR+MINOR+PATCH</td>
</tr>
</tbody>
</table>

```
"require": {
  "magento/module-customer": "100.5.2"
}
```
Prohibited Code Changes
Prohibited Code Changes. PHP

1. Interface/class removal

2. Public/protected method removal

3. Introduction of a method to a class/interface
Prohibited Code Changes. PHP

4. Parameter addition in public methods

5. Parameter addition in protected methods

6. Method argument type modification
Allowed constructor modifications

class ExistingClass
{
    private $newDependency;

    public function __construct(
        \Old\Dependency\Interface $oldDependency,
        $oldRequiredConstructorParameter,
        $oldOptionalConstructorParameter = null,
        \New\Dependency\Interface $newDependency = null
    ) {

        $this->newDependency = $newDependency ? \Magento\Framework\App\ObjectManager::getInstance()
        ->get(\New\Dependency\Interface::class);

    }

    public function existingFunction()
    {
        // Use $this->newDependency wherever the new dependency is needed
    }
}
Keep backward compatibility in:

- JavaScript
- CSS
- XML
- PHP
- MySQL
The main rule of backwards compatibility

Backwards compatibility is more important than effort of the implementation
Do all backward compatible fixes look ugly?
This is how Backward Compatible fix should look like
