Magento Live

Australia | 2016

Magento 2 Request Flow

Eugene Tulika

Magento 2 Architect

Entry Point

Everything starts here

Entry Point

- Index.php or pub/index.php – web application
- pub/static.php – retrieve dynamically generated static files
- pub/get.php – retrieve media from database
- pub/cron.php – run commands by Cron
- bin/magento – command line tool

Entry Point

/index.php

```
require __DIR__ . '/app/bootstrap.php';
$bootstrap = \Magento\Framework\App\Bootstrap::create(BP, $_SERVER);
$app = $bootstrap->createApplication(\Magento\Framework\App\Http::class);
$bootstrap->run($app);
```

Bootstrap::createApplication

/lib/internal/Magento/Framework/App/Bootstrap.php

```
public function createApplication($type, $arguments = [])
{
    $this->initObjectManager();
    $application = $this->objectManager->create($type, $arguments);
    // … Checks that Application Implements AppInterface.
    return $application;
}
```

Bootstrap::initObjectManager

- Loads initial configuration (`config.php`, `env.php`, `di.xml`)
- Loads `etc/di.xml` files from all modules (global configuration)
- We discourage usage `init*` methods. There was a lot of them in M1 which made it really hard to use classes reliably
- This is the last point I request flow where `init*` can still be used

Bootstrap::createApplication

/lib/internal/Magento/Framework/App/Bootstrap.php

```
public function createApplication($type, $arguments = [])
{
    $this->initObjectManager();
    $application = $this->objectManager->create($type, $arguments);
    // … type check
    return $application;
}
```

Entry Point

/index.php

```
require __DIR__ . '/app/bootstrap.php';
$bootstrap = \Magento\Framework\App\Bootstrap::create(BP, $_SERVER);
$app = $bootstrap->createApplication(\Magento\Framework\App\Http::class);
$bootstrap->run($app);
```

Application\Http

/lib/internal/Magento/Framework/App/Http.php

```
public function launch()
{
    $frontName = $this->_request->getFrontName();
    $areaCode = $this->areaList->getCodeByFrontName($frontName);
    $this->state->setAreaCode($areaCode);
    $this->objectManager->configure($this->_configLoader->load($areaCode));
    $frontController = $this->objectManager->get(FrontControllerInterface::class);
    $result = $frontController->dispatch($this->_request);
    $result->renderResult($this->_response);
    return $this->_response;
}
```

Application Areas

- Configuration scopes
- `etc/<areaCode>/*` in module folder
- Loaded on top of global configuration
- Admin, Frontend, Webapi_Rest, Webapi_Soap, Cron

Middleware

onion rings of application

Middleware

function (request) : response

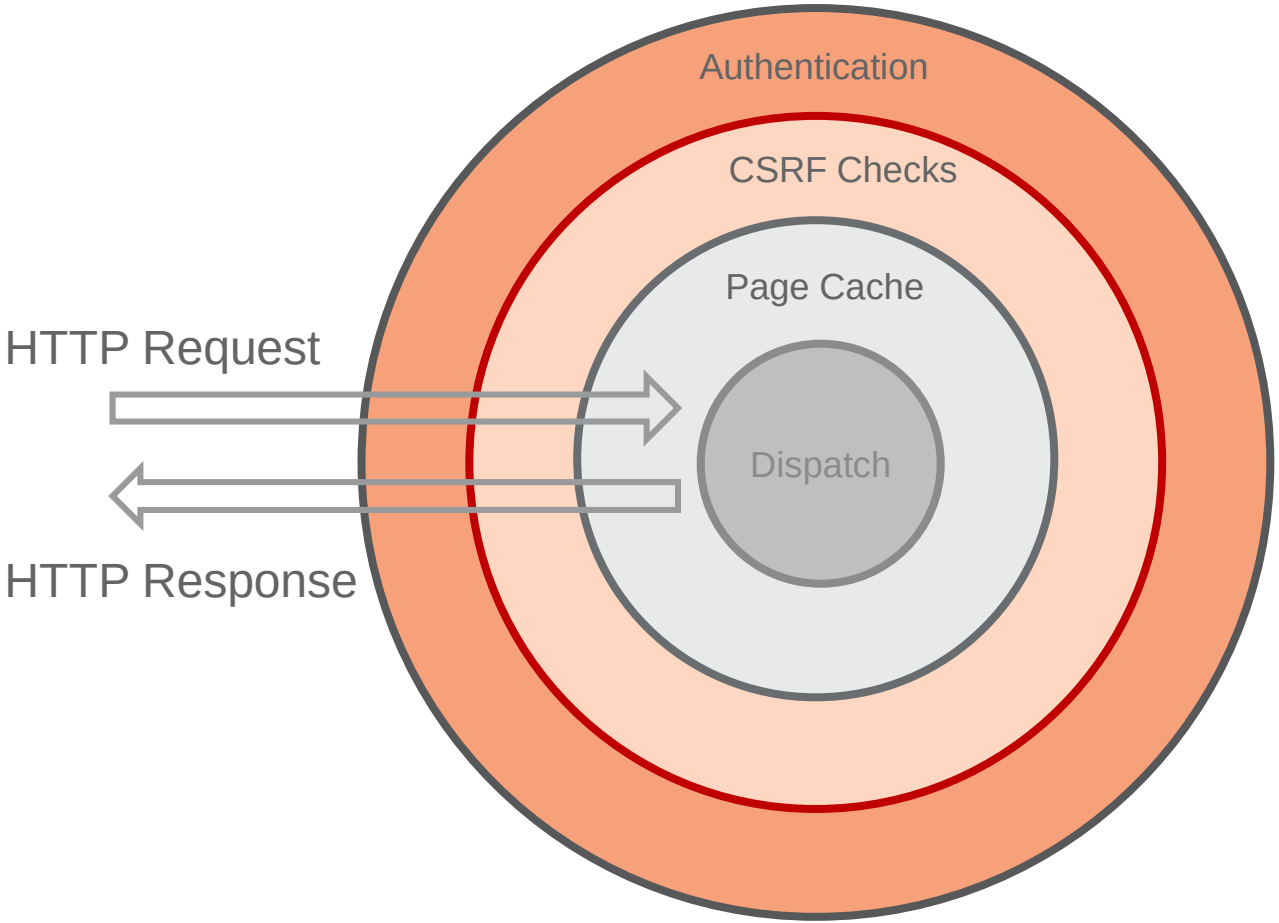
- PSR-7 defines request and response, middleware maps them together
- Function applied to request in order to generate response

Application\Http

/lib/internal/Magento/Framework/App/Http.php

```
public function launch()
{
    $frontName = $this->_request->getFrontName();
    $areaCode = $this->areaList->getCodeByFrontName($frontName);
    $this->state->setAreaCode($areaCode);
    $this->objectManager->configure($this->_configLoader->load($areaCode));
    $frontController = $this->objectManager->get(FrontControllerInterface::class);
    $result = $frontController->dispatch($this->_request);
    $result->renderResult($this->_response);
    return $this->_response;
}
```

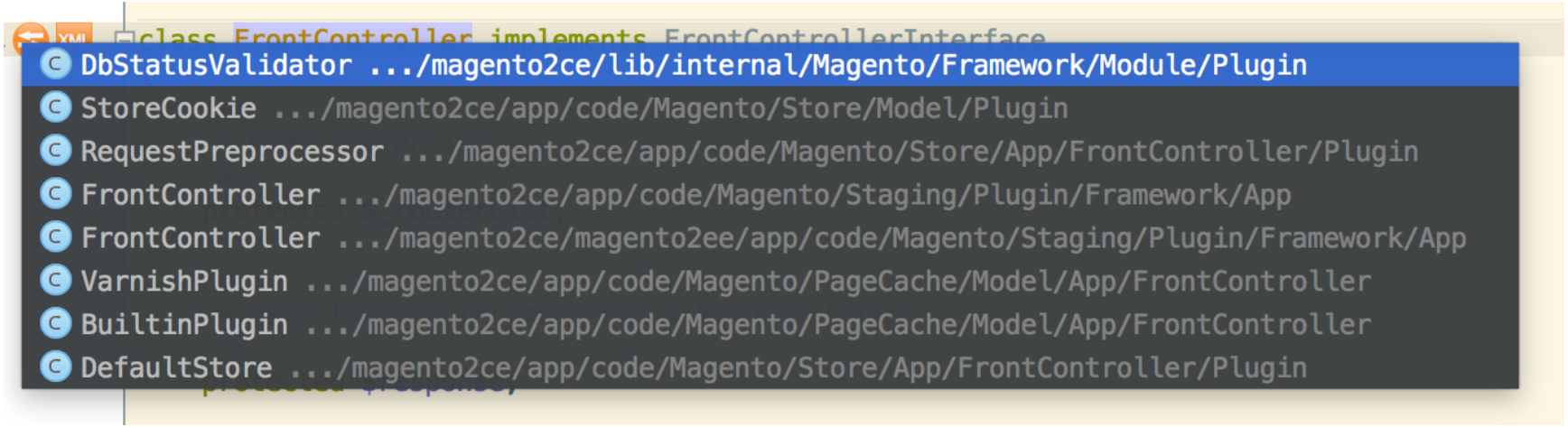
Middleware


Authentication
CSRF Checks
Page Cache
Dispatch
HTTP Request
HTTP Response

Middleware

- If some logic should be executed on all controllers, it should be added as a plugin on Front Controller
- Keep POST & GET in mind

Existing Plugins on Front Controller



Routing

Front Controller

/lib/internal/Magento/Framework/App/FrontController.php

```
public function dispatch(RequestInterface $request){
    // …
    foreach ($this->routerList as $router) {
        $actionInstance = $router->match($request);
        if ($actionInstance) {
            $result = $actionInstance->execute();
            break;
        }
    }
    // …
    return $result;
}
```

Routing

- Every area has own set of routers
- Important routers: standard for frontend & admin for admin
- Both configured in `/etc/<areaCode>/routes.xml`
- Cover 95% of cases

Routing

/My/Module/etc/frontend/routes.xml

```
<config>
    <router id="standard">
        <route id=”blog" frontName=”blog">
            <module name=”My_Module" />
        </route>
    </router>
</config>
```

Actions

Front Controller

/lib/internal/Magento/Framework/App/FrontController.php

```
public function dispatch(RequestInterface $request){
    // …
    foreach ($this->routerList as $router) {
        $actionInstance = $router->match($request);
        if ($actionInstance) {
            $result = $actionInstance->execute();
            break;
        }
    }
    // …
    return $result;
}
```

Action Controllers

- Handles application logic execution
- Has single method `execute` implements ActionInterface
- By convention it is located in `Controller` folder of a module
- Request object should not be passed further. Instead, data should be extracted and passed as arguments.

Action Controllers

My/Module/Controller/Blog/Post/View.php

```
class View extends \Magento\Framework\App\Action\Action
{
    // constructor and properties

    public function execute()
    {
        // do something
        $result = $this->resultFactory->create(ResultFactory::TYPE_PAGE);
        return $result;
    }
}
```

POST

POST Action Controllers

- Handles requests from UI to business logic
- Should not render pages, layout is forbidden for POST
- All business logic should be delegated to Service Contracts
- Single place to manage application workflow:
  - The only point to do Session management, including adding messages
  - Redirect or forward result object should be returned

POST Action Controller

My/Module/Controller/Blog/Post/Save.php

```
class Save extends \Magento\Framework\App\Action\Action
{
    public function execute()
    {
        $post = $this->postFactory->create();
        $post->setTitle($this->request->getParam('title'));
        $post->setText($this->request->getParam('text'));
        $post = $this->postRepository->save($post);
        $result = $this->resultFactory->create(ResultFactory::TYPE_REDIRECT);
        $result->setPath('blogs/post/view', ['post_id' => $post->getId()]);
    }
}
```

Service Contracts

- Represent public API of a module
- Covered by Backwards Compatibility Policy
- Located `Api` folder of a module and marked with `@api`
- Expose procedural API (Service Interfaces) that communicate with Data Transfer Objects (Data Interfaces)

Service Contracts

My/Module/Api/PostRepository.php

```
class PostRepository implements \My\Module\Api\PostRepositoryInterface
{
    // ..
    public function save(PostInterface $post)
    {
        $this->postResourceModel->save($post);
        return $post;
    }
}
```

Service Contracts for Persistence

- Uses Resource Models to Load/Save/Delete data
- Uses Collections to load lists of data
- `load`, `save`, and `delete` on model are deprecated

POST Action Controller

My/Module/Controller/Blog/Post/Save.php

```
class Save extends \Magento\Framework\App\Action\Action
{
    public function execute()
    {
        $post = $this->postFactory->create();
        $post->setTitle($this->request->getParam('title'));
        $post->setText($this->request->getParam('text'));
        $post = $this->postRepository->save($post);
        $result = $this->resultFactory->create(ResultFactory::TYPE_REDIRECT);
        $result->setPath('blogs/post/view', ['post_id' => $post->getId()]);
        return $result;
    }
}
```

GET

GET Action Controller

/app/code/My/Module/Controller/Blog/Post/View.php

```
class View extends \Magento\Framework\App\Action\Action
{
    // constructor and properties

    public function execute()
    {
        $result = $this->resultFactory->create(ResultFactory::TYPE_PAGE);
        return $result;
    }
}
```

GET Action Controllers

- Blocks should not be referenced directly from Controller. Layout should create them.
- Models should not be pre-loaded and put to registry. Instead, Blocks should load them.
- Action Controller which respond to GET requests and render pages should DO NOTHING
- It should only return PageResult
- User-Specific content should not be rendered on the server side

Layout

/My/Module/view/frontend/layout/blog_post_view.xml

```
<page>
    <body>
        <referenceBlock name="container">
            <block class="My\Module\Block\Blog\Post\View”
                   name=”blog.post.view”
                   template=”My_Module::blog/post/view.phtml"/>
        </referenceBlock>
    </body>
</page>
```

Template

/My/Module/view/frontend/templates/blog/post/view.phtml

```
<?php $post = $block->getBlogPost(); ?>
<article>
    <header>
        <h1><?php echo $block->escapeHtml($post->getTitle());?></h1>
        <span class="date">
            <?php echo $block->formatDate($post->getPublicationDate());?>
        </span>
        <a href="<?php echo $block->getBlogLink($post); ?>" class="blog">
            <?php echo $block->escapeHtml(__('To Blog'));?>
        </a>
    </header>
    <?php echo $post->getText(); ?>
</article>
```

Blocks

- View Logic for the GET Action Controllers
- Should not reference other sibling and parent blocks
- Should not perform state-modifications
- Should retrieve data from public Service Contracts and service classes
- Request object should not be passed further. Instead, data should be extracted and passed as arguments.

Blocks

/My/Module/Block/Blog/Post/View.php

```php
namespace My\Module\Block\Blog\Post;
class View
{
    public function getBlogPost()
    {
        return $this->postRepository->get($this->request->getParam('post_id'));
    }

    public function getBlogLink(Post $post)
    {
        $this->urlBuilder->getUrl('blogs', ['blog_id' => $post->getBlogId()]);
    }
}
```

Blocks

/My/Module/view/frontend/templates/blog/post/view.phtml

```
<?php $post = $block->getBlogPost(); ?>
<article>
    <header>
        <h1><?php echo $block->escapeHtml($post->getTitle());?></h1>
        <span class="date">
            <?php echo $block->formatDate($post->getPublicationDate());?>
        </span>
        <a href="<?php echo $block->getBlogLink($post); ?>" class="blog">
            <?php echo $block->escapeHtml(__('To Blog'));?>
        </a>
    </header>
    <?php echo $post->getText(); ?>
</article>
```

Response

Action Results

/lib/internal/Magento/Framework/Controller/ResultFactory.php

```
namespace Magento\Framework\Controller;
class ResultFactory
{
    const TYPE_JSON     = 'json';
    const TYPE_RAW      = 'raw';
    const TYPE_REDIRECT = 'redirect';
    const TYPE_FORWARD  = 'forward';
    const TYPE_LAYOUT   = 'layout';
    const TYPE_PAGE     = 'page';

    public function create($type, array $arguments = [])
    {
        // ...
        return $resultInstance;
    }
}
```

GET Action Controller

/app/code/My/Module/Controller/Blog/Post/View.php

```
class View extends \Magento\Framework\App\Action\Action
{
    // constructor and properties

    public function execute()
    {
        $result = $this->resultFactory->create(ResultFactory::TYPE_PAGE);
        return $result;
    }
}
```

Application\Http

/lib/internal/Magento/Framework/App/Http.php

```
public function launch()
{
    $frontName = $this->_request->getFrontName();
    $areaCode = $this->areaList->getCodeByFrontName($frontName);
    $this->state->setAreaCode($areaCode);
    $this->objectManager->configure($this->_configLoader->load($areaCode));
    $frontController = $this->objectManager->get(FrontControllerInterface::class);
    $result = $frontController->dispatch($this->_request);
    $result->renderResult($this->_response);
    return $this->_response;
}
```

Q & A