Queries

Já vimos como manipular uma entidade usando o conceito de um *TableGateway* mas em alguns casos precisamos criar consultas mais complexas, como fazer *join* entre tabelas ou criar *subqueries*. Para facilitar a criação de consultas que funcionem em diversos bancos de dados o framework fornece componentes, dentro do *namespace Zend\Db\Sql*.

Vamos ver alguns exemplos.

O primeiro passo que precisamos fazer é indicar o uso do componente:

```
use Zend\Db\Sql\Sql;
```

A construção do objeto *Sql* depende de uma conexão com o banco de dados. Para isso vamos usar a conexão configurada no projeto:

```
$adapter = $this->getServiceLocator()-
>get('Zend\Db\Adapter\Adapter');
```

Consulta simples, como um *SELECT * FROM posts*

```
$sql = new Sql($adapter);
$select = $sql->select()->from('posts');
$statement = $sql-
>prepareStatementForSqlObject($select);
$results = $statement->execute();
foreach ($results as $r) {
    echo $r['id'], ' - ', $r['title'], '<br>';
}
```

Consulta com clausula WHERE, como um *SELECT * FROM posts WHERE id = 10*:

```
$sql = new Sql($adapter);
$select = $sql->select()
    ->from('posts')
    ->where(array('id' => 10));

$statement = $sql-
>prepareStatementForSqlObject($select);
$results = $statement->execute();
foreach ($results as $r) {
    echo $r['id'], ' - ', $r['title'], '<br>';
}
```

Consulta com seleção de campos e cláusula WHERE, como um *SELECT id, title FROM posts WHERE id > 10*:

Neste caso podemos usar de duas formas:

```
$sql = new Sql($adapter);
$select = $sql->select()
    ->columns(array('id','title'))
    ->from('posts')
    ->where('id > 10');
$statement = $sql-
>prepareStatementForSqlObject($select);
$results = $statement->execute();
foreach ($results as $r) {
    echo $r['id'], ' - ', $r['title'], '<br>';
}
```

Ou usando o componente *Predicate*. Primeiro precisamos importar o componente:

```
use Zend\Db\Sql\Predicate\Predicate;
```

E usamos da seguinte forma:

```
$sql = new Sql($adapter);
$select = $sql->select()
    ->columns(array('id','title'))
    ->from('posts');

$where = new Predicate();
$where->greaterThan('id',10);
$select->getRawState($select::WHERE)-
>addPredicate($where);
$statement = $sql-
>prepareStatementForSqlObject($select);
$results = $statement->execute();
foreach ($results as $r) {
    echo $r['id'], ' - ', $r['title'], '<br>';
}
```

Consulta com seleção de campos e cláusula WHERE e INNER JOIN, como um *SELECT posts. id, posts. title, comments. description, comments.name FROM posts INNER JOIN comments ON comments.post_id = post.id WHERE posts.id > 10* order by title:

```
$sql = new Sql($adapter);
$select = $sql->select();
$select->columns(array('id','title'))
    ->from('posts')
```

```
    ->join(
        'comments',
        'comments.post_id = posts.id',
        array('description'),
        $select::JOIN_INNER
    )
    ->order(array('title'));
$where = new Predicate();
$where->greaterThan('posts.id',10);
$select->getRawState($select::WHERE)-
>addPredicate($where);
$statement = $sql-
>prepareStatementForSqlObject($select);
$results = $statement->execute();
foreach ($results as $r) {
    echo $r['id'], ' - ', $r['title'], '-',
$r['name'], ' - ', $r['description'] . '<br>';
}
```

Em qualquer momento é possível imprimir a consulta que foi gerada, o que pode ser útil para fins de debug:

```
echo $select->getSqlString();
```

Mais detalhes no manual do framework

Paginador

A *indexAction* que criamos no *PostController* possui um problema sério. Atualmente ela faz a busca de todos os registros da tabela *post* e simplesmente as envia para a *ViewModel* renderizar. Mas o que acontece

se tivermos milhares de registros na tabela? Um sério problema de performance, pois o usuário irá esperar por vários minutos até a página ser renderizada (isso se o servidor não cortar a transmissão antes). Para resolver esse problema usaremos o componente *Zend\Paginator*.

O *Paginator* facilita a paginação da informação e foi desenvolvido com os seguintes princípios:

- Paginar qualquer tipo de dados, não apenas o resultado de banco de dados. Podemos paginar *arrays* ou classes que implementem a interface *Iterator* do *SPL*.
- Recuperar apenas os resultados necessários para a visualização, aumentando a performance.
- Ser independente dos outros componentes do framework para facilitar ao desenvolvedor usá-lo inclusive em outros projetos.

Adicionando o paginador no PostController

Precisamos incluir novos *namespaces* no início do arquivo, para usarmos o paginador e o Sql:

```
use Zend\Paginator\Paginator;
use Zend\Paginator\Adapter\DbSelect as
PaginatorDbSelectAdapter;
use Zend\Db\Sql\Sql;
```

Agora vamos alterar a *indexAction*:

```
/**
* Mostra os posts cadastrados
```

```
* @return void
*/
public function indexAction()
{
    $adapter = $this->getServiceLocator()-
>get('Zend\Db\Adapter\Adapter');
    $sql = new Sql($adapter);
    $select = $sql->select()->from('posts');
    $paginatorAdapter = new
PaginatorDbSelectAdapter($select, $sql);
    $paginator = new Paginator($paginatorAdapter);
    $paginator->setCurrentPageNumber($this->params()-
>fromRoute('page'));

    return new ViewModel(array(
        'posts' => $paginator
    ));
}
```

Vamos aos detalhes do código.

```
$sql = new Sql($adapter);
$select = $sql->select()->from('posts');
```

Como vamos paginar informações vindas do banco de dados precisamos de um objeto *select* e um *sql*, que são necessários para o paginador funcionar.

```
$paginatorAdapter = new
PaginatorDbSelectAdapter($select, $sql);
$paginator = new Paginator($paginatorAdapter);
```

Conforme comentado no início do capítulo, o paginador é genérico, então precisamos passar como parâmetro o que iremos paginar, nesse caso um *DbSelect* (que eu apelidei de *PaginatorDbSelectAdapter* na importação do *namespace*).

```
$paginator->setCurrentPageNumber($this->params()-
>fromRoute('page'));
```

Essa configuração diz ao paginador em qual página ele se encontra. Caso não seja indicada nenhuma página pela URL (*page/3* por exemplo) a primeira página é apresentada.

Precisamos agora alterar a visão para que ela mostre o paginador.

Adicionamos o código abaixo no final do arquivo

module/Application/post/application/index/index.phtml:

```
<?php
echo $this->paginationControl(
    $posts,
    'Sliding',
    'partials/paginator/control.phtml'
);
?>
```

Estamos imprimindo o controlador de paginadores e passando como parâmetros o paginador (*$posts*), o formato (*Sliding*, similar ao paginador da página de busca do *Google*) e qual é o trecho de código que contém os links (*próxima página*, *página anterior*, etc).

O conteúdo do terceiro parâmetro é algo novo para nós.

Partials

No terceiro parâmetro da chamada ao *paginationControl* indicamos o *partial* onde consta o código dos links do paginador. *Partials* são porções de código que podemos salvar e usar em diversas visões e *layouts*. Isso ajuda bastante a organização de código, pois podemos compartilhar um trecho de *HTML* ou *PHP* entre diversas visões.

Precisamos criar o diretório de *partials* do módulo:

```
mkdir module/Application/view/partials
```

E criar o diretório para armazenar os *partials* do paginador:

```
mkdir module/Application/view/partials/paginator
```

O código do arquivo

mkdir module/Application/view/partials/paginator/control.phtml é:

```
<?php if ($this->pageCount > 1): ?>
<div class="navigation clearfix">
<?php if($this->current != 1): ?>
    <a href="<?= $this->url(null, array('page' => 1),
true) ?>">
        &laquo;
    </a>
<?php else: ?>
    <a href="<?= $this->url(null, array('page' => 1),
true) ?>" class="disabled">
        &laquo;
    </a>
```

```
<?php endif;?>

<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
    <a href="<?= $this->url(null, array('page' =>
$this->previous), true) ?>">
        &lt;
    </a>
<?php else: ?>
    <a href="#" class="disabled">&lt;</a>
<?php endif; ?>

<!-- Numbered page links -->
<?php foreach ($this->pagesInRange as $page): ?>
    <?php if ($page != $this->current): ?>
        <a href="<?= $this->url(null, array('page' =>
$page), true) ?>" class="btn">
            <?php echo $page; ?>
        </a>
    <?php else: ?>
        <a href="#" class="btn disabled"><?php echo
$page; ?></a>
    <?php endif; ?>
<?php endforeach; ?>
<!-- Next page link -->

<?php if (isset($this->next)): ?>
    <a href="<?= $this->url(null, array('page' =>
$this->next), true) ?>">
        &gt;
    </a>
<?php else: ?>
    <a href="#" class="disabled">&gt;</a>
```

```
<?php endif; ?>

<?php if($this->current != $this->pageCount): ?>
    <a href="<?= $this->url(null, array('page' =>
$this->pageCount), true) ?>">
        &raquo;
    </a>
<?php else: ?>
    <a href="<?= $this->url(null, array('page' => 1),
true) ?>" class="disabled">
        &raquo;
    </a>
<?php endif;?>

</ul>
<span class="info">
    <?php echo $this->current;?> de
    <?php echo $this->pageCount;?>
</span>
</div>
<?php endif; ?>
```

https://gist.github.com/4011827

Ele é baseado em um dos exemplos que a Zend fornece no manual do framework

Também foi preciso alterar a nossa rota para suportar a inclusão do número da página. No *module.config.php* alteramos a rota:

```
'post' => array(
    'type'    => 'segment',
    'options' => array(
```

```
        'route'    => '/post[/][:action][/page/:page]
[/:id]',
        'constraints' => array(
            'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
            'id'     => '[0-9]+',
        ),
        'defaults' => array(
            'controller' =>
'Application\Controller\Post',
            'action'     => 'index',
        ),
    ),
),
```

Cache

Introdução

A técnica de cache é muito usada para melhorar a performance de sites, sejam eles de grande tráfego ou não.

Teoricamente quase tudo pode ser armazenado em cache: resultados de consultas, imagens, arquivos css, arquivos js, trechos de código html, etc. A idéia é reaproveitarmos conteúdos já gerados e entregá-los rapidamente ao usuário. Um exemplo clássico é armazenar o resultado de uma consulta ao banco de dados, para não precisar acessar o banco todas as vezes.

No Zend Framework o cache é fornecido usando-se as classes do namespace *Zend\Cache*. O cacheamento é fornecido ao programador através de *"adapters"*.

Os principais adapters disponíveis são:

- *Apc*: usa o cache em memória fornecido pela extensão *APC (Alternative PHP Cache)* do PHP
- *Filesystem*: os dados do cache são armazenados em arquivos no sistema operacional
- *Memcached*: os dados serão salvos no *Memcached*, um servidor específico para cache, usado por grandes arquiteturas como o *Facebook*
- *Memory*: salva o cache na memória, na forma de *arrays*.

Também é possível criarmos nossos próprios *adapters*, apesar de ser necessário em poucos casos.

Vamos ver alguns exemplos de uso. O primeiro passo é criarmos entradas no nosso arquivo de configurações, no *application.config.php*:

```
'cache' => array(
    'adapter' => 'memory'
),
```

Cada *adapter* possui configurações especiais que podem mudar seu comportamento padrão. Mais detalhes podem ser encontrados no manual oficial do framework.

Configurando o Cache

A forma mais fácil de acessar o cache é usando o *ServiceManager*, para que as dependências de criação sejam facilmente criadas sempre que precisarmos. Para isso vamos configurar o nosso módulo *Admin* para que seja registrado o serviço Cache.

No arquivo *module/Admin/config/module.config.php* vamos adicionar:

```
'service_manager' => array(
  'factories' => array(
        'Session' => function($sm) {
            return new
Zend\Session\Container('ZF2napratica');
        },
        'Admin\Service\Auth' => function($sm) {
        $dbAdapter = $sm->get('DbAdapter');
        return new Admin\Service\Auth($dbAdapter);
    },
    'Cache' => function($sm) {
        $config = include __DIR__ .

'/../../../config/application.config.php';
        $cache = \Zend\Cache\StorageFactory::factory(
            array(
                'adapter' => $config['cache']
['adapter'],
                'plugins' => array(
                    'exception_handler' =>
                        array('throw_exceptions' =>
false),
                    'Serializer'
                ),
            )
        );

        return $cache;
    }
)
),
```

https://gist.github.com/4012076

Usando o cache

Com o serviço de cache registrado no *ServiceManager* podemos usá-lo em controladores ou serviços, conforme o exemplo abaixo.

```
$cache = $this->getServiceManager()->get('Cache');
ou
$cache = $this->getServiceLocator()->get('Cache');
```

Para adicionar um item ao cache basta:

```
$cache->addItem('chave', $valor);
```

Desta forma estamos incluindo no cache o conteúdo da variável *$valor* com o nome *'chave'*. O conteúdo da variável *$valor* vai ser serializado automaticamente, sendo ele uma string, inteiro ou mesmo objeto. A única ressalva é que alguns *adapters* tem restrições quanto ao que podem armazenar, sendo indicado uma leitura no manual caso algum item não consiga ser salvo no cache.

Para recuperarmos esse valor usamos:

```
$cache->getItem('chave');
```

Vários componentes do framework possuem integração com o cache, bastando indicar que estamos usando um cache. Um exemplo disso é o *Paginator* que estamos usando em nosso *Application\Controller\PostController*.

Podemos alterá-lo para usar o cache, conforme o exemplo:

```
$paginator = new Paginator($paginatorAdapter);
$cache = $this->getServiceLocator()->get('Cache');
$paginator->setCache($cache);
```

OBS: Não são todos os *adapters* que podem ser usados nesse exemplo, com o *Paginator*. No momento da escrita desse texto apenas o *Filesystem* e o *Memory* são suportados.

Traduções

Traduzindo o projeto

É cada vez mais comum a criação de projetos de software que precisem atender a públicos de países diferentes, com línguas distintas.

O Zend Framework possui um componente de tradução que facilita a criação deste tipo de projeto, o *Zend\I18n\Translator\Translator*.

Para fazer uso do componente vamos inicialmente criar um diretório no projeto onde iremos armazenar os arquivos das traduções, o *module/Application/language*.

As traduções podem ser salvas usando-se um dos padrões suportados:

- *PHP arrays*
- *Gettext*
- *Tmx*

- *Xliff*

  Vamos adicionar as linhas abaixo na configuração do módulo, no arquivo *module/Application/config/module.config.php*:

  ‘service_manager’ => array(

```
'factories' => array(
        'translator' =>
'Zend\I18n\Translator\TranslatorServiceFactory',
    ),
),
'translator' => array(
    'locale' => 'pt_BR',
    'translation_file_patterns' => array(
        array(
            'type'     => 'phparray',
            'base_dir' => __DIR__ .
'/../language',
            'pattern'  => '%s.php',
        ),
),

),
```

https://gist.github.com/4012079

Vamos agora criar o arquivo de traduções para o português do Brasil: *language/pt_BR.php*:

```
<?php
return array(
    'Home' => 'Página inicial',
```

```
    'logout' => 'Sair'
);
```

Podemos ter arquivos para cada língua, e mudar a configuração do *Translator* conforme a seleção do usuário ou a URL acessada.

Para usar as traduções vamos usar o *View Helper translate()* na nossa view ou layout. No arquivo *layout.phtml* podemos ter o seguinte código:

```
<li class="active">
    <a href="/"><?php echo $this->translate('Home') ?>
</a>
</li>
```

Caso a chave *Home* exista no arquivo de traduções ela será substituída pelo conteúdo traduzido e caso não exista a própria palavra *Home* vai ser exibida, sem causar erros para o usuário.

Traduzindo formulários

Também podemos usar o *translator* para traduzir as mensagens de validação de nossos formulários.

No *Application\Controller\PostController* usamos a validação da entidade *Post* para validar nosso formulário:

```
$form->setInputFilter($post->getInputFilter());
```

Podemos indicar que os filtros devem usar um *translator* para traduzir as mensagens de erro. No controlador adicionamos, antes da criação do formulário:

```
$translator = $this->getServiceLocator()
                    ->get('translator');
\Zend\Validator\AbstractValidator::setDefaultTranslato
r($translator);
```

Precisamos também adicionar as mensagens em português no nosso arquivo *pt_BR.php*:

```
<?php
return array(
    'Home' => 'Página inicial',
    'logout' => 'Sair',
    'notEmptyInvalid' => 'Valor nulo ou inválido',
    'isEmpty' => 'Valor obrigatório e não pode ser
nulo',
);
```

As chaves *notEmptyInvalid* e *isEmpty* encontram-se documentadas no manual, ou diretamente no código do validador, no caso o arquivo

vendor/zendframework/zendframework/library/Zend/Validator/NotEmpty.php

Outra funcionalidade interessante do *Translator* é a possibilidade de usar cache. Assim as traduções são armazenadas no cache, não necessitando que o arquivo seja aberto e processado todas as vezes que for usado. Para isso basta usar o método *setCache()* conforme o exemplo:

```
$translator = $this->getServiceLocator()-
>get('translator');
$cache = $this->getServiceLocator()->get('Cache');
$translator->setCache($cache);
```