Oxwall Documentation

Выпуск latest

Oxwall Foundation

авг. 14, 2017

Оглавление

Содержание:

Глава 1

Жизненный цикл приложения

Жизненный цикл приложения начинается с файла - **index.php**, который является единственной точкой входа (есть исключения такие как: *Cron (Планировщик задач)*, запуск скриптов обновления), другими словами все запросы поступающие от клиентов направляются прямо в этот файл (реализация паттерна - Front controller). Такое поведение задано с помощью директив файла **.htaccess** который находится в корне приложения и имеет следующие настройки:

```
Options +FollowSymLinks
RewriteEngine On

RewriteCond %{REQUEST_URI} !^/index\.php
RewriteCond %{REQUEST_URI} !/ow_updates/index\.php
RewriteCond %{REQUEST_URI} !/ow_updates/
RewriteCond %{REQUEST_URI} !/ow_cron/run\.php
RewriteCond %{REQUEST_URI} !/e500\.php
RewriteCond %{REQUEST_URI} !/captcha\.php
#RewriteCond %{REQUEST_URI} (/|\.php|\.html|\.htm|\.xml|\.feed|robots\.txt|\.raw|/[^.]*)$  [NC]
RewriteCond %{REQUEST_FILENAME} (/|\.php|\.html|\.htm|\.xml|\.feed|robots\.txt|\.raw|/[^.]*)$  [NC]
RewriteRule (.*) index.php
```

Инициализация программного окружения

После того как запрос был передан в файл **index.php** в нем начинается серия инициализаций программного окружения.

1. Определение констант.
2. Инициализация автолоадеров.
3. Инициализация профайлера.
4. Инициализация логирования.
5. Инициализация сессий.

Инициализация приложения

На этапе инициализации приложения происходит ряд важных установок и вызовов на которые можно повлиять из плагинов. Ниже перечислены наиболее важные из них:

Определение контекста приложения

В Oxwall существует 3 контекста: мобильный, десктопный и api. После определения контекста, приложение просто регистрирует пекеджпойинтеры (неймспейсы), т.е по сути подключает файлы инициализации из определенного контекста. На данный момент из кода плагинов не возможно программно поменять контекст.

Инициализация плагинов

На данном этапе у каждого установленного и активного плагина вызывается файл **init.php** (подробнее про структуру плагина - *Структура плагина*) который регистрирует ресурсы плагина, такие как : маршрутизация контроллеров (подробнее в разделе - *Маршрутизация и контроллеры*), системные события, регистрация автолоадеров итд). **Внимание в файле инициализации плагинов не должны выполняться тяжелые операции, их лучше зарегистрировать в файле обработки системных событий.**

После инициализации плагинов срабатывает системное событие которое оповещает приложение о том, что инициализация плагинов закончена:

```
<?php

$event = new OW_Event(OW_EventManager::ON_PLUGINS_INIT);
OW::getEventManager()->trigger($event);
```

Это событие можно использовать в свои целях, к примеру после загрузки всех плагинов можно выполнить логику своего плагина.

```
<?php

class MYSUPERPLUGIN_CLASS_EventHandler
{
    public function genericInit()
    {
      OW::getEventManager()->bind(OW_EventManager::ON_PLUGINS_INIT, array($this, 'afterPluginsInit
↪'));
    }

    public function afterPluginsInit()
    {
        // do something...
    }
}
```

Инициализация темы

Далее приложение пытается определить тему по умолчанию (тема по умолчанию задается в настройках админ панели) и активировать ее, взяв название темы из системных настроек приложения. Однако

плагины могут повлиять на этот процесс, для этого нужно подписаться на системное событие и передать в него название другой темы. Пример системного события:

```
<?php

$activeThemeName = OW::getEventManager()->call('base.get_active_theme_name');
$activeThemeName = $activeThemeName ? $activeThemeName : OW::getConfig()->getValue('base',
↪'selectedTheme');
```

Меняем дефолтную тему

```
<?php

class MYSUPERPLUGIN_CLASS_EventHandler
{
    public function genericInit()
    {
      OW::getEventManager()->bind('base.get_active_theme_name', array($this, 'changeTheme'));
    }

    public function changeTheme()
    {
        return 'new_theme_name';
    }
}
```

Инициализация объекта ответа

В самом конце инициализации приложения, создается объект ответа "Response", который и будет в последствии возвращен клиенту. В этом объекте содержаться заголовки ответа, которые можно менять по ходу жизненного цикла приложения. А также объект документ "Document", который содержит непосредственно сгенерированный html (json, xml, итд. все зависит от контекста прилождения) код. Сторонние плагины имеют возможность изменять содержимое объекта документ, для этого существует ряд системных событий

1. Событие срабатывает, перед отправкой сгенерированного контента клиенту

```
<?php

$event = new OW_Event(OW_EventManager::ON_BEFORE_DOCUMENT_RENDER);
OW::getEventManager()->trigger($event);
```

2. Событие срабатывает, после отправки сгенерированного контента клиенту

```
<?php

$event = new OW_Event(OW_EventManager::ON_AFTER_DOCUMENT_RENDER);
OW::getEventManager()->trigger($event);
```

После инициализации приложения срабатывает событие:

```
<?php

$event = new OW_Event(OW_EventManager::ON_APPLICATION_INIT);
OW::getEventManager()->trigger($event);
```

Маршрутизация

После инициализации приложения, **oxwall** пытается определить название контроллера, и метода которому нужно передать управление. Для этого сравнивается URI объекта Request со списком зарегистрированных маршрутов. Если маршрут не найден то генерируется исключительная ситуация и идет перенаправление на контроллер отображающий 404 ошибку. После того, как приложение определило нужный маршрут срабатывает системное событие:

```
<?php

$event = new OW_Event(OW_EventManager::ON_AFTER_ROUTE);
```

На этом этапе мы уже знаем адресата (контроллер и метод), мы можем использовать это сообщение в наших целях, к примеру сделать редирект на другую страницу, проверить настройки доступа к странице, итд.

Обработка контроллера

На данном этапе идет создание объекта контроллера и вызов метода **init** внутри него (метод используется для инициализации окружения контроллера, загрузки ресурсов итд), а также вызов метода **(action)** внутри контроллера в котором и сосредоточена вся логика запроса. Если в методе контроллера не инициализирован файл представления **(view)**

```
<?php

class MYSUPERPLUGIN_CTRL_Base extends OW_ActionController
{
  function test()
  {
      OW::getDocument()->setTemplate(OW::getPluginManager()->getPlugin('mysuperplugin')->
↪getRootDir() . 'views/controllers/custom.html');
  }
}
```

То приложение попытается найти нужный шаблон самостоятельно, используя название контроллера и метода, к примеру **my_controller_test.html** и добавить его в метод контроллера. После того как, контроллер отработал срабатывает системное сообщение:

```
<?php

  $event = new OW_Event(OW_EventManager::ON_AFTER_REQUEST_HANDLE);
```

Компиляция и отправка сгенерированного документа

После того, как приложение получило контент от метода контроллера оно обрамляет полученный контент другим файлом представления - **master page** (Реализация паттерна Decorator) В коде контроллера можно легко подменять master page :

```
<?php

class MYSUPERPLUGIN_CTRL_Base extends OW_ActionController
{
```

```
  function test()
  {
        // change the master page
        OW::getDocument()->getMasterPage()->
                setTemplate(OW::getPluginManager()->getPlugin('mysuperplugin')->getRootDir() .
↪'views/master_pages/custom.html');
  }
}
```

После обрамления контента он отправляется клиенту и срабатывает событие:

```
<?php

$event = new OW_Event(OW_EventManager::ON_FINALIZE);
```

В самом конце, жизненного цикла приложения срабатывает систменое сообщение

```
<?php

$event = new OW_Event('core.exit');
```

Глава 2

Основные сервисы приложения

Глава 3

Маршрутизация и контроллеры

Маршрутизация помогает определять и доставлять все входящие запросы от клиентов в контроллеры, которые в свою очередь обрабатывают и возвращают результат назад клиенту.

Регистрация маршрутов

В платформе **Oxwall** есть два способа использования маршрутизации для того, чтобы запросить или отправить данные на сайт.

Именованные маршруты

При использовании именованных маршрутов, мы сами создаем шаблон того как будет выглядеть адрес запроса и какие параметры он будет принимать. Основным преимуществом использования именованных маршрутов это - создания красивых и понятных адресов, которые хорошо воспринимаются человеком. Также именованные маршруты хороши для SEO оптимизации. Регистрация маршрутов происходит в файле - **init.php** (подробнее в разделе - *Структура плагина*), ниже пример регистрации нескольких маршрутов:

```
<?php

    OW::getRouter()->addRoute(new OW_Route('superplugin_list_index', 'superplugin/', 'SUPERPLUGIN_
↪CTRL_Index', 'viewList'));
    OW::getRouter()->addRoute(new OW_Route('superplugin_item_edit', 'superplugin/edit/:id/',
↪'SUPERPLUGIN_CTRL_Index', 'edit'));
```

При регистрации маршрута нам нужно указать:

1. Название.
2. Адрес с параметрами если нужно.
3. Название контроллера.

4. Название метода действия куда придет запрос на обработку.

После того как именованные маршруты зарегистрированы, мы можем их использовать в PHP коде (в контроллере к примеру) или напрямую в коде шаблонов (подробнее в разделе - *Представление и компоненты*):

Внутри кода шаблона страницы:

```
<html>
    <body>
        <a href="{url_for_route for='superplugin_list_index'}">View list</a>
        <br />
        <a href="{url_for_route for="superplugin_item_edit:[id=>`$item->id`]"}">Edit item</a>
    </body>
</html>
```

Внутри PHP кода:

```
<?php

    $listUrl = OW::getRouter()->urlForRoute('superplugin_list_index');
    $editUrl = OW::getRouter()->urlForRoute('superplugin_item_edit',  array('id' => $item->id));
```

Маршруты по умолчанию

Маршруты по умолчанию не нуждаются в инициализации и для того, чтобы их использовать вам нужно лишь указать название контроллера и название действия внутри контроллера.

Внутри кода шаблона страницы:

```
<html>
    <body>
        <a href="{url_for for="MYSUPERPLUGIN_CTRL_Index:ajaxSaveItem"}">Save</a>
    </body>
</html>
```

Внутри PHP кода:

```
<?php

    $saveUrl = OW::getRouter()->urlFor('MYSUPERPLUGIN_CTRL_Index', 'ajaxSaveItem');
```

Так как маршруты по умолчанию возвращают не красивые адреса с точки зрения **SEO** их стоит использовать только для внутренних целей (к примеру отправка ajax запросов), для всех остальных случаев рекомендуется использовать именованные адреса.

Контроллеры

После того как маршрут установлен то управление передается коду контроллера которому нужно обработать входящий запрос и вернуть данные в подходящем формате. Ниже приведен пример кода контроллера с описанием методов которые были использованы выше при описании маршрутизации.

```
<?php

    class MYSUPERPLUGIN_CTRL_Index extends OW_ActionController
```

```
{
    /**
     * View list
     */
    public function viewList()
    {
        // do some logic
        //...

        // init view variables
        $this->assign('foo', 'bar');
        // ...
    }

    /**
     * Edit item
     *
     * @param array $params
     */
    public function edit($params = array())
    {
        $itemId = isset($params['id']) ? (int) $params['id'] : 0;

        // do some logic
        //...

        // init view variables
        $this->assign('foo', 'bar');
        // ...
    }

    /**
     * Ajax save item
     */
    public function ajaxSaveItem()
    {
       if ( OW::getRequest()->isAjax() )
       {
            // do some logic
            //...

            // show result
            echo json_encode(
                'foo' => 'bar'
            );
       }

       exit;
    }
}
```

Глава 4

Представление и компоненты

Глава 5

Формы

Платформа **Oxwall** предоставляет достаточно богатый функционал для приема и обработки данных от клиента используя **HTML формы**. Ниже мы рассмотрим основные моменты по созданию форм, приему и обработки, валидации и фильтрации данных формы.

Фильтры

Фильтры в формах необходимы для первичной очистки вводимых данных перед их валидацией, к примеру если нам не нужны **html** теги или пробелы, мы с легкостью можем от них избавится добавив нужный фильтр к полю формы. Ниже перечислены встроенные фильтры:

ow_core/filter.php

1. **StripTagsFilter** - вырезает все html теги
2. **TrimFilter** - обрезает пробелы из строки

У вас есть возможность создать собственный фильтр реализуя методы интерфейса - **OW_IFilter**

Валидаторы

Основная задача валидаторов - это проверить корректность вводимых данных в поля формы. Для этого в платформе **Oxwall** предусмотрено огромное количество валидаторов:

ow_core/validator.php

1. **RequiredValidator** - проверяет были ли введены данные.
2. **WyswygRequiredValidator** - проверяет были ли введены данные в визуальный редактор (для того, чтобы проверить данные сперва удаляет все html теги, затем проверяет оставшийся текст).
3. **StringValidator** - проверяет строку или массив строк (опционально можно задать минимальную и максимальную длину строки).

4. **RegExpValidator** - проверяет строку или массив строк на соответствует регулярному выражению.
5. **EmailValidator** - проверяет email или массив email.
6. **UrlValidator** - проверяет url или массив url.
7. **AlphaNumericValidator** - проверяет строку или массив строк на наличие в ней только цифр и символов латинского алфавита.
8. **InArrayValidator** - проверяет строку на ее наличие в списке предопределенных значений.
9. **IntValidator** - проверяет число или массив чисел (опционально можно задать минимальное и максимальное число).
10. **FloatValidator** - проверяет действительное число или массив действительных чисел (опционально можно задать минимальное и максимальное число).
11. **DateValidator** - проверяет дату или массив дат (опционально можно задать формат даты, а также минимальный и максимальный год).
12. **CaptchaValidator** - проверяет значение captcha.
13. **RangeValidator** - проверяет число или массив чисел на нахождение их в некотором интервале.

Не смотря на большое количество встроенных валидатров, у вас всегда есть возможность создать собственный валидатор реализуя методы абстрактного класса - **OW_Validator**

Создание формы

Пример создания формы :

classes/user_form.php

```php
<?php

    class MYSUPERPLUGIN_CLASS_UserForm extends Form
    {
        /**
         * Class constructor
         */
        public function __construct(MYSUPERPLUGIN_BOL_User $user = null)
        {
            // set the form name
            parent::__construct('user-form');

            // allow upload files
            $this->setEnctype(Form::ENCTYPE_MULTYPART_FORMDATA);

            // add user name field
            $userName = new TextField('userName');
            $userName->addFilter(new StripTagsFilter());
            $userName->setRequired(true);
            $userName->setValue(($user ? $user->userName : null));
            $this->addElement($userName);

            // add user email field
            $email = new TextField('email');
            $email->setRequired(true);
            $email->addValidator(new EmailValidator());
```

```
            $email->setValue(($user ? $user->email : null));
            $this->addElement($email);

            // add user age field
            $age = new TextField('age');
            $age->setRequired(true);
            $age->addValidator(new IntValidator());
            $age->setValue(($user ? $user->age : null));
            $this->addElement($age);

            // add user description field
            $description = new Textarea('description');
            $description->addFilter(new StripTagsFilter());
            $description->addFilter(new TrimFilter());
            $description->setValue(($user ? $user->description : null));
            $this->addElement($description);

            // add user image field
            $image = new FileField('image');
            $image->addValidator(new CustomImageValidator());
            $this->addElement($image);
        }
    }

    /**
     * Custom image validator
     */
    class CustomImageValidator extends OW_Validator
    {
        /**
         * Class constructor
         */
        public function __construct()
        {
            $this->setErrorMessage(OW::getLanguage()->text('mysuperplugin', 'image_validator_error_
↪message'));
        }

        /**
         * Is image valid
         *
         * @param mixed $value
         * @return boolean
         */
        public function isValid( $value )
        {
            if ( !empty($value['name']) && isset($value['tmp_name'], $value['error']) )
            {
                // validate image
                return (int) $value['error'] === 0 &&
                        is_uploaded_file($value['tmp_name']) &&
                        UTIL_File::validateImage($value['name']) && getimagesize($value['tmp_name
↪']);
            }

            return true;
        }
    }
```

Валидация формы

Для проверки введенных данных вы можете использовать ниже приведенный кусок кода в коде контроллера:

controller/user.php

```
<?php

    class MYSUPERPLUGIN_CTRL_User extends OW_ActionController
    {
        /**
         * Service
         *
         * @var MYSUPERPLUGIN_BOL_Service
         */
        protected $service;

        /**
         * Constructor
         */
        public function __construct()
        {
            parent::__construct();
            $this->service = MYSUPERPLUGIN_BOL_Service::getInstance();
        }

        /**
         * Add user
         */
        public function addUser()
        {
            // check permission
            $isAddAllowed = OW::getUser()->isAuthorized('superplugin', 'add_user');

            if ( !isViewAllowed )
            {
                // get error message
                $errorMessage = BOL_AuthorizationService::getInstance()->getActionStatus(
↪'superplugin', 'add_user');
                throw new AuthorizationException($errorMessage['msg']);
            }

            // validate the form data
            $form = new MYSUPERPLUGIN_CLASS_UserForm();

            // make certain to merge the files info!
            $post = array_merge_recursive(
                $_POST,
                $_FILES
            );

            // validate the form
            if ( OW::getRequest()->isPost() && $form->isValid($post) )
            {
                // get validated and filtered form values
                $formValues = $form->getValues();
```

```
                // add a new user
                $userBol = new MYSUPERPLUGIN_BOL_User;
                $userBol->userName = $formValues['userName'];
                $userBol->email = $formValues['email'];
                $userBol->age = $formValues['age'];
                $userBol->description = !empty($formValues['description']) ? $formValues[
↪'description'] : null;
                $userBol->createdStamp = time();
                $userBol->ownerId = OW::getUser()->getId();

                $this->service->addUser($userBol, $formValues['image']);

                OW::getFeedback()->info(OW::getLanguage()->text('superplugin', 'user_successfully_
↪added'));
                $this->redirect();
            }

            // init components
            $this->addForm($form);

            // init page settings
            OW::getDocument()->setHeading(OW::getLanguage()->text('superplugin', 'page_title_add_
↪user'));
            OW::getDocument()->setHeadingIconClass('ow_ic_user');
            OW::getDocument()->setTitle(OW::getLanguage()->text('superplugin', 'page_title_add_user
↪'));
        }
    }
```

Отображение формы

views/controllers/add_user.html

```
{form name="user-form"}
    <table class="ow_table_1 ow_form">
        <tbody class="ow_paging">
            <tr>
                <td class="ow_td_label">* {text key="superplugin+user_name"}</td>
                <td class="ow_value ow_valign_middle">
                    <div>{input name="userName"}</div>
                    <div>{error name="userName"}</div>
                </td>
            </tr>
            <tr>
                <td class="ow_td_label">* {text key="superplugin+email"}</td>
                <td class="ow_value ow_valign_middle">
                    <div>{input name="email"}</div>
                    <div>{error name="email"}</div>
                </td>
            </tr>
            <tr>
                <td class="ow_td_label">* {text key="superplugin+age"}</td>
                <td class="ow_value ow_valign_middle">
                    <div>{input name="age"}</div>
                    <div>{error name="age"}</div>
```

```
                </td>
            </tr>
            <tr>
                <td class="ow_td_label">{text key="superplugin+description"}</td>
                <td class="ow_value ow_valign_middle">
                    <div>{input name="description"}</div>
                    <div>{error name="description"}</div>
                </td>
            </tr>
            <tr>
                <td class="ow_td_label">{text key="superplugin+image"}</td>
                <td class="ow_value ow_valign_middle">
                    <div>{input name="image"}</div>
                    <div>{error name="image"}</div>
                </td>
            </tr>
        </tbody>
    </table>
    <div class="clearfix">
        {submit name="save" class="ow_ic_save ow_submit ow_right"}
    </div>
{/form}
```

Глава 6

Слой бизнес логики

В платформе **oxwall** все классы для работы с бизнес логикой принято хранить в директории **bol** внутри плагина (подробнее в разделе - "*Структура плагина*").

База данных

Для работы с базой данных нужно описать классы **DTO** (Data Transfer Object) и **DAO** (Data Access Object) для для каждой таблицы базы данных используемых в плагине.

DTO - Data Transfer Object

Класс **DTO** описывает сущность таблицы и используется при добавлении или изменении данных в таблице. Пример класса описывающий сущность - "**message**":

```php
<?php

    class MYSUPERPLUGIN_BOL_Message extends OW_Entity
    {
      /**
       *  Title
       *
       * @var string
       */
       public $title;

      /**
       *  Message
       *
       * @var string
       */
       public $message;
```

```
    /**
     *  Created timestamp
     *
     * @var integer
     */
    public $createdTimestamp;

    /**
     *  Approved flag
     *
     * @var integer
     */
    public $approved;
}
```

Как видно на примере, класс **MYSUPERPLUGIN_BOL_Message** описывает структуру таблицы **message**, перечисляя в этом классе все поля, а также значения по умолчанию этих полей если нужно.

DAO - Data Access Object

В классе **DAO** описывается вся бизнес логика работы с базой данных: поиск, создание и удаление записей. Все классы **DAO** всегда реализуют паттерн - Singleton Object. Пример класса **DAO**:

```
<?php

class MYSUPERPLUGIN_BOL_MessageDao extends OW_BaseDao
{
    /**
     * Approved status
     */
    const APPROVED_STATUS = 1;

    /**
     * Disapproved status
     */
    const DISAPPROVED_STATUS = -1;

    /**
     * Singleton instance.
     *
     * @var MYSUPERPLUGIN_BOL_MessageDao
     */
    private static $classInstance;

    /**
     * Returns an instance of class (singleton pattern implementation).
     *
     * @return MYSUPERPLUGIN_BOL_MessageDao
     */
    public static function getInstance()
    {
        if ( self::$classInstance === null )
        {
            self::$classInstance = new self();
        }
```

```
        return self::$classInstance;
    }

    /**
     * Constructor.
     */
    protected function __construct()
    {
        parent::__construct();
    }

    /**
     * Get DTO class name
     *
     * @return string
     */
    public function getDtoClassName()
    {
        return 'MYSUPERPLUGIN_BOL_Message';
    }

    /**
     * Get table name
     *
     * @return string
     */
    public function getTableName()
    {
        return OW_DB_PREFIX . 'mysuperplugin_message';
    }

    /**
     * Delete message
     *
     * @param integer $userId
     * @param integer $recipientId
     * @return void
     */
    public function deleteMessage($userId, $recipientId)
    {
        $example = new OW_Example();
        $example->andFieldEqual('userId', $userId);
        $example->andFieldEqual('recipientId', $recipientId);
        $this->deleteByExample($example);
    }

    /**
     * Find active messages
     *
     * @param integer $limit
     * @return array
     */
    public function findActiveMessages($limit)
    {
        $example = new OW_Example();
        $example->setOrder('`id` ASC');
        $example->andFieldEqual('status', self::APPROVED_STATUS);
        $example->setLimitClause(0, $limit);
```

```
            return $this->findListByExample($example);
        }

        /**
         * Find active messages using the raw sql
         *
         * @param integer $limit
         * @return array
         */
        public function findActiveMessagesRawSql($limit)
        {
            $query = "SELECT * FROM `" . $this->getTableName() . "` LIMIT ?";

            return $this->dbo->queryForList($query, array($limit));
        }
    }
```

В данном классе нужно указать название таблицы базы данных в методе **getTableName**, а также название **DTO** класса в методе **getDtoClassName**. Следует отметить тот факт, что если вы работаете только с одной таблицей то необходимо использовать конструктор запросов - **OW_Example**, а если в запросе необходимы сложные объединения тогда нужно писать сырые запросы к базе данных, как это сделано в методе - **findActiveMessagesRawSql**.

Сервис

Класс **service.php** является центральным для плагина, так как именно его нужно использовать в качестве провайдера данных, а также инкапсулировать в нем всю бизнес логику плагина (даже если вы не работаете с БД). Стоит отметить, что если вы используете работу с базой данных то класс **service.php** будет выступать как бы промежуточным слоем, т.е нельзя из кода контроллеров или откуда-либо еще напрямую обращаться к классам для работы с базой данных для этого нужен сервис. Класс **service.php** так же как и классы **DAO** всегда реализует паттерн - **Singleton**. Пример сервиса и инкапсуляции в нем работы с базой данных :

```
<?php

    class MYSUPERPLUGIN_BOL_Service
    {
        /**
         * Class instance
         *
         * @var MYSUPERPLUGIN_BOL_Service
         */
        private static $classInstance;

        /**
         * Message DAO
         *
         * @var MYSUPERPLUGIN_BOL_MessageDao
         */
        private $messageDao;

        /**
         * Class constructor
         */
```

```
        private function __construct()
        {
            $this->messageDao = MYSUPERPLUGIN_BOL_MessageDao::getInstance();
        }

        /**
         * Returns class instance
         *
         * @return MYSUPERPLUGIN_BOL_Service
         */
        public static function getInstance()
        {
            if ( self::$classInstance === null )
            {
                self::$classInstance = new self();
            }

            return self::$classInstance;
        }

        /**
         * Add message
         *
         * @param MYSUPERPLUGIN_BOL_Message $messageDto
         * @return void
         */
        public function addMessage(MYSUPERPLUGIN_BOL_Message $messageDto)
        {
            $messageDto->createdTimestamp = time();
            $messageDto->approved = MYSUPERPLUGIN_BOL_MessageDao::APPROVED_STATUS;
            $this->messageDao->save($messageDto);
        }

        /**
         * Remove message
         *
         * @param integer $messageId
         * @return void
         */
        public function deleteMessage($messageId)
        {
            $this->messageDao->deleteById($messageId);
        }

        /**
         * Find active messages
         *
         * @param integer $limit
         * @return array
         */
        public function findActiveMessages($limit)
        {
            return $this->messageDao->findActiveMessages($limit);
        }

        // ... etc
    }
```

PS: По возможности нужно максимально выносить логику из контроллеров и компонентов в сервисы и делать эти методы готовыми к повторному использованию.

Глава 7

Авторизация

В платформе **Oxwall** авторизация реализована на основе ролей пользователей. Каждая роль в платформе разрешает или запрещает пользователям определенный набор действий.

Регистрация действий

Регистрация списка действий плагина производится в файле **install.php** (подробнее в разделе - *Структура плагина*). Ниже приведен пример регистрации нескольких действий:

install.php

```php
<?php

    $authorization = OW::getAuthorization();

    $authorization->addGroup('superplugin');

    $authorization->addAction('superplugin', 'view_items');
    $authorization->addAction('superplugin', 'edit_items');
    $authorization->addAction('superplugin', 'delete_items');
```

В приведенном примере при установке плагина мы создаем новую группу авторизации - **superplugin** (название группы авторизации должно совпадать с ключем плагина описанным в plugin.xml). Далее к созданной группе добавляем список действий которые в дальнейшем будем использовать к примеру для проверки прав пользователей просматривать, редактировать, удалять контент итд.

После регистрации списка действий нужно добавить для них описание для того, чтобы пользователи и администратор сайта смогли понять за, что отвечают эти действия. Для этого нам нужно подписаться на системное событие в файле - **classes/event_handler.php** (подробнее в разделе - *Системные события*) и вернуть описания этих действий:

classes/event_handler.php

```php
<?php

    class MYSUPERPLUGIN_CLASS_EventHandler
    {
        /**
         * Class instance
         *
         * @var MYSUPERPLUGIN_CLASS_EventHandler
         */
        private static $classInstance;

        /**
         * Class constructor
         */
        private function __construct()
        {}

        /**
         * Get instance
         *
         * @return MYSUPERPLUGIN_CLASS_EventHandler
         */
        public static function getInstance()
        {
            if ( self::$classInstance === null )
            {
                self::$classInstance = new self();
            }

            return self::$classInstance;
        }

        /**
         * Generic init (should be started for all contexts such as: mobile, desktop, cli and api)
         *
         * @return void
         */
        public function genericInit()
        {
            $em = OW::getEventManager();

            // init auth labels
            $em->bind('admin.add_auth_labels', array($this, 'addAuthLabels'));
        }

         /**
          * Add auth labels
          *
          * @param BASE_CLASS_EventCollector $event
          * @return void
          */
         public function addAuthLabels(BASE_CLASS_EventCollector $event)
         {
             $event->add(
                 array(
                     'superplugin' => array(
                         'label' => OW::getLanguage()->text('superplugin', 'auth_group_label'),
                         'actions' => array(
```

```
                            'view_items' => OW::getLanguage()->text('superplugin', 'auth_action_
↪label_view_items'),
                            'edit_items' => OW::getLanguage()->text('superplugin', 'auth_action_
↪label_edit_items'),
                            'delete_items' => OW::getLanguage()->text('superplugin', 'auth_action_
↪label_delete_items')
                        )
                    )
                )
            );
        }
    }
```

В файле **init.php** который запускается при каждом запросе от клиента, запускаем метод который подписывается на системные события:

init.php

```
<?php

    MYSUPERPLUGIN_CLASS_EventHandler::getInstance()->genericInit();
```

Проверка прав пользователей

Для того, чтобы проверить разрешено ли текущему пользователю выполнить те или иные действия нужно выполнить код:

```
<?php

    // is view items allowed for current user ?
    $isViewAllowed = OW::getUser()->isAuthorized('superplugin', 'view_items');

    if ( !$isViewAllowed )
    {
        // get error message
        $errorMessage = BOL_AuthorizationService::getInstance()->getActionStatus('superplugin',
↪'view_items');
        throw new AuthorizationException($errorMessage['msg']);
    }
```

Модераторы

Модераторы эта группа пользователей которых назначает администратор сайта в админ панели. Администратор сайта может разрешить управлять одним и более плагином, что в свою очередь значит, что модератору будут разрешены любые действия в плагине по управлению контентом пользователей.

Пример кода который проверяет является ли текущий пользователь модератором:

```
<?php

    // is current user a moderator of the "superplugin" ?
    $isModerator = OW::getUser()->isAuthorized('superplugin');
```

```
// is it an content owner or a moderator?
if ( $isContentOwner || $isModerator )
{
    // do some logic
}
```

Глава 8

Cron (Планировщик задач)

Фоновые задачи в **oxwall** занимают одно из основных мест в приложении, так как приложение работает с большими массивами данных (профили пользователей, фотографии, форумы, итд) то зачастую приходится обрабатывать эти данные порционно. При установке приложения одно из требований это - настройка планировщика задач (Cron) на запуск файла **“ow_cron/run.php”** каждую минуту.

У вас может возникнуть вопрос - “Зачем мне запускать мою задачу каждую минуту если мне хватит и одного запуска один раз в час”, это верно, но данный подход избавляет разработчиков от необходимости в ручную прописывать каждую свою задачу в cron и тем самым облегчает работу администраторам сайта. Задать время выполнения своей задачи вы сможете позже, самостоятельно в файле cron.php.

Инициализация планировщика задач

Для инициализации файла cron, разработчику достаточно создать файл cron.php в корне своего плагина (подробнее про структуру плагина можно прочитать в разделе - *Структура плагина*) и реализовать необходимую логику внутри него, пример такого файла:

```php
<?php

class MYSUPERPLUGIN_Cron extends OW_Cron
{
    /**
     * Run command every minute
     *
     * @return void
     */
    public function run()
    {
        // do my own logic
    }
}
```

Как видно на примере в методе **run** будет выполняться некоторая логика каждую минуту (это время по умолчанию). Если необходимо использовать другие промежутки времени, их можно зарегистрировать в конструкторе класса или переопределить метод **getRunInterval** который изменит время выполнения метода **run**:

```php
<?php

class MYSUPERPLUGIN_Cron extends OW_Cron
{
    /**
     * Class constructor
     */
    public function __construct()
    {
        parent::__construct();
        $this->addJob('doSomething', 5);
    }

    /**
     *  Do something every 5 minutes
     *
     * @return void
     */
    public function doSomething()
    {
        // do my own logic
    }

    /**
     *  Return run interval in minutes
     *
     * @return int
     */
    public function getRunInterval()
    {
        return 10; // minutes
    }

    /**
     * Run command every 10 minutes
     *
     * @return void
     */
    public function run()
    {
        // do my another own logic
    }
}
```

Стоит отметить тот факт, что даже если администратор сайта не настроил cron, то выполнение фоновых задач все равно будет продолжаться. Это реализовано с помощью клиентских запросов, т.е когда клиент запрашивает страницу, на сервер посылается ajax запрос запускающий весь стек фоновых задач.

Глава 9

Системные события

Глава 10

Файлы переводов

Зачастую в плагинах необходимо использовать переводы на различные языки некоторых видов контента сайта: ссылки, пункты меню, итд. Для этого можно воспользоваться системой переводов **oxwall**.

Регистрация префикса языка

Перед началом разработки плагина нужно зарегистрировать новый префикс языка плагина (префикс это просто категория переводов), для того, чтобы в последующем можно было добавлять новые переводы используя этот префикс. Для регистрации префикса нужно добавить запись в таблицу **ow_base_language_prefix**

1. prefix: **mysuperplugin**
2. label: **My Super Plugin**

В качестве префикса языка принято использовать ключ плагина.

Добавление переводов

Для того, чтобы добавить переводы нужно перейти в админ панель на страницу управления языками **(your_site.com/admin/settings/dev-tools/languages)**. На этой странице найти и нажать кнопку **"Add new text"**, в всплывающем окне нужно выбрать ранее созданный нами префикс **"My Super Plugin"** и добавить необходимый ключ (ключ это указание на значение языка) и значения переводов.

Экспортирование переводов

После того, как все работы с плагином закончены нужно сделать экспорт переводов и подключить полученный файл в плагин. Для этого на странице **your_site.com/admin/settings/dev-tools/languages/mod** нужно нажать на кнопку **"Export"** и выбрать ранее созданный префикс **"My Super Plugin"**. Полученный в итоге zip архив нужно скопировать в корень плагина.

Подключение переводов

Для того, чтобы подключить сгенерированный zip файл с переводами его нужно подключить в инсталляционный файл плагина (*Структура плагина*), пример как это можно сделать:

```
<?php

    // import langs
    OW::getLanguage()->importPluginLangs(OW::getPluginManager()->getPlugin('mysuperplugin')->
↪getRootDir() . 'langs.zip', 'mysuperplugin');
```

Глава 11

Виджеты

Виджеты - это независимые элементы на страницах сайта, которые легко можно удалить, добавить из кода плагинов. С помощью виджетов можно решать различные задачи по отображению контента, к примеру отобразить список пользователей на странице или список последних фотографий на странице профайла, организовать поиск итд.

Создание виджета

Для начала нам нужно создать пустой файл виджета внутри директории плагина, в поддиректории **components** (подробнее в разделе - "*Структура плагина*"). Затем добавляем в него PHP код:

```php
<?php

class MYSUPERPLUGIN_CMP_ExampleWidget extends BASE_CLASS_Widget
{
    /**
     * Default repeat greeting count
     */
    const DEFAULT_REPEAT_GREETING_COUNT = 10;

    /**
     * Repeat greeting count
     *
     * @var integer
     */
    protected $repeatGreetingCount;

    /**
     * Repeat greeting
     *
     * @var boolean
     */
    protected $repeatGreeting;
```

```
    /**
     * Class constructor
     *
     * @param BASE_CLASS_WidgetParameter $paramObj
     */
    public function __construct( BASE_CLASS_WidgetParameter $paramObj )
    {
        parent::__construct();

        // process custom widget's settings
        $this->repeatGreeting = !empty($paramObj->customParamList['repeatGreeting']);

        $this->repeatGreetingCount = isset($paramObj->customParamList['repeatGreetingCount'])
            ? (int) $paramObj->customParamList['repeatGreetingCount']
            : self::DEFAULT_REPEAT_GREETING_COUNT;

    }

    /**
     * Init view variables
     *
     * @return void
     */
    public function onBeforeRender()
    {
        parent::onBeforeRender();

        // assign view variables
        $this->assign('greetingMesasge', 'Hello World');
        $this->assign('date', date('Y-m-d'));
        $this->assign('repeatGreetingCount', $this->repeatGreetingCount);
        $this->assign('repeatGreeting', $this->repeatGreeting);
    }

    /**
     * Get widget auth access
     *
     * @return string
     */
    public static function getAccess()
    {
        return self::ACCESS_MEMBER;
    }

    /**
     * Get custom widget setting list
     *
     * @return array
     */
    public static function getSettingList()
    {
        $settingList = array();

        $settingList['repeatGreetingCount'] = array(
            'presentation' => self::PRESENTATION_NUMBER,
            'label' => OW::getLanguage()->text('mysuperplugin', 'widget_example_count_repeat_
↪greeting'),
```

```
            'value' => 3
        );

        $settingList['repeatGreeting'] = array(
            'presentation' => self::PRESENTATION_CHECKBOX,
            'label' => OW::getLanguage()->text('mysuperplugin', 'widget_example_repeat_greeting'),
            'value' => true
        );

        return $settingList;
    }

    /**
     * Get standard widget settings
     *
     * @return array
     */
    public static function getStandardSettingValueList()
    {
        return array(
            self::SETTING_TITLE => OW::getLanguage()->text('mysuperplugin', 'example_widget_title
↪'),
            self::SETTING_ICON => self::ICON_INFO,
            self::SETTING_SHOW_TITLE => true,
            self::SETTING_WRAP_IN_BOX => true
        );
    }
}
```

Рассмотрим класс более подробно:

1. Все классы виджетов являются потомками класса - **BASE_CLASS_Widget**, который в свою очередь является потомком класса **OW_Component**, который используется как вспомогательный класс для отображения страниц (*Представление и компоненты*).
2. **В конструктор класса передаются экземпляр класса BASE_CLASS_WidgetParameter который со**

   (a) **$customParamList** - ассоциативный массив пользовательских настроек добавленных администратором или обычным пользователем (детально эти настройки их рассмотрим ниже).

   (b) **$additionalParamList** - ассоциативный массив дополнительных параметров, к примеру если виджет расположен на странице профиля то в этот массив будет передан ID просматриваемого пользователя.
3. В методе **onBeforeRender** мы подготавливаем данные которые будут переданы в последствии в файл представления (view).
4. В методе **getAccess** мы выставляем уровень доступа к виджету, т.е кому он будет отображен. Следует отметить, что есть всего три уровня доступа к виджету: **гость - BASE_CLASS_Widget::ACCESS_GUEST**, **пользователь - BASE_CLASS_Widget::ACCESS_MEMBER** и **все - BASE_CLASS_Widget::ACCESS_ALL**.
5. В методе **getSettingList** мы декларативно описываем список пользовательских настроек виджета, к примеру в нашем примере мы даем возможность админу или пользователю вводить какое количество раз вывести текст приветствия в виджете.

6. В методе **getStandardSettingValueList** мы определяем настройки внешнего вида виджета (заголовок, иконка итд.)

Для того, чтобы закончить виджет осталось только создать файл представления (view), для этого создаём пустой **html** файл в директории - **"views/components/example_widget.html"**. Добавляем в него контент отображения:

```
<ul>
    {if $repeatGreeting}
        {for $foo=1 to $repeatGreetingCount}
            <li>{$greetingMesasge} ({$date})</li>
        {/for}
    {else}
        <li>{$greetingMesasge} ({$date})</li>
    {/if}
</ul>
```

В файлах представления используется синтаксис шаблонизатора - Smarty

Активация виджета

Для того, чтобы виджет был зарегистрирован системой, а в последствии запущен нужно написать его инициализацию. Так как виджеты должны работать только из активных плагинов, мы делаем инициализацию только в файле - **activate.php** (*Структура плагина*)

```
<?php

    $widgetService = BOL_ComponentAdminService::getInstance();

    // add example widget on profile page
    $widget = $widgetService->addWidget('MYSUPERPLUGIN_CMP_ExampleWidget', false);
    $widgetPlace = $widgetService->addWidgetToPlace($widget, BOL_ComponentService::PLACE_PROFILE);
    $widgetService->addWidgetToPosition($widgetPlace, BOL_ComponentService::SECTION_LEFT);
```

Данный пример кода добавялет виджет на страницу просмотра профайлов (список доступных системных страниц можно найти в классе - BASE_CLASS_Widget) в левую часть страницы. Если нужно задать конкретную позицию виджета, то ее можно передать третьим параметром в метод **"addWidgetToPosition"**

Деактивация виджета

Как только плагин удаляется или становится не активным, то виджет нужно деактивировать:

```
<?php

    BOL_ComponentAdminService::getInstance()->deleteWidget('MYSUPERPLUGIN_CMP_ExampleWidget');
```

Глава 12

Структура плагина

Oxwall является одной из самых простых MVC структурированных платформ. Он отлично подходит для разработчиков, желающих разобраться в работе веб-фреймворков, написанных на PHP.

Плагины Oxwall разрабатываются, основываясь на принципах паттерна MVC. Составляющие шаблона проектирования - MVC (Model View Controller). Ниже можно более подробно ознакомиться с этими понятиями, а также увидеть структуру плагина.

Model (Модель)

Model позволяет осуществлять непосредственное взаимодействие с базой данных или с другими источниками данных, получая информацию и представляя ее в нужном виде для отображения. Model в основном состоит из запросов к базе данных, а также возможности преобразования данных в необходимый формат. В Oxwall используется технология ORM, но также остается возможность писать прямые запросы (минуя ORM) к базе данных. Подробнее о моделях данных можно прочитать в разделе - "*Слой бизнес логики*".

View (Представление)

View отвечает за отображение информации. В нем должно содержаться как можно меньше логики, которую, по возможности, нужно выносить в Model, Controller или в Component. Это подразумевает простые и понятные шаблоны для вывода информации. Шаблоны в Oxwall имеют расширение ".html".

Компонент - эта некоторая часть инкапсулированной логики из View. Т.е компоненты это некие помощники View. Иногда требуется вынести часть логики которая больше относится к View нежели к Controller, вот для этого и нужен компонент. Также компоненты используются при повторном использовании логики View. К примеру отображение формы или меню. Компоненты могут содержать собственный файл шаблона (.html), а также вызваны через ajax. Подробнее о представляних и об компонентах можно прочитать в разделе - "*Представление и компоненты*".

Controller (Контроллер)

Controller обрабатывает данные от различных клиентов и выводит их с помощью соответствующих файлов View или Component. Подробнее о работе контроллеров можно прочитать в разделе - "*Маршрутизация и контроллеры*"

Базовая структура директорий плагина

/
- - **bol**/
- - **classes**/
- - **components**/
- - **controllers**/
- - **mobile**/
- - - - **classes**/
- - - - **components**/
- - - - **controllers**/
- - - - **views**/
- - - - *init.php*
- - **static**/
- - - - **css**/
- - - - **js**/
- - - - **images**/
- - **views**/
- - - - **components**/
- - - - **controllers**/
- - **update**/
- - *init.php*
- - *cron.php*
- - *activate.php*
- - *deactivate.php*
- - *install.php*
- - *uninstall.php*
- - *langs.zip*
- - *plugin.xml*

Описание директориий плагина:

1. **bol** - содержит файлы модели, которые непосредственно работают с базой данных или с каким-либо другим хранилищем или выполняют любую другую бизнес логику (работа с файлами, картинками итд). Детально как работать с файлами моделей можно прочитать в разделе - *Слой бизнес логики*.
2. **classes** - содержит различные классы плагина не относящиеся к работе с моделью или с контроллером, например классы системных событий или скаченные библиотеки.

3. **controllers** - содержит классы контроллеров, которые непосредственно взаимодействуют с клиентом посредством зарегистрированных роутов. Более подробно про маршрутизацию/роутинг и контроллеры можно прочитать в разделе - *Маршрутизация и контроллеры*.
4. **mobile** - содержит классы контроллеров, моделей и компонентов для работы плагина в мобильном режиме, т.е когда пользователь нажимает на ссылку перехода в мобильную версию сайта или же сайт автоматически определяет устройство пользователя как мобильное.
5. **static** - содержит в себе js, css, а также файлы изображений необходимых для работы плагина. Следует отметить тот факт, что данная директория не доступна из браузера, но после установки плагина все ее содержимое автоматически копируется в директорию доступную для браузеров - **ow_static**.
6. **views** - содержит файлы представления для всех контроллеров и компонентов плагина. Следует отметить тот факт, что файлы представления автоматически запрашиваются для всех методов контроллера или компонента если вы конечно принудительно не завершите выполнения метода (это требуется к примеру в ajax запросах где не всегда нужны файлы представления).
7. **update** - содержит файлы обновлений плагина. Подробнее с процессом обновления плагина можно ознакомиться в разделе - *Обновление плагина*.
8. **init.php** - является входной точкой плагина. Т.е данный файл запускается всегда при любом запросе. Его основной задачей является регистрация маршрутов плагина (*Маршрутизация и контроллеры*), а также какой-либо дополнительной функциональности, к примеру регистрация или реагирование на какие-либо системные события.
9. **cron.php** - выполняет фоновые задачи плагина с помощью планировщика задач CRON. Подробнее о планировщике можно прочесть в разделе - *Cron (Планировщик задач)*. Следует отметить, что данный файл не обязательно должен присутствовать в плагине. Если вы не используете CRON вы можете просто удалить его из плагина. Если файл все-таки есть в плагине то он будет автоматически подключен и использован системой.
10. **activate.php** - в данный файл можно вынести логику плагина которая будет запущена когда администратор активирует плагин в админ панели или установит его. К примеру после активации плагина вы можете зарегистрировать виджеты (подробнее в разделе - *Виджеты*) на страницах сайта или добавить новые пункты меню. Данный файл опционален, т.е может не присутствовать в плагине если в нем нет нужды. Если файл все-таки есть в плагине то он будет автоматически подключен и использован системой.
11. **deactivate.php** - в данный файл можно вынести логику плагина которая будет запущена когда администратор деактивирует плагин или деинсталлирует его в админ панели. К примеру после деактивации плагина вы можете удалить ранее добавленые виджеты плагина со страниц сайта. Данный файл опционален, т.е может не присутствовать в плагине если в нем нет нужды. Если файл все-таки есть в плагине то он будет автоматически подключен и использован системой.
12. **install.php** - данный файл запускается только при установке плагина, его можно использовать для того, чтобы выполнить SQL запросы создающие таблицы в базе данных, для импорта файла переводов (подробнее в разделе - *Файлы переводов*), регистрации настроек плагина, регистрации групп и действий авторизации (подробнее в разделе - *Авторизация*). Данный файл опционален, т.е может не присутствовать в плагине если в нем нет нужды. Если файл все-таки есть в плагине то он будет автоматически подключен и использован системой.
13. **uninstall.php** - данный файл запускается только при деинсталяции плагина, его можно использовать для того, чтобы удалить ранее созданные таблицы в базе данных и.т.д. Следует отметить, что большинство ресурсов такие как: файлы переводов, группы и действия авторизации, настройки плагина удаляются автоматически и нет нужды их в ручную удалять в этом файле. Данный файл опционален, т.е может не присутствовать в плагине если в нем нет нужды. Если файл все-таки есть в плагине то он будет автоматически подключен и использован системой.

14. **plugin.xml** - содержит описание плагина и является обязательным, ниже мы покажем и расскажем про структуру данного файла.

Описание файла plugin.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<plugin>
    <name>My Super Plugin</name>
    <key>superplugin</key>
    <description>My super plugin.</description>
    <author>Me</author>
    <authorEmail>me@oxwall.org</authorEmail>
    <authorUrl>http://www.me.com</authorUrl>
    <developerKey>MY_DEV_KEY</developerKey>
    <build>1</build>
    <copyright>(C) 2015 My. All rights reserved.</copyright>
    <license>OSCL</license>
    <licenseUrl>http://www.oxwall.org/store/oscl</licenseUrl>
</plugin>
```

1. В секции **name** пишем название плагина.
2. В секции **key** пишем название плагина без пробелов в нижнем регистре используя только символы латиницы, а также цифры [a-z0-9]. Также нужно удостовериться, что плагина с таким ключом не существует, сделать это можно здесь.
3. В секции **description** пишем краткое описание плагина, что он делает или умеет.
4. В секции **author** пишем имя разработчика плагина.
5. В секции **authorEmail** пишем email разработчика плагина.
6. В секции **developerKey** пишем ключ разработчика плагина. Данный ключ будет использоваться для получения обновлений плагина. Подробно как получить ключ разработчика и начать продавать собственные плагины описано в разделе - *Ключи разработчика и плагина*.
7. В секции **build** пишем номер билда плагина.
8. В секции **copyright** пишем информацию об авторских правах на плагин.
9. В секции **license** пишем о типе используемой лицензии плагина.
10. В секции **licenseUrl** указываем url где мы можем познакомиться подробнее с типом выбранной лицензией плагина.

Глава 13

Обновление плагина

После того как плагин написан и выложен в магазине Oxwall разработчикам зачастую требуется постоянный механизм позволяющий просто и быстро вносить свои изменения в плагины и распространять эти изменения среди своих клиентов. Давайте посмотрим как это можно просто сделать на платформе Oxwall.

Исправляем ошибки в плагине

Наша задача исправить вымышленную ошибку в коде нашего плагина. Представим, что мы уже написали наш первый плагин и его **plugin.xml** описывающий мета информацию плагина выглядит так: (подробнее про структуру файлов плагина можно прочитать в разделе - *Структура плагина*) :

```
<?xml version="1.0" encoding="utf-8"?>
<plugin>
    <name>My Super Plugin</name>
    <key>superplugin</key>
    <description>My super plugin.</description>
    <author>Me</author>
    <authorEmail>me@oxwall.org</authorEmail>
    <authorUrl>http://www.me.com</authorUrl>
    <developerKey>MY_DEV_KEY</developerKey>
    <build>1</build>
    <copyright>(C) 2015 My. All rights reserved.</copyright>
    <license>OSCL</license>
    <licenseUrl>http://www.oxwall.org/store/oscl</licenseUrl>
</plugin>
```

Мы нашли и исправили ошибку и теперь просто повышаем номер билда в нашем **plugin.xml** с **1** до **2**. И наш результирующий файл стал выглядеть так:

```
<?xml version="1.0" encoding="utf-8"?>
<plugin>
    <name>My Super Plugin</name>
```

```
    <key>superplugin</key>
    <description>My super plugin.</description>
    <author>Me</author>
    <authorEmail>me@oxwall.org</authorEmail>
    <authorUrl>http://www.me.com</authorUrl>
    <developerKey>MY_DEV_KEY</developerKey>
    <build>2</build>
    <copyright>(C) 2015 My. All rights reserved.</copyright>
    <license>OSCL</license>
    <licenseUrl>http://www.oxwall.org/store/oscl</licenseUrl>
</plugin>
```

Теперь можно запаковать наш плагин с обновленным файлом **plugin.xml**, а также с нашими исправлениями в коде и загрузить архив в магазин Oxwall (не забудьте сделать пакет с исправлениями основным в настройках загруженного плагина). Система автоматически оповестит всех клиентов о наличии обновлений и клиентам останется только нажать на ссылку обновить плагин в админ панели своего сайта. Но бывает ситуации когда не достаточно внести изменения только в файлы кода, но также необходимо внести изменения в структуру базы данных, добавить/удалить виджет или просто выполнить произвольный **php** код, для этого мы можем написать скрипт обновления, о чем и пойдет речь ниже.

Скрипты обновления

Что такое скрипт обновления? - это просто произвольный **php** скрипт который завязан на определенный билд плагина, т.е если номер билда у нас **2**, то ему возможно будет соответствовать скрипт обновления с таким же номером, но не всегда, скрипты обновлений является опциональными. И так представим, что нам нужно внести изменения в базу данных в таблицы нашего плагина, а также добавить новые переводы. Для этого нам нужно создать в директории **update** нашего плагина директорию с названием **2** которая соответствует нашему последнему билду с исправлениями (выше в разделе - *Обновление плагина* мы исправляли вымышленную ошибку) и так структура директорий нашего плагина должны выглядеть примерно так:

/
- - **bol**/
- - **classes**/
- - **components**/
- - **controllers**/
- - **mobile**/
- - - - **classes**/
- - - - **components**/
- - - - **controllers**/
- - - - **views**/
- - - - *init.php*
- - **static**/
- - - - **css**/
- - - - **js**/
- - - - **images**/
- - **views**/
- - - - **components**/

- - - - **controllers**/
- - **update**/
- - - - **2**/ **(наш скрипт обновлений)**
- - *init.php*
- - *cron.php*
- - *activate.php*
- - *deactivate.php*
- - *install.php*
- - *uninstall.php*
- - *langs.zip*
- - *plugin.xml*

Внутрь созданной директории помещаем наш скрипт обновления - **update.php** (название скрипта менять не нужно). Ниже приведу пример содержимого этого файла:

```
<?php

// импорт новых переводов плагина
Updater::getLanguageService()->importPrefixFromZip(dirname(__FILE__) . DS . 'langs.zip',
↪'superplugin') ;

// добавляем новый индекс в таблицу
$sql = "ALTER TABLE `".OW_DB_PREFIX."my_table` ADD UNIQUE `userId` (`userId`)";
Updater::getDbo()->query($sql);
```

Скрипт обновления создан, теперь мы можем запаковать весь плагин и выложить его в магазине Oxwall. С основными сервисами которые могут вам помогут эффективно работать при написании плагинов или обновлений можно познакомиться в разделе - *Основные сервисы приложения*

Глава 14

Ключи разработчика и плагина

Для того, чтобы начать разработку собственных плагинов или тем вам нужен ключ разработчика, а также ключ плагина или темы.

Что такое ключ разработчика?

Если вы планируете развивать свой бизнес вместе с **Oxwall** то вам необходимо получить уникальный идентификационный номер разработчика, который нужно будет включать во все свои плагины или темы в файл **plugin.xml** или **theme.xml**.

Ключ разработчика идентифицирует разработчиков плагинов или тем позволяя им сохранять авторство. Это один из наиболее важных параметров в файле **plugin.xml** (Подробнее про структуру плагина можно прочитать в разделе *Структура плагина*) и **theme.xml**.

Для получения ключа разработчика вам нужно зарегистрировать новый аккаунт на oxwall.org или использовать уже существующий аккаунт если вы регистрировались ранее. После регистрации нужно перейти на страницу - Dev-tools и оттуда скопировать уже сгенерированный для вас ключ.

В примере ниже можно видеть куда прописывать полученный вами ключ разработчика в файл **plugin.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<plugin>
    <name>My Super Plugin</name>
    <key>superplugin</key>
    <description>My super plugin.</description>
    <author>Me</author>
    <authorEmail>me@oxwall.org</authorEmail>
    <authorUrl>http://www.me.com</authorUrl>
    <developerKey>------->MY_UNIQUE_DEV_KEY<-------</developerKey>
    <build>1</build>
    <copyright>(C) 2015 My. All rights reserved.</copyright>
    <license>OSCL</license>
```

```
    <licenseUrl>http://www.oxwall.org/store/oscl</licenseUrl>
</plugin>
```

Что такое ключ плагина/темы?

Ключ плагина или темы - это просто уникальное название. Перед тем как начать разрабатывать собственный плагин или тему вам нужно удостовериться, что выбранный вами ключ плагина или темы не используется другими плагинами или темами, проверка уникальности ключа осуществляеться на странице - проверки ключа. **Стоит отметить, что ключ плагина и темы должен быть прописан в нижнем регистре и не содержать никаких символов кроме латинских, а также цифр (a-z0-9).**

Ключ разработчика в купе с ключом плагина или темы идентифицирует ваш плагин среди других плагинов или тем, а также они используется во всей внутренней экосистеме Oxwall. К примеру панель администратора именно по этим ключам в плагинах или в темах может определить наличие новых обновлений или наличие прав на их использование.

В примере ниже можно видеть куда прописывать полученный вами ключ плагина в файле **plugin.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<plugin>
    <name>My Super Plugin</name>
    <key>------->superplugin<-------</key>
    <description>My super plugin.</description>
    <author>Me</author>
    <authorEmail>me@oxwall.org</authorEmail>
    <authorUrl>http://www.me.com</authorUrl>
    <developerKey>MY_UNIQUE_DEV_KEY</developerKey>
    <build>1</build>
    <copyright>(C) 2015 My. All rights reserved.</copyright>
    <license>OSCL</license>
    <licenseUrl>http://www.oxwall.org/store/oscl</licenseUrl>
</plugin>
```

Глава 15

Стандарты кодирования

Если вы решили писать код для платформы **Oxwall**, то вам желательно стараться соблюдать все стандарты кодирования описанные ниже.

Общие

Всегда используйте полную форму **PHP** тега:

```
<?php
?>
```

Для файлов, содержащих только PHP код, закрывающий тег - ?> должен быть опущен.

Используя операторы **include_once** и **require_once** не нужно использовать скобки.

```
<?php
    //RIGHT
    require_once 'header.php';

    //WRONG
    require_once('header.php');
?>
```

Отступы

Всегда используйте **4 пробела** для отступов вместо символов таба.

Соглашения об именовании

Названия файлов

```
view.php
base_dao.php
my_super_class.php
```

Классы

```
class MySuperClass
{
    //code here
}

class PREFIX_MySuperClass
{
    //code here
}
```

Функции

```
function connect()
function camelCaseFunction()
function fooBar()
my_global_function()
```

Переменные

```
public    $myVar;
private   $hisVar;
protected $x;
```

Константы

```
<?php

    define("MY_SUPER_CONSTANT", "Hello world");
```

Управляющие структуры

Нужно использовать пробелы внутри скобок во всех управляющих структурах (foreach, for, while, if, switch, try, catch, итд).

Используйте **фигурные скобки**, даже в случае, если они не являются обязательными, поскольку это сделает код более удобным для чтения и поможет избежать логических ошибок, возникающих при добавлении новых строк кода.

switch

```php
<?php

    switch ( condition )
    {
        case 1:
            action1();
            break;

        case 2:
            action2();
            break;

        default:
            defaultAction();
            break;
    }
```

if, else

Используйте оператор **else if** вместо **elseif**

```php
<?php

    if ( $a !== $b )
    {
        return false;
    }
    else if ( false )
    {
        doSomething1();
    }
    else
    {
        doSomething2();
    }
```

Разделяете длинное условие на несколько строк

```php
<?php

    if ( condition1
        || condition2
        && condition3 )
    {
        //code here
    }
```

foreach, for, while

```php
<?php

    foreach ( $a as $v )
```

```
    {
        echo $v;
    }
```

try, catch

```
<?php

    try
    {
        //code here
    }
    catch ( Exception $e )
    {
        //code here
    }
```

Объявление функций

Все имена функций должны быть в стиле - **camelCase**. Глобальные функции являются исключением, они должны состоять из слов в нижнем регистре и символов подчеркивания (**my_global_function**). В функциях не должно быть пробелов между именем функции и открывающей скобкой, а также перед возвратом значения должен быть символ новой строки.

```
<?php

    function fooBar( $param1, $param2 )
    {
        if ( $param1 !=== $param2 )
        {
            //code here
        }

        return true;
    }
```

Вы всегда должны объявлять типы входных параметров в функциях, когда это возможно:

```
<?php

    function doSomethingGood( MyClass $obj )
    {
        //code here
    }
```

Пример глобальной функции:

```
<?php

    function print_var( $var, $echo = false )
    {
        //code here
    }
```

Вызов функций

```php
<?php

    myCoolFunction(1, 2, 3);
    $this->myCoolMethod(1, 2, 3);
```

Массивы

Индексированные массивы

```php
<?php

    $arr = array ( 1, 2, 'no', 'pain', 'no', 'gain' );
    $longArr = array ( 1, 2, 3,
                       4, 5, 6,
                       7, 8, 9 );
```

Ассоциативные массивы

```php
<?php

    $assoc = array ( 'key1' => 'value1',
                     'key2' => 'value2',
                     'key3' => 'value3' );
```

Глава 16

Инструменты отладки

Использование инструментов отладки позволяют вам измерять скорость запросов **MySQL**, выводить ошибки и предупреждения в коде, а также видеть сделанные изменения сразу после обновления страницы. Все эти инструменты активируются и деактивируются в конфигурационном файле - **ow_includes/config.php**. Стоит обратить внимание, что все перечисленные ниже инструменты должны быть отключены на рабочем сайте и использоваться только в режиме разработки.

Профайлер

Если вам необходимо видеть список запросов сделанных к **БД** на странице или посмотреть список сработанных системных событий или же просто узнать на сколько быстро загружается страница - то нужно активировать профайлер в конфигурационном файле. Найдите константу **OW_PROFILER_ENABLE** и присвойте ей значение **true**. Теперь на сайте вам будет доступна информация по всем ранее перечисленным данным, стоит отметить, что вывод данных профайлера реализован пока только на страницах сайта (для режима CLI это пока не реализовано).

Режим разработки

Режим разработки задается через битовую маску в константе **OW_DEV_MODE** в конфигурационном файле. Этот режим влияет на то как будет собираться приложение каждый раз при обновлении страницы, ниже пример сборок:

1. **define('OW_DEBUG_MODE', 2)** - Чистит кэш смарти шаблонов.
2. **define('OW_DEBUG_MODE', 4)** - Пересборка тем (копирование **css** в публичную директорию из **ow_themes** в **ow_static** доступную из браузера).
3. **define('OW_DEBUG_MODE', 8)** - Очистка кеша файлов переводов.
4. **define('OW_DEBUG_MODE', 32)** - Копирование файлов статики плагинов в публичную директорию **ow_static/plugins/plugin_key** доступную из браузера.

Можно комбинировать типы сборок, к примеру можно задать битовую маску для очистки кэша смарти шаблонов и очистки кеша файлов переводов. Для этого нужно задать: **define('OW_DEBUG_MODE', 10);**

Если просто указать **define('OW_DEBUG_MODE', 1);** или **define('OW_DEBUG_MODE', true);** то приложение выполнит все виды сборок перечисленные выше.

Режим отладки

Для того, чтобы видеть ошибки выполнения сценариев на сайте, нужно переменной **OW_DEBUG_MODE** выставить значение **true** в конфигурационном файле.