Como Criar um Plugin para o osTicket

Introdução ao Desenvolvimento de Plugins

O desenvolvimento de plugins para o osTicket representa um desafio significativo, principalmente devido à falta de documentação oficial. Este guia baseia-se em informações obtidas a partir de um blog não oficial, que descreve como criar um plugin no osTicket.

Neste guia, focamos na construção de um plugin compatível com a versão 1.18 do osTicket, enquanto o guia original foi elaborado para a versão 1.9.3. Abordaremos os desafios resultantes dessas diferenças de versão e forneceremos soluções adequadas ao longo do processo.

Para que um plugin seja reconhecido pelo osTicket, ele deve estar numa pasta que contenha um ficheiro `plugin.php`, um ficheiro de configuração `Config.php` e um ficheiro principal com a lógica do plugin. Esta pasta deve ser colocada na diretoria `[INSTALL_ROOT]/include/plugins` do OsTicket.

```
[INSTALL_ROOT]
  └── include
      └── plugins
          └── nome_do_plugin
              ├── plugin.php
              ├── Config.php
              └── PluginExemplo.php
```

Passo 1: Criar o Ficheiro `plugin.php`

O primeiro passo será criar um ficheiro `plugin.php` que contém a descrição e os dados essenciais do plugin. Abaixo, fornecemos um exemplo prático de como este ficheiro deve ser estruturado:

```php
<?php
return array(
    'id' => 'rodrigo:PluginExemplo',
    'version' => '1.0',
    'name' => 'Plugin Exemplo',
    'author' => 'Rodrigo',
    'description' => 'Plugin exemplo que faz alguma coisa.',
    'url' => 'https://github.com/poctob/OSTEquipmentPlugin/wiki/Plugin-
Development-Introduction',
    'plugin' => 'PluginExemplo.php:PluginExemplo'
);
?>
```

Elementos do Array

- **id**: Identificador único para o plugin, ex. 'rodrigo:PluginExemplo'.

- **version**: Versão do plugin, ex. '1.0'.
- **name**: Nome do plugin exibido na interface, ex. 'Plugin Exemplo'.
- **author**: Nome do(s) autor(es), ex. 'Rodrigo'.
- **description**: Breve descrição do plugin.
- **url**: URL para o helpdesk se aplicavél.
- **plugin**: Ficheiro principal com a lógica do plugin e a classe a ser instanciada, ex. 'PluginExemplo.php:PluginExemplo'.

Passo 2. Página de Configuração do Plugin

Como mencionado anteriormente, o plugin necessita ter uma classe de configuração para ser reconhecido pelo osTicket. A seguir, apresentamos um exemplo prático da estrutura que esta classe deve adotar:

Exemplo de Ficheiro de Configuração `Config.php`

```
class PluginExemploConfig extends PluginConfig {
    function getOptions() {
        return array(
            'plugin_exemplo_backup' => new BooleanField([
                'id' => 'plugin_exemplo_backup',
                'label' => 'Fazer Backup',
                'configuration' => array(
                    'desc' => 'Ativar ou desativar backups automáticos.'
                )
            ]),
        );
    }

    function pre_save(&$config, &$errors) {
        global $msg;

        // Guardar a configuração antes de salvar
        if (!empty($this->get('plugin_exemplo_backup', ''))) {
            $this->set('plugin_exemplo_backup', $config['plugin_exemplo_backup']);
        }

        if (!$errors)
            $msg = 'Configuration updated successfully';

        return true;
    }

    function post_save(&$config, &$errors) {
        global $msg;

        // Lógica post_save, se necessário
        if (isset($config['plugin_exemplo_backup'])) {
            $msg = 'Configuração post_save realizada com sucesso';
        }

        return true;
```

```
    }
}
```

Detalhes da Configuração

A classe `PluginExemploConfig` deriva de `PluginConfig`, que se encontra localizada em `[INSTALL_ROOT]/include/class.plugin.php`. A função `getOptions()` é projetada para retornar um array com as opções de configuração do plugin.

A opção de configuração `plugin_exemplo_backup` é representada por uma select box com o ID `plugin_exemplo_backup` e o label "Fazer Backup". Esta select box inclui uma descrição que refere a sua funcionalidade "Ativar ou desativar backups automáticos". Os dados configurados são armazenados na tabela `ost_config` na base de dados, designada pelo número do plugin. Cada nova instalação de um plugin aumenta este número, criando namespaces como `plugin.3` ou `plugin.44`. Este sistema permite a instalação e gestão independente de múltiplos plugins, cada um acedendo as suas configurações específicas através do método `$this->getConfig()->get('setting');`, garantindo a unicidade de cada configuração. Os dados são armazenados de forma escalar ou em formato JSON no campo de valor da tabela.

Tipos de Campos Disponíveis

Os campos disponíveis para configuração, definidos em `[INSTALL_ROOT]/include/class.forms.php`, incluem:

- `TextboxField` - Caixa de texto simples.
- `TextareaField` - Área de texto para conteúdos mais extensos.
- `ThreadEntryField` - Área de texto, normalmente utilizada para discussões em fóruns.
- `DatetimeField` - Select de datas baseado em JQuery.
- `PhoneField` - Caixa de texto otimizada para a inserção de números de telefone.
- `BooleanField` - select box para opções binárias de verdadeiro/falso.
- `ChoiceField` - Campo dropdown para seleção entre várias opções.
- `SectionBreakField` - Quebra de secção horizontal, que auxilia na organização visual das configurações.

Função `pre_save(&$config, &$errors)`

Esta função é ativada quando as configurações são submetidas pelo utilizador. Realiza a validação do lado do servidor e implementa a lógica necessária para o processamento das configurações. Para indicar uma falha durante as verificações, um erro é adicionado ao array de erros com `$errors['err'] = "mensagem";`, e a função retorna `false` em `pre_save`, impedindo a atualização das configurações.

Função `post_save(&$config, &$errors)`

Esta função é chamada após as configurações terem sido guardadas com sucesso. Aqui podemos adicionar qualquer lógica que precise de ser executada após a atualização das configurações. Por exemplo, enviar notificações ou atualizar caches.

Passo 3. Classe Principal do Plugin

Estrutura do Plugin

Para que um plugin seja válido no osTicket, a classe do plugin deve incluir apenas um método obrigatório: o método `bootstrap()`. Os demais métodos são opcionais e servem para ampliar a funcionalidade do plugin.

Tendo isso em mente, o próximo passo é criar um ficheiro denominado `PluginExemplo.php` para definir a classe principal do plugin, que deve conter toda a lógica necessária. Esta classe deve estender `Plugin` e incluir os métodos de inicialização e configuração do plugin. No exemplo abaixo, alguns métodos não estão completamente implementados, pois o objetivo principal é demonstrar a estrutura do plugin e não a lógica completa de funcionalidades específicas.

```php
<?php
require_once(INCLUDE_DIR . 'class.plugin.php');
require_once(INCLUDE_DIR . 'class.signal.php');
require_once(INCLUDE_DIR . 'class.osticket.php');
require_once(__DIR__ . '/config.php');

define('PLUGIN_DIR', __DIR__ . '/');
define('EXAMPLE_TABLE', TABLE_PREFIX . 'example_table');
define('SESSION_PREVIOUS_STATE_PLUGIN_PREFIX', 'previousActiveStatePlugin_');

class PluginExemplo extends Plugin {
    public const PLUGIN_NAME = 'PluginExemplo';
    var $config_class = 'PluginExemploConfig';

    /**
     * Inicializa o plugin.
     *
     * @return void
     */
    function bootstrap() {
        // Conectar ao sinal de criação de tickets
        Signal::connect('model.created', array($this, 'onTicketCreated'),
'Ticket');

        if ($this->firstRun() && !$this->configureFirstRun()) {
            error_log('Falha ao configurar a primeira execução do plugin.');
        }

        if ($this->needUpgrade()) {
            $this->configureUpgrade();
        }

        // Realiza as ações específicas da instância
        $this->makeInstanceSpecificActions();
    }

    /**
     * Realiza as ações específicas da instância.
     *
     * @return void
     */
    private function makeInstanceSpecificActions() {
        $config = $this->getConfig();
```

```
        if ($config->get('plugin_exemplo_backup') && parent::isActive()) {
            $this->realizarBackup();
        }
    }

    /**
     * Função chamada quando um ticket é criado.
     *
     * @param Ticket $ticket O ticket recém-criado.
     * @return void
     */
    function onTicketCreated($ticket) {
        error_log('Novo ticket criado: ID ' . $ticket->getId());
        // Adicionar lógica adicional a ser executada quando um ticket é criado
    }

    /**
     * Verifica se o plugin está ativo globalmente.
     *
     * @return bool True se ativo, false caso contrário.
     */
    function isActive() {
        $currentlyActive = parent::isActive();
        $previouslyActive = $this->getCurrentActiveStatePlugin();

        if ($previouslyActive !== $currentlyActive) {
            $this->handleActivationChange($currentlyActive);
            $this->setCurrentActiveStatePlugin($currentlyActive);
        }

        return $currentlyActive;
    }

    /**
     * Manipula a mudança de ativação do plugin.
     *
     * @param bool $currentlyActive O estado atual do plugin.
     * @return void
     */
    private function handleActivationChange($currentlyActive) {
        error_log("Plugin está a ser " . ($currentlyActive ? "ativado" :
"desativado") . ".");
        if ($currentlyActive) {
            $this->enablePlugin();
        } else {
            $this->disablePlugin();
        }
    }

    /**
     * Lógica a ser executada quando o plugin é ativado.
     *
     * @return void
     */
```

```
    function enablePlugin() {
        if ($this->createDBTables()) {
            error_log("Plugin Exemplo ativado com sucesso!");
        } else {
            error_log("Erro ao ativar o Plugin Exemplo.");
        }
    }

    /**
     * Lógica a ser executada quando o plugin é desativado.
     *
     * @return void
     */
    function disablePlugin() {
        if ($this->cleanUp()) {
            error_log("Plugin Exemplo desativado com sucesso!");
        } else {
            error_log("Erro ao desativar o Plugin Exemplo.");
        }
    }

    /**
     * Realiza o backup conforme a configuração do plugin.
     *
     * @return void
     */
    function realizarBackup() {
        error_log("Backup realizado com sucesso pelo Plugin Exemplo!");
        // Adicionar lógica de backup
    }

    /**
     * Verifica se esta é a primeira execução do plugin.
     *
     * @return bool True se for a primeira execução, false caso contrário.
     */
    function firstRun() {
        $initialized = !$this->getConfig()->get('plugin_exemplo_initialized',
false);
        error_log("firstRun retornou " . ($initialized ? "true" : "false") . ".");
        return $initialized;
    }

    /**
     * Verifica se o plugin precisa ser atualizado.
     *
     * @return bool True se precisar de atualização, false caso contrário.
     */
    function needUpgrade() {
        return false; // Lógica para determinar se o plugin precisa ser atualizado
    }

    /**
     * Configurações necessárias para a primeira execução do plugin.
```

```
     *
     * @return bool True se a configuração inicial foi bem-sucedida, false caso
contrário.
     */
    function configureFirstRun() {
        $this->getConfig()->set('plugin_exemplo_initialized', true);
        return true;
    }

    /**
     * Lida com a lógica de atualização do plugin.
     *
     * @return void
     */
    function configureUpgrade() {
        // Lógica de atualização do plugin
    }

    /**
     * Cria tabelas na base de dados.
     *
     * @return bool True se a criação das tabelas foi bem-sucedida, false caso
contrário.
     */
    function createDBTables() {
        $sql = "CREATE TABLE IF NOT EXISTS " . EXAMPLE_TABLE . " (
            id INT(11) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
            name VARCHAR(255) NOT NULL,
            description TEXT,
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;";

        if (db_query($sql)) {
            error_log("Tabela " . EXAMPLE_TABLE . " criada com sucesso.");
            return true;
        } else {
            error_log("Falha ao criar tabela " . EXAMPLE_TABLE . ".");
            return false;
        }
    }

    /**
     * Função de limpeza ao desativar o plugin.
     *
     * @return bool True se a remoção das tabelas foi bem sucedida, false caso
contrário.
     */
    function cleanUp() {
        $sql = "DROP TABLE IF EXISTS `" . EXAMPLE_TABLE . "`;";

        if (db_query($sql)) {
            error_log("Tabela " . EXAMPLE_TABLE . " removida com sucesso.");
            return true;
        } else {
```

```
                error_log("Erro ao remover a tabela " . EXAMPLE_TABLE . ".");
                return false;
            }
        }

        /**
         * Define o estado ativo atual na sessão para o plugin.
         *
         * @param bool $state O estado ativo atual.
         */
        private function setCurrentActiveStatePlugin($state) {
            $_SESSION[SESSION_PREVIOUS_STATE_PLUGIN_PREFIX] = $state;
        }

        /**
         * Recupera o estado ativo atual do plugin.
         *
         * @return bool|null O estado ativo atual ou null se não estiver definido.
         */
        private function getCurrentActiveStatePlugin() {
            return $_SESSION[SESSION_PREVIOUS_STATE_PLUGIN_PREFIX] ?? null;
        }
    }
```

Resumo da Classe do Plugin

O código listado acima compõe um plugin de demonstração para o osTicket, projetado para mostrar a estrutura básica e algumas funcionalidades. As principais funcionalidades incluem:

- **Resposta a sinais do sistema**: O plugin reage a eventos gerados pelo osTicket, permitindo a execução de ações específicas, como quando um novo ticket é criado.
- **Gestão de estados do plugin**: Permite saber se é necessário remover ou manter as alterações feitas pelo plugin, dependendo do seu estado atual (ativado ou desativado).
- **Criação e remoção de tabelas na base de dados**: Demonstra como o plugin interage com a base de dados para criar e remover tabelas, facilitando o armazenamento e a gestão de dados específicos.
- **Configuração específica para cada instância**: Suporta múltiplas instâncias do plugin, permitindo configurações e estados distintos para cada uma delas. Cada instância pode realizar ações específicas baseadas nas suas configurações.
- **Primeira execução (first run)**: Verifica se é a primeira vez que o plugin está a ser executado e realiza as configurações iniciais necessárias.
- **Atualizações (upgrades)**: Verifica se o plugin precisa ser atualizado e aplica as configurações de atualização quando necessário.

Classe `PluginExemplo.php` em detalhe

Configuração

A classe `PluginExemplo` estende `Plugin` e utiliza a classe de configuração `PluginExemploConfig`. Isso permite que o plugin utilize as opções de configuração definidas na classe `PluginExemploConfig`, como a opção para ativar ou desativar backups automáticos.

```
var $config_class = 'PluginExemploConfig';
```

Inicialização

A função `bootstrap()` é chamada pelo osTicket para inicializar o plugin. É chamado sempre que um utilizador acede ao OsTicket. Esta função:

- Conecta-se ao sinal de criação de tickets (model.created) para executar uma função (onTicketCreated) sempre que um novo ticket é criado.

```
Signal::connect('model.created', array($this, 'onTicketCreated'), 'Ticket');
```

- Verifica se é a primeira vez que o plugin está a ser executado através da função `firstRun()`. Se for a primeira execução, chama `configureFirstRun()` para realizar as configurações iniciais.

```
if ($this->firstRun() && !$this->configureFirstRun()) {
    error_log('Falha ao configurar a primeira execução do plugin.');
}
```

- Verifica se o plugin precisa ser atualizado através da função `needUpgrade()`. Se precisar, chama `configureUpgrade()`.

```
if ($this->needUpgrade()) {
    $this->configureUpgrade();
}
```

- Executa ações específicas da instância através da função makeInstanceSpecificActions(), como verificar a configuração de backup e realizar o backup se necessário.

```
$this->makeInstanceSpecificActions();
```

Funcionalidades

A função `realizarBackup()` implementa as funcionalidades de backup do plugin. Esta função é chamada se a configuração `plugin_exemplo_backup` estiver ativada e o plugin estiver ativo.

```
function realizarBackup() {
    error_log("Backup realizado com sucesso pelo Plugin Exemplo!");
    // Adicionar lógica de backup
}
```

Ativação

A função `isActive()` verifica se o plugin está ativo. Esta função também lida com a lógica de ativação e desativação do plugin:

- Obtém o estado atual do plugin (`currentlyActive`) e o estado anterior (`previouslyActive`).

```
$currentlyActive = parent::isActive();
$previouslyActive = $this->getCurrentActiveStatePlugin();
```

- Se o estado mudou, chama handleActivationChange() para lidar com a ativação ou desativação do plugin.

```
if ($previouslyActive !== $currentlyActive) {
    $this->handleActivationChange($currentlyActive);
    $this->setCurrentActiveStatePlugin($currentlyActive);
}
```

- Atualiza o estado atual do plugin na sessão através das funções `setCurrentActiveStatePlugin` e `getCurrentActiveStatePlugin`.

```
private function setCurrentActiveStatePlugin($state) {
    $_SESSION[SESSION_PREVIOUS_STATE_PLUGIN_PREFIX] = $state;
}

private function getCurrentActiveStatePlugin() {
    return $_SESSION[SESSION_PREVIOUS_STATE_PLUGIN_PREFIX] ?? null;
}
```

Primeira Execução

A função `firstRun()` verifica se é a primeira vez que o plugin está a ser executado, verificando a configuração `plugin_exemplo_initialized`.

```
function firstRun() {
    $initialized = !$this->getConfig()->get('plugin_exemplo_initialized', false);
    error_log("firstRun retornou " . ($initialized ? "true" : "false") . ".");
```

```
    return $initialized;
}
```

Atualização do Plugin

A função needUpgrade() verifica se o plugin precisa ser atualizado. No exemplo fornecido, esta função retorna sempre false, mas pode ser modificada para incluir lógica para verificação de versões.

```
function needUpgrade() {
    return false; // Lógica para determinar se o plugin precisa ser atualizado
}
```

Configuração Inicial

A função `configureFirstRun()` realiza as configurações necessárias para a primeira execução, como marcar o plugin como inicializado.

```
function configureFirstRun() {
    $this->getConfig()->set('plugin_exemplo_initialized', true);
    return true;
}
```

Atualização do Plugin

A função `configureUpgrade()` lida com a lógica de atualização do plugin. Esta função pode ser expandida para incluir scripts para por exemplo atualizar a base de dados.

```
function configureUpgrade() {
    // Lógica de atualização do plugin
}
```

Criação de Tabelas na Base de Dados

A função `createDBTables()` realiza a criação das tabelas na base de dados. Neste exemplo, é criada uma tabela chamada `example_table` com campos para `id`, `name`, `description` e `created_at`.

```
function createDBTables() {
    $sql = "CREATE TABLE IF NOT EXISTS " . EXAMPLE_TABLE . " (
        id INT(11) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
        name VARCHAR(255) NOT NULL,
        description TEXT,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;";
```

```
    if (db_query($sql)) {
        error_log("Tabela " . EXAMPLE_TABLE . " criada com sucesso.");
        return true;
    } else {
        error_log("Falha ao criar tabela " . EXAMPLE_TABLE . ".");
        return false;
    }
}
```

Limpeza ao Desativar o Plugin

A função `cleanUp()` lida com a remoção das tabelas e a limpeza dos dados quando o plugin é desativado. Ela executa uma consulta SQL para remover a tabela EXAMPLE_TABLE se ela existir.

```
function cleanUp() {
    $sql = "DROP TABLE IF EXISTS `" . EXAMPLE_TABLE . "`;";

    if (db_query($sql)) {
        error_log("Tabela " . EXAMPLE_TABLE . " removida com sucesso.");
        return true;
    } else {
        error_log("Erro ao remover a tabela " . EXAMPLE_TABLE . ".");
        return false;
    }
}
```

Nota sobre a Função `pre_uninstall`

Nas versões mais antigas do osTicket, como a 1.9.3, era possível utilizar uma função chamada `pre_uninstall` para realizar ações específicas antes de desinstalar o plugin. No entanto, essa função não é mais suportada na versão 1.18 do osTicket. Portanto, caso esteja a utilizar a versão 1.18 ou superior, uma alternativa é implementar lógicas para desativar e ativar o plugin. Portanto, sempre que quiser desinstalar o plugin, é necessário desativá-lo primeiro.

Sinais

O osTicket utiliza um sistema de sinais (`signals`) para notificar os plugins sobre eventos específicos, permitindo que os programadores estendam a funcionalidade do sistema de forma modular. Este sistema permite que os plugins inscrevam-se para ouvir eventos específicos e respondam quando esses eventos ocorrem.

Funcionamento dos Sinais

A classe de roteamento de sinais do osTicket está definida em `[INSTALL_ROOT]/include/class.signal.php`. Esta classe oferece duas funções estáticas principais:

- `connect`: Permite que um plugin se inscreva para ouvir um sinal específico. Quando o sinal é emitido, todas as funções inscritas para esse sinal serão chamadas.
- `send`: Permite que um sinal seja emitido. Quando um sinal é emitido, todas as funções que se inscreveram para ouvir esse sinal serão notificadas e executadas.

A utilização dos sinais é bastante direta. Primeiro, inscreve-se para ouvir um sinal utilizando a função `connect`. Em seguida, em algum ponto do código, pode emitir esse sinal utilizando a função `send`.

Exemplo de Utilização

No exemplo fornecido do `PluginExemplo`, o plugin conecta-se ao sinal `model.created`, que é emitido sempre que um novo ticket é criado. Neste caso, o plugin está configurado para ouvir eventos de criação de tickets e chamar a função `onTicketCreated` sempre que um novo ticket é criado.

```
Signal::connect('model.created', array($this, 'onTicketCreated'), 'Ticket');
```

A função onTicketCreated é então definida para realizar ações específicas quando um novo ticket é criado:

```
function onTicketCreated($ticket) {
    error_log('Novo ticket criado: ID ' . $ticket->getId());
    // Adicionar lógica adicional a ser executada quando um ticket é criado
}
```

Logs

Nesta demonstração acima, o plugin está a escrever os logs para o ficheiro de logs do Apache. Portanto, basta aceder ao ficheiro `apache/logs/error.log` para visualizar os mesmos.

Tabelas dos Plugins e das Suas Instâncias

No osTicket, quando um plugin é instalado, ele está representado nas tabelas `ost_plugin` e `ost_plugin_instance`. A primeira tabela, `ost_plugin`, contém os plugins instalados, enquanto a segunda tabela, `ost_plugin_instance`, contém as instâncias dos plugins. Cada instância tem um `plugin_id` que corresponde ao ID do plugin na tabela `ost_plugin`. É nestas tabelas que podemos consultar informações sobre os plugins e as suas instâncias.