JTL-Shop Plugins mit Bootstrap-Klasse registrieren

Neben allgemeinen Webentwicklungsthemen, werde ich in diesem Blog auf Themen rund um die Software-Produkte von JTL zu sprechen kommen. Da ich aktuell täglich mit JTL-Produkten arbeite, habe ich einfach zu diesen Themen manchmal mehr zu berichten.
Mit der Version 4.05 vom JTL-Shop hat JTL den Grundbaustein für eine moderne Plugin-Architektur eingeführt. Um die Entwicklung von Plugins zu verbessern bzw. zu beschleunigen, kann man ab dieser Version Plugins über eine Bootstrap-Klasse initialisieren.
Die Funktionalität steckt zwar noch in den Kinderschuhen, ich möchte aber trotzdem schonmal zeigen, wie das funktioniert und welche Möglichkeiten man damit hat.

Bootstrap-Datei und -Klasse anlegen

Damit der JTL-Shop die Bootstrap-Klasse inkludieren kann, muss zunächst eine PHP-Datei angelegt werden mit dem Pfad “/includes/plugins/{pluginId}/version/{version}/bootstrap.php”. In dieser Datei muss entweder die Klasse definiert werden, eine andere PHP-Datei mit der Klasse geladen werden oder ein Autoloader geladen werden, der die entsprechende Klasse bereitstellt. Für die Bootstrap-Klasse ist folgender Code notwendig:

```
namespace meine_plugin_id; // WICHTIG! Muss gleich Plugin-Id sein.

class Bootstrap extends \AbstractPlugin
{
        public function boot(\EventDispatcher $dispatcher)
        {
            parent::boot(); // TODO: Change the autogenerated stub
        }
        public function installed()
        {
            parent::installed(); // TODO: Change the autogenerated stub
        }
        public function uninstalled()
        {
            parent::uninstalled(); // TODO: Change the autogenerated stub
        }
        public function enabled()
        {
            parent::enabled(); // TODO: Change the autogenerated stub
        }
        public function disabled()
        {
            parent::disabled(); // TODO: Change the autogenerated stub
        }
}
```

Hook- und Eventlistener registrieren

Zur Verständnis:
Dispatcher ist ein „Eventverteiler“. Eingehende Ereignisse werden an die registrierten Stellen weitergeleitet. Wie die Telefonistin, die am Schaltpult sitzt wie man es in alten Filmen sieht (im Englischen heisst dieser Job tatsächlich „phone dispatcher“). Sie nimmt den Anrufer entgegen, stellt fest ob der Empfänger existiert und stellt dann eine Verbindung zwischen den beiden her. Danach ist ihr Job erledigt. Listener für Hooks und Events können innerhalb der Methode boot() registriert werden:

```
namespace meine_plugin_id;

class Bootstrap extends \AbstractPlugin
{
    public function boot(\EventDispatcher $dispatcher) {
        $dispatcher->listen(
            'shop.hook.' . HOOK_SMARTY_OUTPUTFILTER,
            function() {
                pq('body')->append('Hallo Welt');
            }
        );
    }
    ...
}
```

Selbstverständlich muss nicht zwingend eine anonyme Funktion verwendet werden, sondern es kann jede beliebige callable übergeben werden. Parameter werden als Funktionsparameter definiert:

```
namespace meine_plugin_id;

class WarenkorbManipulator
{
    public function __invoke(
        $hookId,
        $productId,
        $allPositions,
        $amount,
        $exists
    ) {
        // Do something
    }
}

class Bootstrap extends \AbstractPlugin
{
    public function boot(\EventDispatcher $dispatcher) {
        $dispatcher->listen(
            'shop.hook.' . HOOK_WARENKORB_CLASS_FUEGEEIN,
            new WarenkorbManipulator()
        );
    }
    ...
```

```
}
```

Abgesehen von den Hooks, die über 'shop.hook.{hookId}' erreichbar sind, kann man über den EventDispatcher auf weitere Funktionalität zugreifen. Aktuell gibt es folgende Events abgesehen von den Hooks:

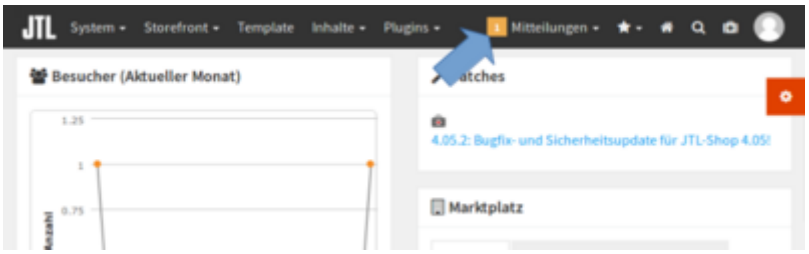
- backend.notification
- shop.run

Über backend.notification können Notifications im Shop-Backend angezeigt werden:

```
namespace meine_plugin_id;

class Bootstrap extends \AbstractPlugin
{
    public function boot(\EventDispatcher $dispatcher) {
        $dispatcher->listen(
            'backend.notification',
            function(\Notification $notify) {
                $notify->add(
                    \NotificationEntry::TYPE_WARNING,
                    "Das Plugin xyz ist nicht vollständig eingerichtet."
                );
            }
        );
    }
    ...
}
```

Die angegebene Notification wird dann im Shop-Backend angezeigt:



Migrations-Scripte

Wenn man der von JTL vorgesehenen Plugin-Struktur folgt, sollten Migrations-Scripte über die info.xml definiert werden. Dort kann für jede Version eine SQL-Datei angegeben werden, in der man SQL-Befehle zum Upgrade auf die nächste Version definieren kann.

```
<Install>
    <Version nr="105">
        <CreateDate>2017-01-01</CreateDate>
        <SQL>up.sql</SQL>
    </Version>
</Install>
```

Diese Struktur hat verschiedene Probleme:

- Es können nicht mehrere Migrations-Scripte pro Version ausgeführt werden
- Definition von Down-Scripten nicht möglich
- JTL setzt fehlerhaften SQL-Parser ein, um Namenskonventionen durchzusetzen

Über Methoden installed() und uninstalled() kann man diesen Mechanismus umgehen und richtige Migrations-Scripte schreiben. Es wäre natürlich wünschenswert, wenn JTL in Zukunft eine Möglichkeit bieten würde um Migrationen zu registrieren. Dies würde eine einheitliche Struktur bei den Plugins ermöglichen. Bis dahin kann man verschiedene Libs nutzen, um die Migrationen zu verwalten. Eine hiervon wäre z.B. Phinx. Hierbei sollte allerdings darauf geachtet werden, dass die Migrationstabelle in der Datenbank mit der Plugin-Id gepräfixt wird, da hier ansonsten die Migrationen der verschiedenen Plugins durcheinander kommen.

```
namespace meine_plugin_id;

class Bootstrap extends \AbstractPlugin
{
    public function installed()
    {
        // execute up scripts
    }
    public function uninstalled()
    {
        // execute down scripts
    }
    ...
}
```

Fazit

Natürlich ist man nicht auf die von mir genannten Bespiele eingeschränkt. Wenn die Methode boot() aufgerufen wird, ist bereits der JTL-Shop-Kontext geladen. Dadurch kann man auf die gesamte API des JTL-Shops zugreifen. Ich hoffe, dass JTL in Zukunft alle Inhalte aus der info.xml über die Bootstrap konfigurierbar machen wird. Ich halte es für den richtigen Weg, die info.xml Schritt für Schritt zu beschneiden und die Funktionalitäten in die Bootstrap-Klasse zu überführen. Ich bin gespannt, in welche Richtung JTL den Shop entwickeln wird.

Quelle: https://mschop.de/blog/jtl-shop-plugins-mit-bootstrap-klasse-registrieren-ger/