Handbuch
für die Entwicklung eigener
Papoo-Plugins
für Papoo Version 3.5

Autor: Stephan Bergmann
S.Bergmann@Sprechomat.de
Stand: Juni 2007

Vorwort 3 (Januar 2007)

Mit Papoo Version 3.5 werden wieder ein paar Kleinigkeiten geändet. So bringt eben jeder Fortschritt seine Opfer, sprich bestehende Plugins müssen schon wieder angepasst werden. Wie auch schon beim letzten Versions-Sprung, sind die Änderungen aber nicht besonders umfangreich und können im Normalfall mit wenigen Handgriffen erledigt werden.

Der Grund für die Änderungen liegt hauptsächlich darin, dass wir uns intensiv mit der Mehrsprachigkeit beschäftigt haben. So haben wir Papoo Version 3.5 komplett auf UTF-8 umgestellt. Es können jetzt Seiten erstellt werden, deren Sprache nicht den Zeichensatz ISO-8859-1 (auch bekannt als Latin-1) verwenden, z.B. Russisch oder gar Japanisch. Zwar bieten wir im Moment noch keine Übersetzungen der sogenannten "Messages" für solche Sprachen an, das wird aber in Zukunft sicher kommen. Mit tatkräftiger Untertützung der Papoo Gemeinde ist damit sogar in naher Zukunft zu rechnen.

Vorwort 2 (April 2006)

Mit dieser überarbeiteten Version des Handbuchs werden auch die neuen Gegebenheiten von Papoo Version 3 berücksichtig. Die Grundlagen sind dabei weitestgehend gleich geblieben. Als signifikante Neuerung gegenüber der vorgherigen Version bietet Papoo in der Version 3 nun sogenannte Module. Dafür gibt es in der Administration von Papoo unter dem Menü-Punkt "System" den Modul-Manager. Um sich einen groben Überblick darüber zu verschaffen, was dieser Modul-Manager macht, kann damit einfach mal ein bisschen "herumgespielt" werden. Im Prinzip können mit dem Modul-Manager Module verschiedenen Bereichen zugeordnet werden.
Da diese Funktionalität natürlich auch für Plugins hervorragende Möglichkeiten bietet, wurde die Plugin-Schnittstelle um die Möglichkeit der Definition und Einbindung von Modulen erweitert. Mehr dazu dann im Handbuch selbst.

Da sich Neuerungen meist auch auf bestehende Dinge auswirken, müssen Plugins, welche für Papoo Version 2 entwickelt wurden, an die Version 3 angepasst werden. Jetzt aber keine Panik. Die Änderungen sind nur minimal. Der einzige Unteschied besteht in den Templates für das Frontend. Dort mussten bisher immer includes für den Kopf, das Menü, die dritte Spalte und den Fuss (`{include file="top.inc.html"}` etc.) angegeben werden. Das entfällt mit der Version 3. Im einfachsten Fall müssen zur Anpassung also lediglich die entsprechenden Einträge aus den Frontend-Templates entfernt werden. Für die Templates des Backends bleibt alles beim Alten.

Vorwort 1 (Juni 2005)

Hallo lieber Entwickler, hallo liebe Entwicklerin.

Schön das wir dich in der Gemeinde der Papoo-Entwickler begrüßen dürfen. Sicher hast du dich schon mit Papoo, dem barrierefreien Content-Management-System (CMS) beschäftigt. Du bist auch sicher zu der Erkenntniss gekommen, dass Papoo ein sehr einfach zu bedienendes CMS ist. Nur hättest du dir schon immer diese oder jene kleine Erweiterung gewünscht. Genau dafür ist der sogenannte Plugin-Manager gedacht. Der Plugin-Manager bietet die Möglichkeit die Grundfunktionen von Papoo durch neue Funktionalitäten zu erweitern. Es besteht dabei die Möglichkeit bereits vorhandene Plugins in Papoo einzubinden oder eben neue Plugins zu programmieren. Da du dieses Dokument gerade liest, gehe ich davon aus, dass du dich eher für den zweiten Punkt, also die Programmierung eigener Plugins interessierst. Bei diesem Bestreben möchte ich dich nun mit diesem kleinen Handbuch möglichst gut unterstützen. Sollte dir trotzdem etwas nicht klar werden, kannst du dich mit deinen Fragen gerne an das Papoo-Forum wenden.

Noch eine kleine Anmerkung zum Plugin-Manager. Da dieser von mir im Moment noch programmiert wird, ist er noch nicht 100-prozentig ausgereift. So wirst du auch einige Punkte in diesem Handbuch finden, die noch nicht wirklich toll sind. Im Großen und Ganzen sollte es aber so sein, dass du auf dieser Basis arbeiten kannst, ohne später böse Überaschungen zu erleben, indem zu z.B. feststellst, dass du Teile deiner Arbeit nochmals machen mußt.

Jetzt aber genug gelabert. Die Ärmel hochgekrempelt, den Kaffe/Tee bereitgestellt, das Gehirn eingeschalten und frohen Mutes an's Werk.

Stephan

Inhaltsverzeichnis

Plugin Datenstruktur

Die Papoo-eigene Datenstruktur ist dir sicher bekannt. Sie ist in ziemlich logisch strukturierte Verzeichnisse unterteilt. Es gibt z.B. ein Verzeichnis /lib welches sämtliche Papoo-Klassen enthält. Oder das Verzeichnis /bilder, in dem sämtliche Bilder liegen welche Papoo benutzt, oder eben das Verzeichnis /plugins, mit welchem du dich die nächste Zeit beschäftigen wirst. Da außer dem /plugins-Verzeichnis alle Verzeichnisse von Papoo benutzt werden, wirst du sicher vestehen, dass du deine gesamte Arbeit auf das Verzeichnis /plugins zu beschränken hast. Warum? ganz einfach. Stell dir mal vor, was passiert, wenn ein UpDate für Papoo erscheint. Würden Teile deiner Arbeit außerhalb des /plugins-Verzeichnisses liegen, wären diese nach dem Update überschrieben und du müsstest deine Arbeit wieder unter großen Mühen in Papoo integrieren. Hört sich nicht besonders sinnvoll an und ist es auch nicht. Deshalb bietet Papoo eben die Möglichkeit der Plugins. So kann der Kern von Papoo problemlos ersetzt werden ohne deine eigene Entwicklung zu beeinflussen. Um das Ganze also kurz zusammen zu fassen:

Sämtliche Daten der Plugins liegen im Verzeichnis /plugins.

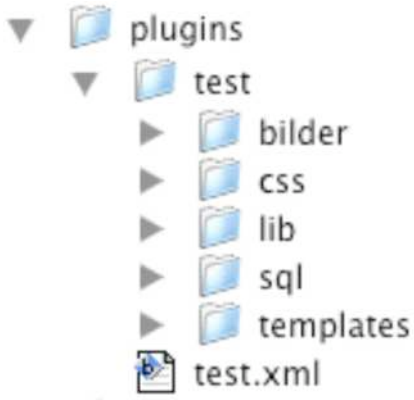
Innerhalb des Verzeichnisses /plugins befinden sich pro Plugin zwei Einträge. Der Eine ist eine XML-Datei, welche wichtige Informationen über das Plugin enthält. Der zweite Eintrag ist ein Verzeichnis, welcher sämtliche Dateien des Plugins enthält. Dieses Verzeichnis enthält zur besseren Strukturierung wiederum einige Unterverzeichnisse, welche im Großen und Ganzen der Struktur von Papoo selbst entsprechen. Es sind dort also, wie bei Papoo selbst, Verzeichnisse wie /lib oder /bilder zu finden. Diese Struktur ist zwar nicht zwingend vorgeschrieben, erweist sich aber als durchaus hilfreich und sollte von dir auch so übernommen werden.

Die wichtigsten Verzeichnisse sind (in alphabetischer Reihenfolge):

- /bilder (enthält sämtliche Bilder, welche dein Plugin benötigt)
- /css (enthält die CSS-Auszeichnungs-Datei deines Plugins)
- /lib (enthält die Klassen deines Plugins)
- /sql (enthält Installations- und Deinstallations-Anweisungen für die Datenbank in Form zweier SQL-Dateien)
- /templates (enthält sämtliche Template-Vorlagen deines Plugins)

Wie schon gesagt ist diese Struktur nicht zwingend notwendig, hat sich aber als sehr praktisch erwiesen und sollte deshalb von dir auch so übernommen werden. Sollte deine Entwicklung weitere Verzeichnisse benötigen, so kannst du diese natürlich dieser Struktur hinzufügen.

Damit du dir jetzt mal ein bisschen besser vorstellen kannst wie das Ganze aussehen soll, hier ein Bild der Verzeichnis-Struktur (Bilder sagen ja bekanntlich mehr als tausend Worte):



Verzeichnis-Struktur Plugins

Wie du dem kleinen Bildchen entnehmen kannst liegt hier ein Test-Plugin vor. Dieses Test-Plugin sollte dieser Dokumentation in gepackter Form (als ZIP-Archiv) beiliegen. Ansonsten kannst du dir das Ding sicher auch von der Papoo-Site www.papoo.de unter "PlugIns" herunter laden. Was dir an dem Bildchen vielleicht auch noch auffällt, ist die etwas "ungewöhnliche" Darstellung der Verzeichnisse. Dies liegt ganz einfach daran, dass ich mit einem Apple Macintosh arbeite (jaja solche Leute gibt es tatsächlich). Dem

Informationsgehalt sollte das aber keinen Abbruch tun. Um die Sache nochmals zusammen zu fassen:

Jedes Plugin besteht aus zwei Einträgen im /plugins-Verzeichnis:
1. einer XML-Datei, sowie
2. einem Verzeichnis mit mehreren Unterverzeichnissen.

Die XML-Datei

Das Herz eines jeden Plugins ist die zugehörige XML-Datei. Diese XML-Datei enthält viele Informationen über das Plugin. Diese Informationen lassen sich grob in zwei verschiedene Bereiche unterteilen:

1. allgemeine Informationen,
2. interne Infomationen.

Unter allgemeine Informationen fallen z.B. der Name des Plugins, eine kurze Beschreibung oder auch Daten des Autors (Name, eMail etc.). Zu den internen Informationen zähle ich z.B. die Angabe der Datenbank-Installations-Datei, oder Namen der eigenen Klassen. Einige Informationen sind auch nicht klar dem einen oder anderen Bereich zuzuordnen, wie etwa die Angabe der Versions-Nummer.

Damit du dir einen Eindruck über eine solche XML-Datei verschaffen kannst, hier jetzt mal die XML-Datei des Test-Plugins. Diese ist mit zahlreichen Kommentaren versehen, sodass die einzelnen Einträge (bei XML-Dateien "Tags" genannt) eigentlich verständlich werden sollten:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!-- version: Versions-Nummer papoo für die das Plugin gedacht ist / funktioniert -->
<plugin version="3.0">

<!-- ALLGEMEINE-INFORMATIONEN -->
<!-- ********************** -->
<!-- name: Name des Plugins -->
<name>Test-Plugin</name>

<!-- beschreibung_de: Kurz-Beschreibung des Plugins auf deutsch (wird im Plugin-Manager angezeigt) -->
<beschreibung_de><![CDATA[Dieses Plugin dient zu reinen Testzwecken.
Es hat also keinen weiteren Sinn außer die Einbindung von Plugins in papoo zu erklären.]]></beschreibung_de>

<!-- beschreibung_en: Kurz-Beschreibung des Plugins auf englisch (wird im Plugin-Manager angezeigt) -->
<beschreibung_en><![CDATA[This Plugin is only for Tests.
The only reason for this plugin is to show you how you can integrate plugins into papoo.]]></beschreibung_en>

<!-- version: Versions-Nummer des Plugins -->
<version>3.0</version>

<!-- papooid: Eindeutige Nummer offizieller papoo-Plugins -->
<papooid>3</papooid>

<!-- link: z.Z. noch ohne Funktion -->
<link>http://www.papoo.de</link>

<!-- datum: Erscheinungs-Datum bzw. Stand des Plugins -->
<datum>29.03.2006</datum>


<!-- AUTOR-INFORMATIONEN -->
<!-- ******************* -->
<!-- autor: Informationen zum Autor (Programmierer) des Plugins -->
<autor>
        <!-- autor.name: Name des Authors -->
        <name>Stephan Bergmann</name>

        <!-- autor.kontakt: postalische Kontakt-Informationen des Autors -->
        <kontakt><![CDATA[Stephan Bergmann
Rembrandtstr. 10/1
72768 Reutlingen]]></kontakt>

        <!-- autor.email: eMail-Adresse des Autors -->
        <email>S.Bergmann@XiMiX.de</email>

        <!-- autor.telefon: Telefon-Nummer des Autors -->
        <telefon>07121 / 60 39 88</telefon>
</autor>


<!-- MENÜ-INFORMATIONEN -->
<!-- ****************** -->
<!-- menue: Informationen für den Menü-Eintrag im Backend (papoo/interna) -->
<menue>
        <!-- menue.eintrag_de: Name des Menü-Punktes im Backend auf deutsch -->
        <eintrag_de>TEST-PlugIn</eintrag_de>
        <!-- menue.eintrag_en: Name des Menü-Punktes im Backend auf englisch -->
        <eintrag_en>TEST-PlugIn</eintrag_en>
        <!-- menue.icon: Pfad zum Menü-Icon des Menü-Punktes im Backend (relativ zum Plugin-Verzeichnis) -->
        <icon>test/bilder/pic_01.gif</icon>
        <!-- menue.link: Name der Seite auf welche der Menü-Punkt im Backend verweisen soll -->
        <link>plugin:test/templates/test_back.html</link>
```

```
        <!-- menue.submenue: Untermenü-Punkt für diesen Menü-Punkt im Backend, Struktur wie menue -->
        <submenue>
                <eintrag_de>Submenue1_1</eintrag_de>
                <eintrag_en>Submenue1_1</eintrag_en>
                <icon>test/bilder/pic_01_01.gif</icon>
                <link>plugin:test/templates/test_back.html</link>
        </submenue>

        <submenue>
                <eintrag_de>Submenue1_2</eintrag_de>
                <eintrag_en>Submenue1_2</eintrag_en>
                <icon>test/bilder/pic_01_01.gif</icon>
                <link>plugin:test/templates/test_back.html</link>
        </submenue>
</menue>

<!-- menue: Es können auch weitere Menü-Punkte im Backend eingetragen werden -->
<menue>
        <eintrag_de>TEST-PlugIn2</eintrag_de>
        <eintrag_en>TEST-PlugIn2</eintrag_en>
        <icon>test/bilder/pic_02.gif</icon>
        <link>plugin:test/templates/test_back.html</link>
</menue>


<!-- KLASSEN-INFORMATIONEN -->
<!-- ********************* -->
<!-- klasse: Informationen der Plugin-Klasse welche eingebunden werden sollen -->
<klasse>
        <!-- klasse.name: Name der Klassen-Instanz / des Objekts (also der Name des Dings, das mit $name = new
xx_class(); erzeugt wird) -->
        <name>testplugin</name>
        <!-- klasse.datei: Pfad zur Klassen-Datei (relativ zum Plugin-Verzeichnis) -->
        <datei>test/lib/testplugin_class.php</datei>
</klasse>

<!-- klasse: Es können auch weitere Plugin-Klassen eingebunden werden -->
<klasse>
        <name>testplugin2</name>
        <datei>test/lib/testplugin2_class.php</datei>
</klasse>


<!-- CSS-INFORMATIONEN -->
<!-- ***************** -->
<!-- css: Pfad zur CSS-Datei (relativ zum Plugin-Verzeichnis) -->
<css>test/css/test.css</css>


<!-- DATENBANK-INFORMATIONEN -->
<!-- ********************** -->
<datenbank>
        <!-- datenbank.installation: Pfad zur SQL-Installations-Datei (relativ zum Plugin-Verzeichnis) -->
        <installation>test/sql/test_install.sql</installation>
        <!-- datenbank.deinstallation: Pfad zur SQL-Deinstallations-Datei (relativ zum Plugin-Verzeichnis) -->
        <deinstallation>test/sql/test_deinstall.sql</deinstallation>
</datenbank>

<!-- MODUL-INFORMATIONEN -->
<!-- ******************* -->
<!-- modul: Informationen des Frontend-Moduls -->
<modul>
        <!-- modul.datei: Datei-Name des Modul-Templates (relativ zum Plugin-Verzeichnis) -->
        <datei>plugin:test/templates/mod_test_front.html</datei>

        <!-- modul.name_de: Der deutsche Name des Moduls -->
        <name_de>Test-Modul</name_de>
        <!-- modul.beschreibung_de: Die deutsche Beschreibung des Moduls -->
        <beschreibung_de>Die ist lediglich ein Test-Modul des Test-Plugins.</beschreibung_de>

        <!-- modul.name_en: Der englische Name des Moduls -->
        <name_en>test-modul</name_en>
        <!-- modul.beschreibung_en: Die englische Beschreibung des Moduls -->
        <beschreibung_en>this is a test-modul of the test-plugin.</beschreibung_en>
</modul>

<!-- Beispiel eines "fixen" Frontend-Moduls. Dieses Modul ist nur für den Bereich "Fuss" (5) verfügbar -->
<modul>
        <!-- modul.datei: Datei-Name des Modul-Templates (relativ zum Plugin-Verzeichnis) -->
        <datei>plugin:test/templates/mod_testfix_front.html</datei>

        <!-- modul.name_de: Der deutsche Name des Moduls -->
        <name_de>Test-Modul (fix)</name_de>
        <!-- modul.beschreibung_de: Die deutsche Beschreibung des Moduls -->
        <beschreibung_de>Die ist lediglich ein Test-Modul des Test-Plugins.</beschreibung_de>
```

```
        <!-- modul.name_en: Der englische Name des Moduls -->
        <name_en>test-modul (fix)</name_en>
        <!-- modul.beschreibung_en: Die englische Beschreibung des Moduls -->
        <beschreibung_en>this is a test-modul of the test-plugin.</beschreibung_en>

        <!-- modul.modus: (optional) legt fest, ob das Modul in allen Bereichen, oder nur in einem bestimmten
        Bereich angezeigt werden kann.
        mögliche Werte:
        "var" (default) Anzeige in allen Bereichen möglich,
        "fix" Anzeige nur in einem bestimmten Bereich möglich -->
        <modus>fix</modus>

        <!-- modul.bereich:
        legt bei "fix"-Modulen die Nummer des Bereichs fest, in dem das Modul angezeigt werden kann.
        mögliche Werte: "1": Kopf; "2": linke Spalte; "3": mittlere Spalte; "4": rechte Spalte; "5": Fuss -->
        <bereich>5</bereich>
</modul>

<!-- Einbindung der Sprach-Dateien.
        In dem hier angegebenen Verzeichnis sollten Dateien mit folgenden Namen liegen:
        - messages_backend_[de|en|fr| .. etc.].inc.php
        - messages_frontend_[de|en|fr| .. etc.].inc.php
-->
<messages>test/messages</messages>

</plugin>
```

Alles klar? Nein? macht nichts. Du bist ja auch noch nicht am Ende dieses Kapitels angelangt. Also jetzt aufgrund der "Komplexität" dieser XML-Datei nur nicht den Mut verlieren. Du wirst das alles recht schnell verstehen. Um dieses Verständnis voran zu treiben, unterteile ich diese Datei nochmal in kleine, mundgerechte Häppchen und schreibe einige erläuternde Worte dazu.

Die allgemeinen Tags

In der XML-Datei finden sich unter anderem folgende Eintragungen:

```
<!-- name: Name des Plugins -->
<name>Test-Plugin</name>

<!-- beschreibung_de: Kurz-Beschreibung des Plugins auf deutsch (wird im Plugin-Manager angezeigt) -->
<beschreibung_de><![CDATA[Dieses Plugin dient zu reinen Testzwecken.
Es hat also keinen weiteren Sinn außer die Einbindung von Plugins in Papoo zu erklären.]]></beschreibung_de>

<!-- beschreibung_en: Kurz-Beschreibung des Plugins auf englisch (wird im Plugin-Manager angezeigt) -->
<beschreibung_en><![CDATA[This Plugin is only for Tests.
The only reason for this plugin is to show you how you can integrate plugins into Papoo.]]></beschreibung_en>

<!-- version: Versions-Nummer des Plugins -->
<version>3.0</version>
```

Ich denke mal zum Sinn dieser Tags ist nicht allzuviel zu sagen. Auf eine Besonderheit möchte ich an dieser Stelle jedoch Hinweisen. Wer genau hinschaut, wird bei den beiden Beschreibungs-Tags das etwas seltsame Konstrukt `<![CDATA[..]]>` entdecken. Dieses Konstrukt ist nötig um besondere Zeichen wie z.B. Umlaute korrekt in XML-Dateien einbinden zu können. Der XML-Interpreter weiß, dass Zeichen innerhalb des CData-Konstrukts als Zeichen des im Kopf der XML-Datei angegebenen Zeichensatzes zu interpretieren sind. Das meint die folgende Zeile:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
```

In unserem Falls ist das `encoding="iso-8859-1"`, auch bekannt als ISO-LATIN-1, also der Zeichensatz, den z.B. Windows in der deutschen oder auch der englischen Version als Standard-Zeichensatz benutzt. "Mac-User" wie ich es einer bin (also Menschen die mit einem Apple Macintosh) arbeiten verwenden standardmässig einen anderen Zeichensatz, was manchmal zu Problemen bei der Darstellung von Umlauten etc. führen kann. Werden in der XML-Datei Zeichen einer Sprache verwendet, welche nicht mi diesem Zeichnsatz darstellbar sind (z.B. russische Zeichen), sollte als encoding UFT-8 gewählt werden, also so:

```
<?xml version="1.0" encoding="utf-8" ?>
```

Die Datei **muss** dann auch in diesem Format gespeichert werden. Dein Editor sollte dazu eine entsprechende Möglichkeit bieten.

Lange Rede kurzer Sinn:

Angaben die Sonderzeichen wie z.B. Umlaute enthalten müssen innerhalb eines `<![CDATA[ blabla ]]>`-Konstrukts angegeben werden.

Die allgemeinen Informationen werden im Plugin-Manager dann so, oder so ähnlich, dargestellt:

Darstellung allgemeiner Informationen

In Zukunft werden wohl noch einige weitere Informationen dazu kommen (z.B. Links zu UpDate-Check o.Ä.). Das Prinzip sollte aber klar sein.

Die in der XML-Datei folgenden Information über Autor werde ich hier nicht näher betrachten, da sie z.Z. von Papoo nicht berücksichtig werden. Wenden wir uns nun lieber mal etwas spannenderen Einträgen zu, nämlich den Menü-Eintragen.

Zusatz-Info "Änderung gegenüber Vorgänger-Version"
Wie du sicher bemerkt hast, werden Angaben zur Sprache jetzt nicht mehr ausgeschrieben, sodern nur noch mit einem zwei Zeichen langen Kürzel angegeben. Alte Angaben wie `beschreibung_deutsch` werden also jetzt als `beschreibung_de` angegeben.
Diese Änderungen betreffen sämtliche Angaben zu Sprachen in der XML-Datei.

Die Menü-Tags

Die meisten Plugins werden wohl einen ode mehrere Menü-Einträge im Menü des Backends benötigen. Aus diesem Grund gibt es in der XML-Datei die Menü-Tags. Diese sehen wie folgt aus:

```
<menue>
        <!-- menue.eintrag_de: Name des Menü-Punktes im Backend auf deutsch -->
        <eintrag_de>TEST-PlugIn</eintrag_de>
        <!-- menue.eintrag_en: Name des Menü-Punktes im Backend auf englisch -->
        <eintrag_en>TEST-PlugIn</eintrag_en>
        <!-- menue.icon: Pfad zum Menü-Icon des Menü-Punktes im Backend (relativ zum Plugin-Verzeichnis) -->
        <icon>test/bilder/pic_01.gif</icon>
        <!-- menue.link: Name der Seite auf welche der Menü-Punkt im Backend verweisen soll -->
        <link>plugin:test/templates/test_back.html</link>
</menue>
```

Sie bestehen im Wesentlichen also aus drei Elementen:
1. dem Text des Menü-Eintrags (in deutsch und englisch),
2. einem Icon für den Menü-Punkt und
3. dem Link auf die Seite, welcher diese Menüeintrag repräsentiert.

Braucht euer Plugin einen Eintrag im Backend-Menü, müsst ihr also nichts weiter tun, als das entsprechende Tag in der XML-Datei mit euren Werten zu versehen. Alles andere macht dann der Plugin-Manager bei der Installation des Plugins für euch. Coole Sache, oder? Na ich finde schon, zumindest wenn ich daran denke, in wieviele Tabellen da was eingetragen werden muß. Die Mühe das alles selber rauszufinden könnt ihr euch also getrost sparen.

Kurz nochmal was zu den jeweiligen Tags. Beim ersten Tag, also dem Name des Menüpunktes müsst ihr, wie oben erwähnt, darauf achten, dass ihr bei Einträgen mit Umlauten das `<![CDATA[..]]>`-Konstrukt verwendet. Sonst wird der Eintrag nicht korrekt angezeigt.

Zusatz-Info "Änderung gegenüber Vorgänger-Version"

Auch hier gilt, dass Angaben zur Sprache jetzt nicht mehr ausgeschrieben, sodern nur noch mit einem zwei Zeichen langen Kürzel angegeben. Alte Angaben wie `eintrag_deutsch` werden also jetzt als `eintrag_de` angegeben.

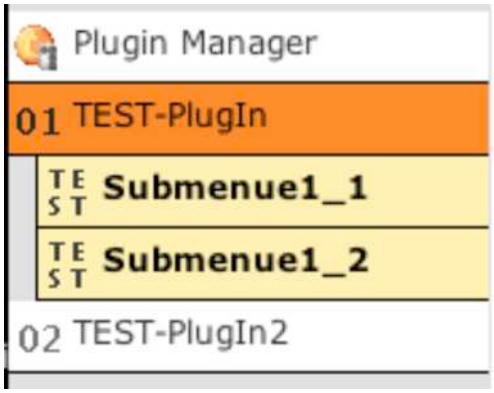
Das Icon des Menü-Punktes sollte eine Größe von 18x18 Pixeln haben. Das ist ein Wert, welcher sich als recht gut erwiesen hat und wird somit jetzt einfach mal zum Standard erhoben.

Die Angabe des Links sieht auf den ersten Blick etwas seltsam aus: `plugin:test/templates/test_back.html`. Das die entsprechende Template-Datei in dem angegebenen Verzeichnis `test/templates/test_back.html` liegt kann ja noch leicht nachvollzogen werden. Warum aber dieses seltsame `plugin:` am Anfang? Ganz einfach. Papoo verweist mit all seinen Menü-Einträgen im Backend auf *.php-Dateien welche im Verzeichnis /interna/templates liegen. Wenn ihr nun eigene Dateien einbinden wollt, müsstet ihr ja auch eine *.php-Datei in das interna Verzeichnis legen. Tja und dann sind wir bei dem ganz zu Anfang erwähnten Problem bei Updates. Ihr müsstet die Dateien jedes mal neu in das /interna/templates-Verzeichnis legen. So viel Arbeit will sich ja keiner machen, zumindest ich nicht (ich bin halt etwas faul :-)). Deshalb habe ich diesen Mechanismus zum Einbinden eigener Template-Seiten in den Plugin-Manager gemacht. Also:

Links zu eigenen Template-Dateien werden per `plugin:pfad/templates/name_des_templates.html` angegeben.

Entsprechend diesen Angaben werden übrigens auch eigene Templates in das Frontend integriert. Dazu einfach im Backend unter "Menü verwalten->Menü-erstellen" unter "formlink" folgendes eingeben: `plugin:pfad/templates/name_des_templates.html` und gut wars. Auch nicht schlecht, oder?

Das war's dann auch schon fast, was zu den Menü-Tags in der XML-Datei zu sagen ist. Aber auch nur fast. Wie der test.xml-Datei zu entnehmen ist, können Erstens eine beliebige Anzahl an Menü-Punkten, und Zweitens auch Untermenü-Punkte angegeben werden. In all seiner Pracht sieht das Backend-Menü nach der Installation des Test-Plugins so aus:



Backend-Menü des Test-Plugins

Das ist doch eine echt coole-Sache, oder? Ihr schreibt einfach ein paar Anweisungen in die XML-Datei, gestaltet ein paar nette Icon-Bildchen und der Plugin-Manager baut euch wunderschöne Menü-Einträge. So muß das sein. Ich denke mal, so langsam verlierst du die Angst vor der auf den ersten Blick doch recht komplexen XML-Datei und findest die Sache zunehmend spannend. Also weiter im Text.

Die Klassen-Tags

Du hast sicher schon gesehen, wie Papoo seine Klassen einbindet bzw. wie Papoo mit Klassen umgeht. Die Einbindung der PHP-Dateien, welche den Code der jeweiligen Klasse beinhaltet, geschieht in der Datei /lib/classes.php. Dort werden alle von Papoo benutzten Klassen, bzw. die entsprechenden PHP-Dateien eingebunden. Im Prinzip müsstest du deine eigenen Klassen ebenfalls in dieser Datei einbinden. Dies würde aber wieder dem Grundsatz 'Sämtliche Daten der Plugins liegen im Verzeichnis /plugins' wiedersprechen. Aus diesem Grund werden die Angaben, welche PHP-Klassen-Dateien eingebunden werden sollen, in die XML-Datei geschrieben. Eigentlich klar, oder? Ein solcher Eintrag sieht dann exemplarisch so aus:

```
<klasse>
        <!-- klasse.name: Name der Klassen-Instanz / des Objekts (also der Name des Dings, das mit $name = new
xx_class(); erzeugt wird) -->
        <name>testplugin</name>
        <!-- klasse.datei: Pfad zur Klassen-Datei (relativ zum Plugin-Verzeichnis) -->
        <datei>test/lib/testplugin_class.php</datei>
</klasse>
```

Du siehst also, dass es pro eingebundener Klasse einen Tag <klasse> in der XML-Datei gibt. Dieser Tag enthält wiederum zwei Tags, nämlich <name> und <datei>. Der Einfachkeit halber fange ich mal mit dem zweiten Tag an.

Der Tag <datei> enthält die Pfadangabe zu deiner eigenen PHP-Datei, welche den Code deiner Klasse enthält. Der Pfad wird dabei relativ zum /plugins-Verzeichnis angegeben. Der Plugin-Manager macht mit dieser Angabe ein `require_once(klasse.datei)` und schon steht die Klasse zur Verfügung.

Soweit so gut. Wozu also noch der zweite Tag <name>? Das hat mit einer Konvention von Papoo zu tun, Objekte einer Klasse global zu instanzieren. Was bedeutet das? Du hast sicher schon gesehen, das am Ende einer jeden Klassen-Definition eine Anweisung der Art: `$blabla = new blabla_class();` steht. Meist steht dann auch noch so ein netter Kommentar wie folgt dabei: `//Hiermit wird die Klasse initialisiert und kann damit sofort überall benutzt werden.` Das ist genau die Sache was ich mit globaler Instanzierung meine. OK, OK. Ich gebe zu, diese Begriffe sind sehr verwirrend. Das ist aber leider bei Objekt-orientierter Programmierung so. Irgendwie konnte man sich nie genau auf bestimmte Begriffe festlegen, was wohl hauptsächlich daran liegt, das sie allesamt eher kryptisch sind (z.B. Instanz, Methode, Attribut). Sei's drum. Ich möchte mich hier nicht weiter über Begrifflichkeiten auslassen, Worte sind schließlich nur "Schall und Rauch". Fakt jedoch ist, dass Papoo nach jeder Definition einer Klasse ein Objekt dieser Klasse erzeugt, welches dann global zur Verfügung steht. Da nun jedes Objekt einen Namen haben muss, genau so wie z.B. ein Variable einen Namen hat, gibt es in der XML-Datei eben dieses zweite Klassen-Tag <name>. Dieser Name entspricht dem Namen des globalen Objekts deiner Klasse. Ein solcher Name ist immer dann wichtig, wenn deine Klasse von einer Anderen benutzt werden soll. Du wirst das noch sehen, wenn wir uns den Code der beiden Beispiel-Klassen etwas genauer anschauen. Dort benutzt nämlich die Klasse testplugin2_class die Klasse testplugin_class. Um dies zu ermöglichen, braucht eben das Objekt der Klasse testplugin_class einen Namen, in unserem Fall "testplugin".

Puuh, das war jetzt zugegeben etwas stressig und verwirrend. Ich hoffe aber, du hast es trotzdem halbwegs kapiert. Spätestens bei der praktischen Umsetzung, wirst du das Konzept aber sicher kapiert haben. Wenden wir uns nun also einem etwas einfacheren Thema zu.

Das CSS-Tag

Schon das ausbleiben des Plurals (Tag statt Tags) deutet darauf hin, dass die Sache mit dem CSS-Tag ganz einfach wird, und so ist es auch. Hier mal der entsprechende Ausschnitt der XML-Datei:

```
<!-- css: Pfad zur CSS-Datei (relativ zum Plugin-Verzeichnis) -->
<css>test/css/test.css</css>
```

Wie du bei genauerem Hinsehen zweifellos feststellst, beschränkt sich das CSS-Tag auf die Angabe eines Pfades. Dieser Pfad verweist auf eine CSS-Datei, in welcher du sämtliche CSS-Anweisungen deines Plugins unterbringen kannst. Der Plugin-Manager liest den Inhalt dieser Datei, bzw. aller CSS-Dateien sämtlicher PlugIns und bastelt daraus eine einzige CSS-Datei. Die so generierte CSS-Datei wird in den HTML-Template-Dateien im Tag `<style>` per `@import url(dateiname.css)` eingebunden. Der Dateiname dieser CSS-Datei lautet etwa "1234567890_plugins.css", wobei die Zahlen am Anfang nur dafür sorgen, die Datei eindeutig zu Kennzeichnen. Der Sinn der hinter diesem Mechanismus steht ist der, dass die Datei einmal generiert wird und wegen des gleichbleibenden Namens vom Browser zwischengespeichert werden kann. Die Datei wird nur geändert, wenn ein Plugin installiert oder deinstalliert wird. So wird gewährleistet, dass der Browser dann diese neue CSS-Datei lädt und wieder aktuell ist. Alles klar? Ich denke schon, das war ja jetzt nicht so schwierig.

Ein kleiner Hinweis noch an dieser Stelle und dann ist das Thema CSS-Tag auch schon abgeschlossen. Der `@import url()`-Befehl liegt ganz bewußt vor der eigentlichen Papoo-CSS-Einbindung. Somit können Papoo-Benutzer die CSS-Anweisungen deines Plugins mit Einträgen in ihren eigenen CSS-Dateien überschreiben. Deine CSS-Angaben stellen also lediglich eine Art Grundkonfiguration dar. Anders wäre es auch nicht so leicht möglich, Style-Änderungen durch den Papoo-StyleSwitcher effizient umzusetzen.

Die Datenbank-Tags

Wir nähern uns dem Ende der XML-Datei. Die letzten Tags dienen zur Einbindung von Tabellen in die Papoo-Datenbank. Es ist ja durchaus möglich, dass dein Plugin auch eigene Tabellen benutzt. Die zugehörigen Tags sehen folgendermasen aus:

```
<datenbank>
        <!-- datenbank.installation: Pfad zur SQL-Installations-Datei (relativ zum Plugin-Verzeichnis) -->
        <installation>test/sql/test_install.sql</installation>
        <!-- datenbank.deinstallation: Pfad zur SQL-Deinstallations-Datei (relativ zum Plugin-Verzeichnis) -->
        <deinstallation>test/sql/test_deinstall.sql</deinstallation>
</datenbank>
```

Im Prinzip werden in den beiden Datenbank-Tags <installation> bzw. <deinstallation> die Pfade zu zwei SQL-Dateien angegeben. Die erste Datei enthält SQL-Anweisungen zur Erzeugung einer oder mehrerer Datenbank-Tabellen, und zum Füllen dieser Tabellen, falls dies gewünscht ist. Die zweite Datei enthält SQL-Anweisungen zum Löschen der von dir erstellten Tabellen, also üblicherweise simple `DROP TABLE blabla`-Anweisungen.

Damit du dir bei der Erzeugung dieser beiden Dateien, insbesondere der Installations-Datei keinen abbrechen mußt, gibt es dafür ein extra Papoo-Plugin was diese Arbeit für dich erledigt. Dazu jedoch später mehr.

An dieser Stelle möchte ich noch kurz was zum Thema Tabellennamen los werden. Du weisst ja sicher, dass jeder Papoo-Benutzer ein eigenes Tabellen-Namen-Präfix verwendet. Also das Ding, welches beim Setup im Schritt 1 unter "Praefix Name" angegeben werden kann. Da nun jeder Papoo-Benutzer sein eigenes Präfix verwendet, wäre es ziemlich schwierig für dich Tabellen mit dem passenden Präfix zu erzeugen. Du kennst das Präfix desjenigen ja nicht, welcher dein Plugin benutzt. Aus diesem Grund besitzen die Tabellen der Installations- und Deinstallation-SQL-Dateien ein sogenanntes "globales Präfix". Dieses globale Präfix ist "XXX_", also wunder dich nicht, wenn du das in der SQL-Datei des Test-Plugins findest. Durch die Verwendung dieses globalen Präfix schafft es der Plugin-Manager die Tabellen-Namen für den Benutzer deines Plugins anzupassen, indem er schlicht und einfach das globale Präfix "XXX_" durch das jeweilige Präfix des Benutzers ersetzt.

Die Modul-Tags

Nachdem das bisher zur XML-Datei gesagte alles nichts Neues gegenüber den Plugins für Papoo 2 war, hier endlich etwas absolut Neues. Die bereits im Vorwort angesprochenen Module.

```
<!-- MODUL-INFORMATIONEN -->
<!-- ****************** -->
<!-- modul: Informationen des Frontend-Moduls -->
```

```
<modul>
        <!-- modul.datei: Datei-Name des Modul-Templates (relativ zum Plugin-Verzeichnis) -->
        <datei>plugin:test/templates/mod_test_front.html</datei>

        <!-- modul.name_de: Der deutsche Name des Moduls -->
        <name_de>Test-Modul</name_de>
        <!-- modul.beschreibung_de: Die deutsche Beschreibung des Moduls -->
        <beschreibung_de>Die ist lediglich ein Test-Modul des Test-Plugins.</beschreibung_de>

        <!-- modul.name_englisch: Der deutsche Name des Moduls -->
        <name_en>test-modul</name_en>
        <!-- modul.beschreibung_en: Die deutsche Beschreibung des Moduls -->
        <beschreibung_en>this is a test-modul of the test-plugin.</beschreibung_en>
</modul>
```

Prinzipiell ist ein Modul "ein Teil eines Templates" welches an einer bestimmten Stelle in den HTML-Code eingebaut wird. Was nun wieder genau "ein Teil eines Templates" bedeutet, wird im nächsten Kapitel "Papoo und die Template-Engine Smarty" erklärt. Hier soll es lediglich um die Einbindung eigener Module in Papoo gehen. Konkret geht es also darum "Wie schaffe ich es, dass Papoo mir im Modul-Manager meine eigenen Module zur Auswahl anzeigt?".
Genau dies machen die oben angegebenen XML-Anweisungen. Hast du das Test-Plugin installiert, dann zeigt dir der Modul-Manager also etwa folgendes Bild:

Liste der verfügbaren Module erweitert um das Test-Modul.

Das Modul steht in der Auswahl-Liste der verfügbaren Modul zur Verfügung. Es kann in einem beliebigen Bereich der Seite plaziert und anschließend in der Reihenfolge verschoben werden. Es verhält sich also genau wie die Module, welche Papoo schon von Hause aus mitbringt.

Zusatz-Info "Änderung gegenüber Vorgänger-Version"

Auch hier gilt, dass Angaben zur Sprache jetzt nicht mehr ausgeschrieben, sodern nur noch mit einem zwei Zeichen langen Kürzel angegeben. Alte Angaben wie `name_deutsch` werden also jetzt als `name_de` angegeben.

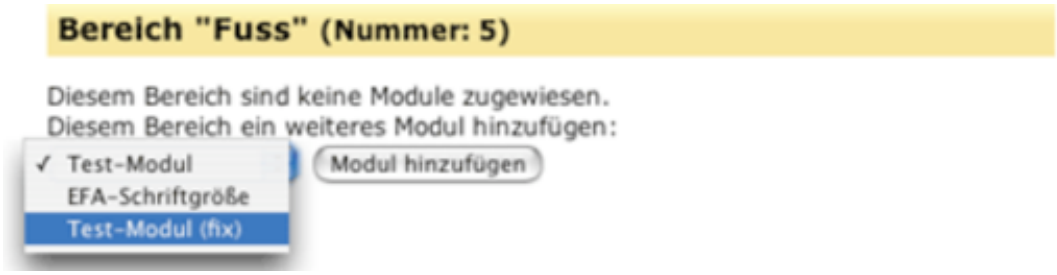
Wer im Modul-Manager genauer hinsieht, wird feststellen, das nicht alle Module in allen Bereichen zur Verfügung stehen. Ein Modul kann auch so definiert werden, dass es nur in einen ganz bestimmten Bereich platziert werden kann. Papoo unterscheidet also zwischen sogenannten "varibalen Modulen" (Standard, können in allen Bereichen platziert werden) und "fixen Modulen" (können nur in einem bsteimmten Bereich platziert werden).
Ob dein Modul variabel oder fix ist, kannst du auch für die in deinem Plugin erstellten Modulen festlegen. Dazu gibt es für Module die beiden optionalen Tags `<modus>` bzw. `<bereich>`. Wird im Tag `<modus>` der Wert `fix` angegeben, so wird damit definiert, dass das Modul nur für einen bestimmten Bereich zulässig ist. Welcher Bereich dass nun genau ist, legt das Tag `<bereich>` fest.

Wie das konkret aussehen kann, zeigt das zweite im Test-Plugin enthaltene Modul. Dafür sehen die XML-Definitionen so aus:

```
<!-- Beispiel eines "fixen" Frontend-Moduls. Dieses Modul ist nur für den Bereich "Fuss" (5) verfügbar -->
```

```
<modul>
        <!-- modul.datei: Datei-Name des Modul-Templates (relativ zum Plugin-Verzeichnis) -->
        <datei>plugin:test/templates/mod_testfix_front.html</datei>

        <!-- modul.name_de: Der deutsche Name des Moduls -->
        <name_de>Test-Modul (fix)</name_de>
        <!-- modul.beschreibung_de: Die deutsche Beschreibung des Moduls -->
        <beschreibung_de>Die ist lediglich ein Test-Modul des Test-Plugins.</beschreibung_de>

        <!-- modul.name_en: Der deutsche Name des Moduls -->
        <name_en>test-modul (fix)</name_en>
        <!-- modul.beschreibung_en: Die deutsche Beschreibung des Moduls -->
        <beschreibung_en>this is a test-modul of the test-plugin.</beschreibung_en>

        <!-- modul.modus: (optional) legt fest, ob das Modul in allen Bereichen, oder nur in einem bestimmten
Bereich angezeigt werden kann.
                mögliche Werte: "var" (default) Anzeige in allen Bereichen möglich, "fix" Anzeige nur in einem
bestimmten Bereich möglich -->
        <modus>fix</modus>

        <!-- modul.bereich: legt bei "fix"-Modulen die Nummer des Bereichs fest, in dem das Modul angezeigt
werden kann.
                mögliche Werte: "1": Kopf; "2": linke Spalte; "3": mittlere Spalte; "4": rechte Spalte; "5": Fuss
-->
        <bereich>5</bereich>
</modul>
```

Neben den bereits im anderen Modul enthaltenen Einträgen besitzt dieses Modul die beiden Einträge `<modus>` und `<bereich>`. Die hier verwendete Bereichs-Nummer 5 sagt, dass das Modul nur im Fuss der Seite platziert werden kann. Das das tatsächlich funktioniert, soll das folgende Bild verdeutlichen:



Auswahl-Liste mit fixem Modul.

Zusammfassend kann also gesagt werden:

In Plugins können Module definiert werden. Diese stehen dann im Modul-Manager entweder als variables oder als fixes Modul zur Verfügung.

Das Sprach-Tag (NEU in Papoo Version 3.5)

Wie du schon im Vorwort gelesen und in den vorigen Abschnitten gemerkt hast, hat sich bei der Unterstützung der Mehrsprachigkeit einiges getan. Bisher musstest du immer selber dafür sorgen, dass die sogenannten "Messages", also Definitionen von verwendeten Text-Schnippseln, in der richtigen Sprache eingebunden werden. Das haben wir durch einen neuen Eintrag in der XML-Datei geändert. Durch den folgenden Eintrag werden automatisch die richtigen Sprach-Dateien eingebunden:

```
<!-- Einbindung der Sprach-Dateien.
        In dem hier angegebenen Verzeichnis sollten Dateien mit folgenden Namen liegen:
        - messages_backend_[de|en|fr| .. etc.].inc.php
        - messages_frontend_[de|en|fr| .. etc.].inc.php
-->
<messages>test/messages</messages>
```

Du musst bei Verwendung des Sprach-Tags deinem Plugin also mindestens zwei Dateien namens `"messages_frontend_de.inc.php"` und `"messages_backend_de.inc.php"` beilegen. Hast du weitere Sprachen definiert, reicht es diese Dateien in das hier angegebene Verzeichnis zu legen. Die Einbindung geschieht automatisch. Als Unterscheidung, um welche Sprache es sich handelt, dient auch hier die Sprachabkürzung der jeweiligen Sprache, also z.B. "`de`" für deutsch, "`en`" für englisch, "`fr`" für französich etc.

Wird die Seite mit deinem Plugin in einer Sprache bertrieben, für die du keine "Messages" definiert hast, so wird automatisch die englische, bzw. wenn auch die nicht vorhanden ist, die deutsche Sprach-Datei eingebunden. Durch diesen "Fallback" wird auf der Seite wenigstens überhaupt ein Text an der entsprechenden Stelle angezeigt. Dieser "Fallback" gilt übrigens auch für alle bisherigen Angaben für eine bestimmte Sprache wie z.B. Modul-Namen etc.

Das Sprach-Tag bindet die sogenannten "Messages" in der richtigen Sprache ein. Sollte es keine "Messages" in der entsprechenden Sprache geben, werden die englischen, bzw. deutschen "Messages" eingebunden.

Damit die Messages korrekt angezeigt werden, müssen die Sprach-Dateien unbedingt im Format UTF-8 gespeichert werden. Dein Editor sollte eine Möglichkeit bieten, die Dateien im entsprechnden Format zu speichern. Aufpassen musst du evtl beim Editieren solcher Dateien, da nicht alle Editoren beim öffnen der Datei das Format korrekt erkennen.

Sprach-Dateien MÜSSEN im Format UTF-8 gespeichert werden.

So, jetzt hast du den ersten Teil des Handbuchs geschafft. Schau dir jetzt am Besten nochmal in Ruhe die ganze XML-Datei an. Dabei werden dir wohl zwei Dinge auffallen. Zum Einen habe ich nicht alle Tags der XML-Datei erklärt. Dies liegt einfach daran, dass sie im Moment noch nicht genutzt werden, also erst in der Zukunft Verwendung finden. Zum Anderen wird es dir hoffentlich so gehen, dass dir die anfangs so unsäglich wirkende XML-Datei inzwischen schon fast so vertraut ist wie ein alter Bekannter. Oder nicht? Nein? dann lies das ganze Kapitel nochmal. Wenn es dann immer noch nicht klar ist hast du zwei Möglichkeiten:

1. Du stellst deine Fragen im Papoo-Forum, oder
2. du bist einfach ein eher praktisch veranlagter Mensch. Dann solltest du die nächsten Kapitel dieses Handbuchs lesen und hoffen, dass dir die Erleuchtung bei der praktischen Arbeit kommt.

Papoo und die Template-Engine Smarty

Du hast sicher schon festgestellt, dass Papoo mit einer sogenannten Template-Engine arbeitet. Template-Engines arbeiten grob gesprochen so, dass sie eine Vorlagen-Datei (eine Template-Datei) mit Inhalten füllen. Damit die Template-Engine weiss, wo etwas in die Vorlagen-Datei eingefügt werden muß, sind in der Vorlagen-Datei Platzhalter vorhanden. Diese Technik erlaubt es Vorlagen frei von programmier-technischen Konstrukten zu halten, sprich es ist nicht wie bei der "normalen" PHP-Programmierung, wo PHP-Anweisungen in HTML-Code eingebettet wird. Durch diese Trennung ist es relativ einfach Möglich, den gewünschten HTML-Code zu gestalten. Es kann dazu im Extrem-Fall sogar ein beiliebiges HTML-WYSIWYG-Tool eingesetzt werden. Das die Gestaltung mit einem solchen Tool dann doch nicht so einfach von der Hand geht, liegt daran, dass Template-Engines selbst noch logische Funktionen beinhalten (z.B. IF-Anweisungen oder Schleifen-Konstrukte). Wie auch immer. Template-Engines sind ein ideales Werkzeug um Programm-Code von Gestaltungs-Code zu trennen.

Papoo benutzt die Template-Engine "Smarty". Die deutsche Dokumentation zu Smarty findest du unter http://smarty.php.net/manual/de/. Dort findest du irgendwo im Download-Bereich auch eine Version zum herunterladen.

Ich möchte hier jetzt nicht lang und breit etwas zu Smarty erzählen, dass kannst du schließlich in der Dokumentation nachlesen. Ich möchte vielmehr zwei Grund-Templates etwas näher betrachten und zwar die beiden Grund-Templates für die Papoo-Frontend-Seiten und für die Papoo-Backend-Seiten.

Template für Papoo-Frontend-Seiten

Eine normale Seite des Frontends unterteilt sich in verschiedene Bereiche. Diese sind im Einzelnen: der Kopf, die linke Spalte, die mittlere Spalte, die rechte Spalte und der Fuss. In diesem Abschnitt geht es jetzt um das was in die mittlere Spalte komt, auf einer normalen Seite also z.B. die Artikel oder das Forum etc. Wie Module aussehen, also Elemente welche Bereichen zugeordnet werden können, erfährt du im nächsten Abschnitt.

Ein Template für eine Frontend-Seite sieht im einfachsten Fall so aus:

```
<div class="artikel">
        Hallo Welt !
</div>
```

Bindest du dieses Template ein, hättest du eine komplette Seite mit Kopf, Menü etc. und dem Text "Hallo Welt!" dort wo sich sonst die Artikel befinden, also in der mittleren Spalte. Du kannst das ja einmal ausprobieren. Wie du solche Seiten in das Frontend-Menü einbinden kannst habe ich ja schon im Kapitel über die XML-Datei unter "Die Menü-Tags" beschrieben. Damit du jetzt aber nicht wie wild blättern mußt, wiederhole ich das dort gesagt nochmal.

Im Backend unter "Menü verwalten->Menü-erstellen" unter "formlink" gibst du den folgenden Pfad ein: `plugin:pfad/templates/name_des_templates.html`. Du mußt den Pfad natürlich noch etwas anpassen indem du `pfad/` und `name_des_templates.html` durch deine Angaben ersetzt. Wichtig an der Sache ist lediglich das `plugin:` am Anfang und das die folgende Pfadangabe relativ zum Verzeichnis /plugins ist. Das Template des Test-Plugins für die Frontend-Seite lässt sich also mit `plugin:test/templates/test_front.html` einbinden.

Template für Module

Templates für Module müssen immer folgende Form haben:

```
{if $module_aktiv.mod_test_front}
        <!-- MODUL: test_front -->
        <div class="modul" id="mod_test_front">
                Hallo Welt,<br />
                dies ist das Test-Modul.
        </div>
        <!-- ENDE MODUL: test_front -->
{/if}
```

Das gesamte Modul ist in eine if-Abfrage gekapselt. Anhand dieser if-Abfrage wird entschieden, ob das Modul ausgegeben wird oder nicht. Wie das genau geschieht erfährst du im Kapitel IV "Eigene Klassen" im Abschnitt "Die Papoo-Klasse 'module_class'". Interessant an dieser Stelle ist nur die Frage "woher kommt der Variablen-Name `.mod_test_front` ?". Dieser Name entspricht dem Dateinamen des Moduls ohne die Endung .html. Deine eigenen Modul solltest du also nach folgender Konvention benennen: mod_modulname.html.

Das Modul selbst muss dann wiederum in ein div gekapselt sein. Dieses div muss die beiden Attribute `class="modul"` und `id="mod_modulname"` besitzen. Der Einfachkeit halber ist hier die `id` genau gleich benannt, wie der Modulname selbst. Du musst dir also nicht tausend verschiedene Namen einfallen lassen. Wähle einmal einen aussagekräftigen Namen aus und verwende diesen dann konsequent.
Das das Modul der Klasse `"modul"` angehört nimmst du am Besten einfach so hin. Diese Zuweisung sorgt dafür, dass das Modul sich "korrekt" verhält.

Template für Papoo-Backend-Seiten

Zu Templates für das Backend ist eigentlich nicht viel mehr zu sagen als ich schon zu denen für das Frontend gesagt habe. Auch dort ist schon entscheidende Vorarbeit für die wiederkehrenden Elemente erbracht worden. Ein Template für das Backend sieht im Grundgerüst dann so aus:

```
{include file=head.inc.html}
{include file=menu.inc.html}

<div class="artikel">

</div>

</body></html>
```

Der einzige Unterschied zu Templates für das Frontend ist, dass du den Kopf und das Menü "von Hand" einbinden, und am Ende `</body>` und `</html>` angeben musst.
Die Einbindung der Seite in's Backend geschieht ausschließlich über die XML-Datei des Plugins. Was dafür zu tun ist kannst du im Kapitel II "Die XML-Datei" im Abschnitt "Die Menü-Tags" nachlesen. Hier möchte ich nur noch erwähnen was zu tun ist, wenn du während deiner Entwicklung feststellst, dass du eine zusätzliche Seite brauchst.

Schreib dir als erstes das entsprechende Template für die neue Seite. Einfaches kopieren des Grundgerüsts reicht oftmals schon. Danach solltest du im Plugin-Manager dein Plugin deinstallieren. Nun kannst du in der XML-Datei deines Plugins ein neues Menü-Tag bzw. ein neues Submenü-Tag mit den entsprechenden Angaben anlegen. Jetzt das Plugin mit dem Plugin-Manager einfach wieder installieren und du hast einen neuen Eintrag im Backend-Menü welcher dich auf deine neue Seite führen sollte. Ist dabei irgendwas schief gelaufen, weil du z.B. den Pfad falsch angegeben hast, einfach das Plugin wieder deinstallieren, die Angaben korrigieren und das Plugin wieder installieren. Dieses Installieren und Deinstallieren wirst du während deiner Entwicklung warscheinlich noch ziemlich häufig machen. Sollten dabei mal richtig krasse Probleme auftreten wird dir das Plugin "Entwickler-Werkzeuge" sicher hilfreiche Dienste erweisen. Die "Entwickler-Werkzeuge" möchte ich jetzt aber nicht weiter beschrieben. Sie haben ein eigenes Kapitel in diesem Handbuch, dem du alles wissenswerte entnehmen kannst.

Bilder in Templates einbinden

Die Einbindung von Bildern in Templatest stellt ein gewisses Problem dar. Der Grund liegt an der nicht ganz so einfachen Angabe des Pfades zum Bild. Soll z.B. das Bild "testbild.jpg" des Test-Plugins in das Template der Frontend-Seiten eingebunden werden, so lautet der Pfad zum Bild: `plugins/test/bilder/testbild.jpg`. Dies liegt daran, dass dein Browser die Seite ausgehend vom Wurzel-Verzeichnis betrachtet. Von diesem Verzeichnis ausgehend liegt das Bild dann eben im /plugins-Verzeichnis (/plugin), weiter in das Verzeichnis des Plugins (/test), dort in das Bilder-Verzeichnis (/bilder) zum Bild (/testbild.jpg). Für Backend-Templates muß diesem Pfad ein ../ vorangestellt werden, da der Pfad dort relativ zum Verzeichniss /interna angegeben werden muß. Also ein Verzeichnisse nach oben (../) in das Verzeichnis /plugins (/plugins) etc.

Die Problematik der verschiedenen Pfad besteht übrigens auch für die CSS-Datei deines Plugins, wobei sie sich dort sogar nochmal um eine Nummer verschärft. Gibst du dort z.B. ein Hintergrundbild für ein Element an, so muß der Pfad zu diesem Bild wie oben beschrieben angegeben werden, wobei du jedesmal noch ein "../", also ein "Verzeichnis nach oben" zusätzlich angibst. Das liegt daran, dass die CSS-Datei eigentlich im Verzeichnis /templates_c bzw. /interna/templates_c liegt. Richtig doof ist es dann, wenn du ein CSS-Element für Frontend und Backend hast, da dort die Pfad unterschiedlich sind. Unterscheide die beiden Elemente dann am Besten durch Angabe einer id und hänge die Information des Hintergrundbildes an die id an.

Du kannst dir das auch in den Dateien des Test-Plugins anschauen. Dort habe ich mal ein Bild in das Template der Frontend-Seite (test_front.html) eingebaut. Außerdem ist dieses Bild auch in der CSS-Datei als Hintergrundbild angegeben. Die Seite sieht zwar jetzt ziemlich bescheiden aus, sollte aber möglichst viele Aspekte erklären.

Eigene Klassen

Endlich das Thema was dich wirklich interessiert. Du willst ja schließlich programmieren und dich nicht nur durch öde Handbücher wälzen. Habe ich recht? Genau so geht es mir zumindest oft. Inzwischen bin ich aber zur Erkenntnis gekommen, dass ein gutes Handbuch einem die Sache erheblich erleichtern kann. Also wälze noch eine Weile weiter. Parallel dazu kannst du aber in diesem Kapitel auch mal den ein oder anderen Programmierversuch einbauen um das von mir gesagte noch besser zu verstehen.

Wie die Überschrift schon verrät, soll es hier um Klassen gehen. Du mußt deinen Code also Objekt-Orientiert schreiben, sonst hast du bei der Entwicklung eigener Papoo-Plugins keine Chance. Falls du darin noch nicht fit bist, eigne dir wenigstens mal die Grundlagen dazu an. Ich kann dir versprechen, so schwierig ist das Ganze nicht und hast du es erstmal so halbwegs kapiert, wirst du feststellen das es einige Vorteile mit sich bringt.

Grundgerüst eigener Klassen

Die PHP-Dateien deiner eigenen Klassen sollten stehts folgende Struktur aufweisen:

```
<?php

class testplugin_class
{
	function testplugin_class()
	{
	}
}

$testplugin = new testplugin_class();

?>
```

Das einzig Besondere ist die letzte Zeile. Hier findet die im Kapitel "Die XML-Datei" unter dem Punkt "Die Klassen-Tags" angesprochene Instanzierung deiner Klasse statt. Es wird also ein konkretes Objekt der Klasse angelegt, wobei dieses Objekt den Namen trägt, welcher in der XML-Datei unter dem Tag `<klasse>` im Tag `<name>` angegeben werden muß.

Du solltest pro Klasse eine eigene PHP-Datei anlegen. Das sollte kein Hinderniss darstellen, da du in der XML-Datei ja beliebig viele Klassen einbinden kannst.

Kurz und schmerzlos war's das auch schon was ich zum Grundgerüst eigener Klassen zu sagen habe.

Eigene Template-Variablen

Als nächsten Punkt möchte ich dir zeigen, wie du deine eigenen Variablen an die Templates durchreichst. Dazu erweitern wir das vorige Grundgerüst einer Klasse um einige Einträge:

```
class testplugin_class
{
	function testplugin_class()
	{
		// Einbindung des globalen Content-Objekts
		global $content;
		$this->content = & $content;

		// Exemplarische Wert-Zuweisung an die Content-Klasse
		// ************************************************
		// 1. einen einfachen Text zuweisen
		// Zuweisung eines Textes an eine Variable
		$test_text = "Hallo Welt!";
		$this->content->template['plugin']['test']['text_01'] = $test_text;
	}
}
```

Der erste Teil dieses Code-Beispiel bindet die Papoo-eigene Klasse Content ein. Dazu findest du mehr Informationen im Abschnitt "Einbindung Papoo-eigener Klassen". Ich möchte mich hier zunächst nur mit dem zweiten Teil des Code-Beispiels beschäftigen. Dort wird zunächst einer PHP-Variablen `$test_text` ein

Text zugewiesen. Nichts ungewöhnliches also. Anschließend wird diese Variable an das Content-Objekt übergeben. Die geschieht, indem die Variable in das Array `template` mit dem Namen `['plugin']['test']['text_01']` geschrieben wird.

Um Variablen an Templates zu übergeben werden diese im Content-Objekt in das Array "template" geschrieben.

Exemplarisch hier nun noch die Einbindung deiner Variable in das Grundgerüst des Backend-Templates:

```
{include file=head.inc.html}
{include file=menu.inc.html}

<div class="artikel">
        {$plugin.test.text_01}
</div>

</body></html>
```

Du schreibst also einfach an der Stelle im Template, an welcher deine übergebene Variable erscheinen soll, den Namen der Variablen mit dem du sie an das Content-Objekt übergeben hast, umgeben von geschweiften Klammern und mit einem $-Zeichen davor.

Wie ganze Arrays an Templates übergeben werden, und wie diese in den Template-Dateien einzubinden sind, kannst du dem Test-Plugin entnehmen.

HTML-Entitäten-Auszeichnung in der Content-Klasse (Änderungen in Papoo Version 3.5)

Im Gegensatz zu früheren Versionen, wandelt die Content-Klasse von Papoo Version 3.5 Umlaute und sonstige Sonderzeichen **nicht** mehr in HTML-Entitäten um. Dies ist für eine korrekt Ausgabe auch nicht (mehr) nötig.
Einzig Zeilenumbrüche werden nach wie vor in
 umgewandelt. Soll auch diese Umwandlung nicht stattfinden, z.B. wenn eine Variable Text enthält welcher in einem <textarea> bearbeitet werden soll, dann ist bei der Übergabe des Textes an die Content-Klasse ein `"nobr:"` vor den Text zu schreiben.
Konkret wird also aus:

```
$this->content->template['mein_text'] = $text
```

Folgendes:

```
$this->content->template['mein_text'] = "nobr".$text;
```

Am Besten du spielts einfach mal ein bisschen damit herum. Du wirst dann recht schnell merken was geht und was nicht.

Die Sache mit den Namen

So langsam wird es Zeit einige Worte zu Namens-Konventionen zu verlieren. Wie du den beiden vorigen Abschnitten entnehmen kannst, habe ich mich dazu entschieden Klassen, Objekten und Content-Variablen immer ein Präfix voranzustellen. Der Einfachkeit halber verwende ich dazu meist den Plugin-Namen. Dieser Konvention solltest du bei deiner Entwicklung ebenfalls folgen. Solltest du das nicht tun, wären Konflikte mit anderen Plugins oder gar mit Elementen von Papoo selbst vorprogrammiert, also

verwende bei Namen von Klassen, Objekten und Content-Variablen stets ein eindeutiges Präfix.

Für die Bezeichnung der Conten-Variablen gehe ich sogar noch einen Schritt weiter. Ich kapsel alle Copntent-Variablen in einer Arrray-Struktur, also z.B. statt:

```
$this->content->template['mein_text'] = $text
```

verwende ich inzwischen lieber etwas in der Art:

```
$this->content->template['plugin']['mein_plugin']['mein_text'] = $text
```

Das macht die Dinger zwar etwas länger, bietet aber einige Vorteile. So ist es mit solchen Namenskonventionen z.B. möglich mir alle Content-Variablen meines Plugins für Testzwecke auszugeben. Ein einfaches:

```
print_r($this->content->template['plugin']['mein_plugin']);
```

reicht dazu.

Bei Content-Variablen welche in Message-Dateien definiert werden, gehe ich sogar noch einen Schritt weiter. Diesen Variablen stelle ich zusätzlich noch ein "message" voran, also z.B. so:

```
$this->content->template['message']['plugin']['mein_plugin']['meine_variable'] = "LALA";
```

So kann ich im Template schnell sehen welche Variable aus einer Message-Datei kommt und welche per PHP erzeugt wird.

Es ist zwar keine Pflicht Content-Variablen auf diese Weise auszuzeichnen, ich möchte euch aber darum bitten diese Vorgaben zu beherzigen. In Papoo selbst, also dem Kern von Papoo, werden diese Konventionen zwar (noch) nicht berücksichtig, irgendwann wollen wir das aber mal einführen. Grund für diese recht strickte Vorgehensweise ist, dass wir versuchen die Content-Variablen übersichtlicher zu strukturieren. Geplant ist auch irgend wann mal eine "Schnittstelle" für sämtliche Message-Texte zu haben, sodass Anpassungen für weitere Sprachen einfacher gemacht werden können. Ist zwar im Moment noch "Zukunftsmusik", aber.. kommt Zeit, kommt Rat.

Datei-Formate (NEU in Papoo Version 3.5)

Mit der Umstellung von Papoo auf "UTF-8" ergeben sich auch einige Konsequenzen für das Datei-Format der Plugin-Dateien. In den älteren Papoo-Versionen wurden einfach alle Dateien im Format "ISO-8859-1" (auch bekannt als "Latin-1") gespeichert. Seit Papoo Version 3.5 müssen jedoch einge Dateien im Format "UTF-8 (No BOM)" gespeichert werden. Im Einzelenen sind das:

- sämtliche "Message-Dateien",
- die (beiden) SQL-Dateien, und
- Template-Dateien, wobei bei diesen das Format UTF-8 nicht zwingend ist.

Du solltest dir also unbedingt anschauen, wie du mit deinem bevorzugten Editor Dateien in diesem Format öffnen und speichern kannst. Eigentlich könntest du auch sämtliche Dateien im Format UTF-8 speichern. Wir haben uns aber bewusst dagegen entschieden, da es dabei ab und an zu Problemen kommen kann. Speichere also deine PHP-Dateien (also deine Klassen-Dateien) bitte nach wie vor im Format ISO-8859-1. Die "Message-Dateien", also die Dateien in deinem message-Verzeichnis (beim Test-Plugin z.B. die Dateie "/messages/messges_frontend_de.inc.php") müssen aber unbedingt im Format UTF-8 gespeichert werden.
Die beiden Datenbank-Dateien müssen ebenfalls in diesem Format gespeichert werden. Wenn du diese Datei mit Hilfe des Entwickler-Plugins erstellst, ist sie schon im richtigen Format. Du musst dann lediglich beim erstellen der De-Installations-Datei aufpassen, die automatisch erstellte Datei im richtigen Format zu öffnen und zu speichern. Mehr dazu am Ende des Handbuchs in Kapitel V.

Bei den Template-Dateien habe ich oben geschrieben, dass du diese nicht unbedingt im Format UTF-8 speichern musst. Der Grund liegt darin, dass diese Dateien "normalerweise" keine Umlaute oder sonstige Sonderzeichen enthält. Schreibst du allerdings eine solche Datei "Quick'n Dirty", also ohne die Möglichkeit diese für verschiedene Sprachen zu benutzen, dann solltest du diese Dateien ebenfalls im Format UTF-8 speichern. Ansonsten werden Umlaute etc. nicht korrekt ausgegeben. Alternativ dazu, kannst du hier Umlaute etc. auch direkt als HTML-Entitäten auszeichenen, sprich du schreibst hier z.B. ä statt ä.
Speicherst du Template-Dateien im UTF-8 Format nöchte ich dich bitte dies auch im Dateinamen ersichtlich zu machen. Papoo verwendet auch einge solche Dateien, welche dann so benamt werden:

eine_utf8_datei**.utf8**.html

Durch eine solche Benamung ist es für dich, oder für jemand anderen der dein Plugin bearbeitet, sehr viel einfacher das korrekte Datei-Format zu ermitteln.

Einbindung Papoo-eigener Klassen

Wie im Abschnitt "Eigene Content-Variablen" versprochen jetzt einige Erläuterungen zur Benutzung Papoo-eigener Klassen. Unter Papoo-eigenen Klassen verstehe ich Klassen welche von Papoo zur Verfügung gestellt werden. Grundsätzlich kannst du jede Klasse die Papoo selbst benutzt auch in deinem Plugin verwenden. Die Einbindung der Content-Klasse geschieht dabei durch folgende Anweisungen:

```
global $content;
$this->content = & $content;
```

Es wird also nicht die tatsächliche Klasse benutzt sondern ein globales Objekt der Klasse eingebunden. In diesem Beispiel ist dass das Objekt `$content`, welches von der Klasse `content_class` ist. Sämtliche Papoo-eigenen Klassen findest du, falls du das nicht schon selbst herausgefunden hast, im Verzeichnis /lib/classes. Da Objekte im Gegensatz zu Klassen nicht perse global sind, mußt du die PHP-Anweisung `global` verwenden, um das gemeinte Objekt anzusprechen. So viel zur ersten Zeile.

Die zweite Zeile erweitert nun deine eigene Klasse um eine Referenz auf das Objekt. Warum eine Referenz? Würdest du in der zweiten Zeile einfach `$this->content = $content;` schreiben, also das Kaufmanns-Und weg lassen, so hättest du in deiner eigenen Klasse zwar auch ein Objekt `$content`, dies wäre aber eine komplette Kopie des globalen Objektes. In der Konsequenz würde das bedeuten, dass du diesem Objekt natürlich auch Content-Variablen zuweisen könntest, diese aber nicht an die Template-Engine weiter gegeben werden. An die Template-Engine werden nur die Variblen des globalen Objekts übergeben, nicht aber die deiner Kopie des Objekts. Alles klar? Wenn ja, gut. Wenn nicht, auch egal. Mach es einfach genau so und du bist auf der sicheren Seite.

Einige versiertere unter euch Entwicklern könnten jetzt noch behaupten, die zweite Zeile wäre eigentlich unnötig. Stimmt kann ich da nur sagen. Prinzipiell würde es auch ohne die zweite Zeile funktionieren in dem du in deinem Code einfach immer `$content` statt `$this->content` benutzt. Was passiert aber, wenn das globale Objekt eines Tages umbenannt wird? Dann mußt du hergehen und deinen ganzen Code umschreiben. Hast du das Ding vorher aber deiner eigenen Klasse per `$this->content = & $content` zugewiesen, müsstest du nur diese eine Zuweisung anpassen und fertig. Wie du also siehst, hat auch das seinen Sinn. Jetzt aber genug zu diesem Thema, obwohl es ja schon erstaunlich ist wieviel mir zu zwei simplen Zeilen Code so einfällt. Wir halten einfach mal fest:

Papoo-eigene Klassen werden mit den folgenden beiden Zeilen in eigene Klassen eingebunden:

```
global  $name_des_objekts;
$this->name_des_objekts  =  &  $name_des_objekts;
```

Einbindung eigener Klassen

Die Einbindung eigener Klassen in eine eigene Klasse geschieht analog zu der im vorigen Abschnitt beschriebenen Technik. Anzumerken ist dazu noch Folgendes. Verwendet eine Klasse A die Klasse B, so muß die Klasse B in der XML-Datei vor der Klasse A aufgeführt werden.

Prinzipiell wäre es auch möglich Klassen anderer Plugins zu benutzen. Aus zwei Gründen ist davon aber abzuraten. Erstens müßten die Plugins in der richtigen Reihenfolge installiert werden. Der Grund dafür ist der Gleiche wie im oben aufgeführten Gedankenspiel zur Benutzung der Klasse A durch die Klasse B. Zweitens würde dein Plugin, welches eine Klasse eines anderen Plugins nutzt, ja nicht mehr funktionieren, wenn das andere Plugin nicht mehr installiert ist. Lass sowas also einfach bleiben, oder mach das nur, wenn du dir dessen absolut sicher bist (z.B. bei Plugins welche du speziell für einen Kunden entwickelst).

Eine Aktionsweiche

Was soll das den jetzt schon wieder? Eine Aktionsweiche? Sinn und Zweck einer Aktionsweiche ist schnell geklärt. Stell dir einmal vor, dein Plugin würde sehr rechenintensive Arbeiten erledigen. Das Ergebnis dieser Arbeit wird jedoch nur in deinen eigenen Seiten benötigt. Da wäre es doch ziemlich blödsinnig diese Arbeit bei jedem beliebigen Seitenaufruf erledigen zu lassen. Aus diesem Grund ist es sinnvoll eine Aktionsweiche in deine Klasse einzubauen. Diese Weiche sorgt dafür, dass nur die für die entsprechende Seite benötigten Arbeiten auch tatsächlich erledigt wird. Diese Technik wird übrigens auch von Papoo selbst eingesetzt. Du kannst dir dazu ja mal kurz die Datei all_inc_front.php im Wurzel-Verzeichnis von Papoo anschauen. Dort gibt es ein `switch`-Konstrukt welches eben eine solche Aktionsweiche darstellt. Damit es für dich jetzt ein bisschen einfacher wird, hier mal der Prototyp einer einfachen Aktionsweiche:

```
function make_test()
{
	global $template;

	if (strpos("XXX".$template, "test_back.html"))
	{
		$this->aufruf_externe_klassenfunktion();
	}
}
```

Die ganze Weiche liegt innerhalb einer Klassen-Funktion make_test(). Diese Funktion wird vom Konstruktor der Klasse aufgerufen. Den Aufruf kannst du dir in der Datei /plugins/test/lib/testplugin2_class.php genauer anschauen.

Die Weiche funktioniert so, dass die Aktion `$this->aufruf_externe_klassenfunktion()` nur aufgerufen wird, wenn die aktuell aufgerufene Seite auf dem Template "test_back.html" beruht. Das Feststellen, ob es sich um die gewünschte Seite handelt, geschieht durch Vergleich der globalen Variablen `$template` mit dem Namen des Templates bei dem die Aktion ausgeführt werden soll. Die aufgerufene Funktion selbst verwendet übrigens eine externe, eigene Klasse, womit du die Bestätigung für den vorigen Abschnitt auch gleich mitgeliefert bekommst.

Kurz noch zu dem `if`-Statement. Die Aktion soll ja nur aufgerufen werden, wenn das aktuell verwendete Template `"test_back.html"` ist. In der gobalen Variablen könnte ja jetzt genau `"test_back.html"` stehen. Damit wäre `strpos($template, "test_back.html")` gleich `0`, da der String an der Position `0` beginnt. Da `0` als `false` intepretiert wird, hätten wir ein Ergebnis, was wir so nicht wollen. Aus diesem Grund habe ich den String `$template` einfach am Anfang um ein paar beliebige Zeichen erweitert. Somit beginnt der gesuchte String `"test_back.html"` sicher nicht an der Position `0` und die `if`-Abfrage bringt das erwünschte Ergebnis. So, jetzt aber genug aus dem Nähkästchen geplaudert.

Eine weitere Möglichkeit, bzw. ein anderer Parameter, auf dem eine Aktionsweiche beruhen kann, ist der global definierte Wert `admin`. Es handelt sich hierbei um keine Variable, sondern um einen per `DEFINE` definierten Wert. Dieser existiert immer dann, wenn du dich im interna-Bereich, also im Backend von Papoo befindest. Ein Aktionsweiche die diesen Paramter benutzt, könnte z.B. so aussehen:

```
if (defined(admin))
{
	// hier die Funktionen aufrufen, welche im Backend benötigt werden
}
else
{
	// hier die Funktionen für das Frontend unterbringen
}
```

Du kannst diese beiden Möglichkeiten der Aktionsweichen natürlich beliebig mischen und dir auch sonst noch so einiges dazu einfallen lassen, wann den welche Aktion ausgeführt werden soll. Prinzipiell gilt:

Aktionen eines Plugins, sollten zur Einsparung von Resourcen und Rechenleistung, über eine Aktionsweiche gesteuert werden.

Parameter-Übergabe

Die wohl häufigste Art und Weise eine Aktionsweiche zu steuern, wird die sein, auf einen per POST oder GET übergebenen Parameter zu reagieren. Als Beispiel: Du hast im Backend deines Plugins irgendwelche Werte die der User beeinflußen kann wie z.B. ein bestimmter Zahlenwert oder ein Text. Dieser Wert wird in einem Formular-Input-Feld zum Ändern zur Verfügung gestellt und soll dann nach einer Änderung gespeichert werden. Einen solchen Wert wirst du wohl üblicherweise nach der Übergabe aus den PHP-Variablen `$_GET` oder `$_POST` abholen und weiter verarbeiten. Dazu mal ein Ausschnitt der Template-Datei `test_back.html`:

```
<form name="test_daten" method="post" action="">
        <fieldset>
                <legend>Testwert</legend>
                <label for="testwert">Einen Testwert per POST übergeben</label>
                <input type="text" name="testwert" value="{$test_testwert}" size="25" /><br />
                <input type="submit" name="submit" value="submit" />
        </fieldset>
</form>
```

An diesem Formular wird dir unter Anderem folgendes auffallen. Zum Einen wird das Formular per POST, also mit `method="post"`, übergeben was weiter nicht ungewöhnlich ist aber stets so gemacht werden sollte. Warum erfährst du gleich. Zum Anderen ist die Formular-Action mit `action=""` angegeben. Das hat den einfachen Grund, dir eine Menge Schreibarbeit zu ersparen. Es wird durch die Angabe von `action=""` die aktuelle Seite aufgerufen. Du könntest das auch erreichen indem du `action="plugin.php?template=plugins/test/templates/test_back.html"` angibst. Tja und wie du messerscharf erkennst ist das einfach viel zu viel Schreibarbeit. Einen weiteren entscheidenden Nachteil hätte diese Art des Vorgehens. Streng genommen müsstest du nämlich `action="plugin.php?menuid=1006&template=plugins/test/templates/test_back.html"` angeben. Der Unterschied ist die Angabe der menuid. Diese Menü-ID kannst du ja aber gar nicht kennen, da sie automatisch beim Installieren des Plugins generiert wird und bei jedem, der dein Plugin installiert, anders lauten kann. Jetzt kommt auch die Sache ins Spiel, warum du Formular-Daten immer per POST übergeben solltest. Würdest du GET benutzen, würden deine Daten die aktuelle menuid überschreiben und das würde dann das Menü etwas durcheinander bringen. Du darfst das gerne mal ausprobieren und schauen was dabei passiert. Zusammenfassend ist also zu sagen:

Formular-Daten sollten immer mit der Methode POST übergeben werden. Die dabei verwendete Action sollte leer sein, so wird wieder die aktuelle Seite mit den richtigen Parametern aufgerufen.

So übergebene Werte könntest du nun also in deiner Klasse mittels `$_POST` wieder abrufen. In Papoo gibt es dazu aber eine eigene Klasse die dir das abnimmt. Du findest deine Werte also nicht (nur) in `$_POST` sondern zusätzlich in der Klasse checked (bzw. im globalen Objekt Namens $checked der Klasse checked_class, aber genug der Spitzfindigkeiten). Abrufen kannst du den Wert per `checked->test_testwert`, also als Attribut oder Variable des Objekts `$checked`. In all seiner Pracht sieht das nach der Einbindung des globalen Objekts `$checked` in die Klasse `testplugin2_class` also so aus: `$this->checked->test_testwert.` Eine mögliche Aktionsweiche sieht dann z.B. wie folgt aus:

```
if (strpos("XXX".$template, "test_back.html"))
{
        $this->aufruf_externe_klassenfunktion();
        if ($this->checked->test_testwert)
        {
                echo "Die Seite wurde mit dem POST-Parameter test_testwert =".$this->checked->test_testwert."
aufgerufen";
        }
}
```

Das ist nun beileibe kein schönes Konstrukt, sollte aber das Prinzip der Sache ganz gut veranschaulichen. Wir merken uns also:

Übergebene Werte eines Formulars werden vom globalen Objekt `$checked` per `$this->checked->name_des_formularwertes` zur Weiterverarbeitung abgeholt.

Die Papoo-Klasse "module_class"

Verwendest du in deinem Plugin eigene Module, will ich dir hier kurz etwas zur Module-Klasse erzählen.

Wie bereits im Abschnitt "Template für Module" des Kapitels III "Papoo und die Template-Engine Smarty" angedeutet, besitzt diese Klasse eine für dich sehr interessante Variable `$module_aktiv`. Diese Variable ist ein assoziatives Array mit den Namen aller aktiven Module. Bevor du jetzt aber völlig verzagst, bindest du am besten diese Klasse in deine eigen Klasse ein und zwar so, wie im Abschnitt "Einbindung Papoo-eigener Klassen" beschrieben:

```
global $module;
$this->module = & $module;
```

Jetzt kannst du dir per `print_r($this->module->module_aktiv)` diese Variable ausgeben lassen. Ist das Test-Plugin installiert, und hast du die beiden Module des Test-Plugins im Modul-Manager einem Bereich zugewiesen, findest du dort unter anderem folgende Einträge:

```
[mod_test_front] => 1
[mod_testfix_front] => 1
```

Du kannst diese Angabe jetzt also z.B. in deiner Aktionsweiche abfragen. Findet sich dein Modul in der Liste, oder aus Programmier-Sicht `if ($this->module->module_aktiv['mod_modulname'])`, dann weisst du, dass das Modul im Modul-Manager auch tatsächlich plaziert wurde. Wurde es nicht platziert, muss dein Plugin evtl. auch keine weiteren Aktionen ausführen, also:

Per `if ($this->module->module_aktiv['mod_modulname'])` kann abgefragt werden, ob ein bestimmtes Modul auch tatsächlich platziert wurde.

Mit der Variablen lässt sich aber noch mehr machen. Ist das Modul platziert, dein Plugin liefert aber gar keinen Inhalt für das Modul, kann das Modul über diese Variable auch komplett ausgeblendet werden.

Beipiel:
Zu schreibst ein Plugin "April-Scherz" welches am 1. April einen netten Spruch ausgeben soll. Dieses Plugin besitzt ein Modul mod_aprilscherz welches mit dem Modul-Manager auch tatsächlich platziert wurde. Da ja nur am 1. April etwas ausgegeben werden soll prüft dein Plugin das aktuelle Datum. Ist das Datum nicht der 1. April, blendet das Plugin das Modul per
`$this->module->module_aktiv['mod_modulname']=false;`
einfach komplett aus. Dadurch wird dann nicht mal mehr das div um das Modul ausgegeben.

Jetzt ist dir hoffentlich auch klar, warum im Template des Moduls das umgebende if-Konstrukt angegeben werden muss. Merke:

Per `$this->module->module_aktiv['mod_modulname']=false;` kann ein platziertes Modul komplett ausgeblendet werden.

Datenbank-Zugriffe

Papoo verwendet für Datenbank-Zugriffe die Datenbank-Abstraktions-Klasse ezSQL. Du solltest also keine Funktionen wie `mysql_query` oder so verwenden, sondern stattdessen immer mit dieser Klasse arbeiten. Zu der Klasse gibt es das globale Objekt `$db` welches du in deine eigene Klasse einbinden kannst. Wie ein solcher Aufruf aussehen kann soll anhand des folgenden Beispiels aus der Klasse testplugin2_class erläutert werden:

```
function read_from_db()
{
        global $db_praefix;
        $query = sprintf("SELECT test_text FROM `%s` WHERE test_id=1",
                        $db_praefix."test_testtabelle"
                        );
        $result = $this->db->get_results($query);
        $this->content->template['test_testwert'] = htmlentities($result[0]->test_text);
}
```

Das Grundprinzip ist wie beim "normalen" Umgang mit Datenbanken. Du bastelst dir deine SQL-Anweisungen und lässt diese Anweisung auf die Datenbank los. Für Papoo sind dabei nun folgende Gegebenheiten zu beachten.

Das Wichtigste ist die Berücksichtigung des Tabellen-Präfix, welcher sich bei jedem Benutzer unterscheiden kann. Dieses Präfix erhälts du aus der globalen Variablen `$db_praefix` welche du mit einem davorstehenden `global` einbinden kannst. Mit diesem Präfix kannst du nun deine SQL-Anweisung zusammenbauen. Ich persönlich verwende dazu meist die PHP-Funktion sprintf(). Das ist aber nur eine von mir bevorzugte Methode da sie, aus meiner Sicht, schon einige gute Möglichkeiten mit sich bringt sogenannt "SQL-Injections" abzufangen.

Deine SQL-Anweisung lässt du dann per `db->get_results(Anweisung)` auf die Datenbank los. Diese Funktion gibt dir das Ergebnis der Anweisung als Objekt zurück. Arbeitest du lieber mit Arrays, kannst du `get_results()` auch mit dem zusätzlichen, optionalen Parameter `ARRAY_A` aufrufen. Dieser Parameter sorgt dafür, dass du das Ergebniss als assoziatives Array zurück bekommst Der Aufruf selbst sieht dann etwa so aus: `db->get_results(anweisung, "ARRAY_A")`. Weitere Optionen zu ezSQL findest du (leider nur in englisch) unter: http://justinvincent.com/home/docs/ezsql/ez_sql_help.htm. Zusammenfassend gilt also:

Datenbank-Zugriffe sollten bei Papoo-Plugins über das Objekt `$db` gemacht werden.

Auf eine Sache möchte ich hier noch kurz hinweisen und dann hat sich das mit den Datenbank-Zugriffen auch schon erledigt. Verwendest du Texte, also Daten des Typs "string", in deinen SQL-Anweisungen, solltest du diese "escapen". Üblicherweise wird das mit `addslashes()` o.Ä gemacht. ezSQL bietet eine dazu äquivalente Funktione `escape()` welche statt dem üblichen `addslashes()` benutzt werden sollte. Wie das funktioniert siehst du in der Klasse `testplugin2_class` in der Funktion `write_into_db()`. Also kurz zusammengefasst:

Zugriffe sollten zum "escapen" von Texten die Funktion `db->escape(text)` verwenden.

post_papoo(), die spezielle Funktion zur Nacharbeit

Nachdem nun die wichtigsten Aspekte der Plugin-Programmierung besprochen sind, noch eine kleine Besonderheit zum Schluß. Es gibt bei Plugins die Möglichkeit eine post_papoo-Funktion zu definieren. Diese Funktion kann in jede beliebige Klasse durch einfaches Angeben einer solchen Funktion integriert werden. Das sieht dann so aus:

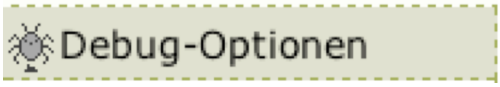
```
function post_papoo()
{
        // hier deine post_papoo-Anweisungen
}
```

Diese Funktion wird von Papoo aufgerufen, wenn alle Papoo Aktionen ausgeführt sind, also nach den Aktionen deines Plugins, nach den Aktionen von Papoo selbst aber vor der Zuweisung der Template-Variablen aus dem Objekt `$content` an die Template-Engine smarty. Diese Funktion ist immer dann sinnvoll, wenn du irgendwelche internen Dinge von Papoo, nachträglich beeinflußen willst, oder wenn du auf bestimmte Dinge von Papoo reagieren willst. Hört sich jetzt ziemlich abstrakt an. Ein kleines Beispiel welches du gerne mal nachvollziehen darfst, bringt hoffentlich etwas Licht in's Dunkel meines Geschwafels.

Mal angenommen du willst den Text "xx Besucher" beeinflußen, der z.B. auf der Startseite die Anzahl der bisherigen Besucher wiedergibt. Der Inhalt dieses Textes wird von Papoo gesetzt und wie du recht schnell feststellen wirst, steht er zu dem Zeitpunkt an dem dein Plugin abgearbeitet wird noch nicht zur Vefügung. Die Abarbeitung der Papoo-internen-Aktionen erfolgt eben erst nach der Abarbeitung der Plugin-Aktionen. Mithilfe einer post_papoo-Funktion könntest du diesen Text noch beeinflußen, da er zu dem Zeitpunkt wenn die post_papoo-Funktion aufgerufen wird zur Verfügung steht. Ich habe das mal exemplarisch in die testplugin2_class integriert um dir eine grobe Vorstellung des Ganzen zu geben. Das dabei herangezogene Beispiel ist natürlich ziemlich an den Haaren herbeigezogen. Alles was ich dir damit zeigen will, ist die Möglichkeit über post_papoo-Funktionen nachträglich in Dinge, die Papoo intern regelt, eingreifen zu können.

Das Plugin "Entwickler-Werkzeuge",

oder auch "devtools" wie es sich im PluginManager meldet. Dieses Plugin bietet dir für deine eigenen Entwicklungen einige nützliche Funktionen. Es ist während meiner Arbeit am Plugin-Manager und der Arbeit an meinen ersten Plugins entstanden. Warscheinlich werden im Lauf der Zeit noch einig Funktionen dazu kommen. Hier nur die Erklärung der Funktionen, welche für dich bei deiner Arbeit interessant sind.

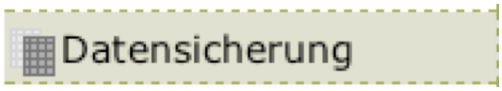


Hier stehen drei Funktionen zur Verfügung. Die Erste unterdrückt auf Wunsch die automatische Weiterleitung bei der Installation und Deinstallation von Plugins. Das ist ziemlich hilfreich um eventuelle Fehler in der XML-Datei zu finden.

Mit Hilfe der zweite Funktion kannst du sämtliche Einträge, welche beim Installieren eines Plugins in die Papoo-internen Datenbank-Tabellen angelegt werden, wieder löschen. Es hilft dir also immer dann, wenn du z.B. aufgrund einer fehlerhaften XML-Datei schon 23 Menü-Einträge deines Plugins im Backend-Menü hast und die Scheißdinger einfach nicht mer verschwinden wollen. Das ist wirklich extrem hilfreich. Ich hab mich da oft durch die verschiedenen Tabellen gehangelt und immer wieder die einzelnen Einträg von Hand gelöscht. Das macht echt keinen Spaß.

Die dritte Funktion ähnelt der ertsen Funktion, unterdrück aber sämtliche Weiterleitungen die in Papoo vorkommen. Das ist ziemlich hilfreich für Menschen wie mich, die am Kern von Papoo arbeiten

Mit dieser Funktion kannst du die Papoo-internen Cache-Dateien löschen. Das ist z.B. hilfreich bei der Entwicklung eigener Plugin-CSS-Dateien. So mußt du nicht jedesmal dein Plugin deinstallieren und anschließend wieder installieren um Änderungen deiner CSS-Datei zu übernehmen.



Diese Funktion bietet dir eine einfach Möglichkeit die Datenbank-Installations-Datei deines Plugins zu erstellen. Du wählst einfach die Tabelle oder die Tabellen, welche du für dein Plugin angelegt hast, und speicherst die automatisch erstellte Dump-Datei. Mit dem passenden Namen versehen und an der richtigen Stelle deines Plugins gespeichert, hast du so in Null-Komma-Nix die Datenbank-Installations-Datei erstellt. Wenn du die Zeile(n) "DROP TABLE.." dieser Datei kopierst und in einer neuen Datei abspeicherts, hast du übrigens auch RuckZuck die Datenbank-Deinstallations-Datei.

Diese beiden Dateien mußt du, wie bereits weiter vorne in deisem Handbuch erwähnt. im Format "UTF-8 (No BOM)" Speichern. Nur so werden Umlaute und sonstige Sonderzeichen korrekt in die Datenbank eingetragen.