P.M.S Perl - Messaging System

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Einleitung

Aufgabenstellung

Im Rahmen der Technickerausbildung im Schuljahr 2011/12 in der Klasse ITT 7/8 wurde die Aufgabe gestellt, eine in Perl realisierte Anwendung mit Datenbankanbindung zu planen, entwickeln und vorzustellen. Als Thema der Projektarbeit haben wir uns das Perl-Messaging-System, kurz P.M.S. ,einen Chatserver mit Client und Administrationsinterface, ausgesucht.

Dieses Dokument beschreibt die Funktionsweise und Implementation des Perl Messaging Systems, also den Core-Server, die Erweiterung durch Module, das Admin Interface und den Chat Client.

Die Arbeiten an der Software wurden gemeinschaftlich von Lukas Michalski, Benjamin Zeller und Thorsten Schwalb durchgeführt.

Zielsetzung Core-System

Das meisste Augenmerk lag hierbei auf dem Herzstück des PMS , dem Core.

Die Hauptziele bei der Impementation des Cores waren wie folgt:

1. Implementation eines Multi-Channel Multi-User Chatservers
2. Quasi parallele Verarbeitung mehrerer Connections
3. Schnelle Asynchrone Verarbeitung der Commands, durch Eventbasierte Programmierung
4. Erweiterbarkeit durch Modularisierung
    a. Erweiterbarkeit der Userfunktionen durch Plugins
    b. Erweiterbarkeit der Connectivity durch sogenannte Connection Provider
5. Implementation eines einfachen Chatprotokolls
6. Möglichst lose Objektkopplung zwischen den Plugins und dem Core

Alle dieser Ziele wurden bis zum Projektende in die Tat umgesetzt und liegen in der aktuellen Softwareversion vor.

Zielsetzung Erweiterungs-Module

Es sollten mehrere Erweiterungsmodule entwickelt werden um zu demonstrieren auf welche Weise es möglich ist den P.M.S. Server zu erweitern. Vor allem das Security Modul ist hierbei hervorzuheben, da es fast die komplette Palette der Möglichkeiten ausnutzt.

Die Ziele waren wie folgt:

1. Security

```
Rechteverwaltungssystem und registrierte User,
einteilung der User in Gruppen und zuweisung von Rechten
an die Gruppen
```

2. Statistik Modul

```
Statistiken zB über die Aufenthaltsdauer eines
Users, wieviele Messages er schreibt usw.
```

3. Backlog Modul

```
Die Messages eines Channels werden in der Datenbank hinterlegt
und werden an einen User gesendet der den Channel betritt, damit
dieser weiss was vorher gesprochen wurde.
```

Im Rahmen des Projekts wurden die Ziele wie folgt umgesetzt:

1. Security Modul

```
Bis auf die Möglichkeit Gruppen anuzlegen und zu verwalten
wurde das Modul vollständig umgesetzt
```

2. Backlog Modul

```
Wurde komplett umgesetzt
```

3. Statistik Modul

```
Konnte im Zeitrahmen des Projektes nicht umgesetzt werden
```

Zielsetzung HTML5 - Chat Client

Der Chat Client ist die Referenzimplementation eines Clients, basierend auf einer Kombination von Html und Javascript.

Die Ziele waren wie folgt:

1. Komplette Referenzimplementation des P.M.S. Chatprotokolls
2. Die Möglichkeit in mehreren Channeln gleichzeitig chatten zu können
3. Moderne und ansprechende HTML5 basierte Oberfläche

Alle dieser Ziele wurden bis zum Projektende in die Tat umgesetzt und liegen in der aktuellen Softwareversion vor.

Zielsetzung Admin-Interface

Das Admin Interface wird benötigt, um auf die von Modulen bereitgestellten Daten zugreifen zu können, oder diese zu konfigurieren. Die Implementierung erfolgte in Perl, Javascript und Html.

Die Ziele waren wie folgt:

1. HTML und Javascript basierte Oberfläche
2. Modularer Aufbau

3. Chatserver Security Modul Verwaltung
4. Chatserver Backlog Modul Verwaltung
5. Chatserver Statistik Modul Verwaltung

Im Rahmen des Projekts wurden die Ziele wie folgt umgesetzt:

1. HTML und Javascript basierte Oberfläche

   ```
   Komplett implementiert
   ```

2. Modularer Aufbau

   ```
   Wurde erfolgreich verwirklicht
   ```

3. Chatserver Security Modul Verwaltung

   ```
   Bis auf die Gruppenverwaltung implementiert
   ```

4. Chatserver Backlog Modul Verwaltung

   ```
   Konnte im Zeitrahmen des Projektes nicht umgesetzt werden
   ```

5. Chatserver Statistik Modul Verwaltung

   ```
   Konnte im Zeitrahmen des Projektes nicht umgesetzt werden
   ```

Chapter 1

Implementation, Funktionsweise und Design des Core-Systems

1.1 Allgemein

Die komplette Implementation des Cores erfolgte in Perl und wurde als reine objektorienterte, eventbasierte Software konzipiert. Der Core ist die zentrale Einheit des P.M.S. es benötigt bis auf einen ConnectionProvider keine Erweiterungsmodule um eine Basis Chat Funktionalität zu Verfügung zu stellen.

Bei der Implementation wurde auf ein modulares Design wert gelegt. Der Server ist somit beliebig erweiterbar. Die Module werden in der Konfigurationsdatei hinterlegt, und vom Server zur Laufzeit geladen und initialisiert.

Es gibt hierbei zwei Arten von Modulen:

ConnectionProvider	Der ConnectionProvider macht es möglich, den Core über beliebige Kommunikationswege zu verbinden, wie zB TCP-Sockets oder Websockets. Selbst ausgefallene Möglichkeiten wie zB HTTP-Server Push wären denkbar.
Plugins	Plugins sind eine Möglichkeit den Server um diverse Funktionalitäten zu erweitern wie z.B. einen Backlog oder selbst ein Chat-Bot (maschineller User) sind möglich.

Es wäre denkbar selbst das Chat Protokoll als Modul zu implementieren, dies war jedoch im Rahmen des Projekts nicht möglich.

MODUL BILD

1.1.1 Verwendete Fremdsoftware im Core

- AnyEvent (6.14)

```
Das AnyEvent Modul stellt den vom P.M.S. verwendeten Event-Loop
und einige Helferklassen bereit.Es ermöglicht eine sogenannte asynchrone-Programmierung,
diese versetzt den P.M.S. Core in die Lage mehrere Operationen quasi-parallel auszuführen.
So muss man zum Beispiel nicht darauf warten, bis eine Schreiboperation auf
einen Dateideskriptor beendet ist, sondern das System kümmert sich um dessen
Fertigstellung. Dabei kann man jedem asynchronen Aufruf diverse Callback Funktionen
mitgeben, die nach erfolgreicher Fertigstellung des Befehls, oder sogar im Fehlerfall
aufgerufen werden.
```

- Object::Event (1.23)

```
Dieses Modul stellt das sogenannte Observer-Pattern zur Verfügung,
das es uns ermöglicht interne Signale (Events) zu verschicken und zu verarbeiten.
Es ermöglicht dem Core komplett ohne das Wissen über eventuell geladene Module,
oder deren Voraussetzungen zu operieren.
```

1.2 Interne Schnittstellen

1.2.1 Signale

Signale und Callbacks werden von uns verwendet um Objekten eine Möglichkeit zu geben sich untereinander zu Verständigen, ohne jedoch zuviel vom eigentlichen Kommunikationspartner wissen zu müssen. Diese Schnittstellen-Technik ist auch als Observer-Pattern bekannt [1];.

Als Ausgangspunkt wird von uns das auch dem CPAN nachinstallierte Modul Object::Event verwendet. Da dies aber keinerlei Möglichkeiten hatte zu testen ob ein bestimmtes Signal eigentlich existiert oder nicht, musste die Klasse noch erweitert werden.

Alle Objekte die Signale versenden wollen, müssen vom Objekt Pms::Core::Object ableiten und in einem globalen Hash **%pmsEvents** alle von ihm versendeten Signale hinterlegen:

```
#!/usr/bin/perl -w

package MyPackage;
use Pms::Core::Object;
use strict;
use utf8;

our @ISA = qw(Pms::Core::Object);

our %PmsEvents = (
  'signal1' => 1,
  'signal2' => 1
);
```

Diese Signale können mit beliebigen Callbacks verknüpft und es können auch mehrere Callbacks pro Signal registriert werden. Die Callbacks werden in der gleichen Reihenfolge aufgerufen, wie sie beim Sender-Objekt registriert werden. Es ist ausserdem möglich Argumente mit an die Callbacks zu übergeben.

Beispiel

```
#Code im Sender Objekt:
sub anyOperation{
  #aulösen des Signals:
  $self->emitSignal('some_signal',"arg1","arg2");
}

#Code im Receiver Objekt:
sub createConnections{
  my $self = shift;
  my $signalSender = shift;

  $signalSender->connect(
    some_signal => $self->_callback()
  );
}

sub _callback{
  my $self = shift;
```

[1] http://de.wikipedia.org/wiki/Observer_(Entwurfsmuster)

```
  return sub{
    my $eventChain = shift; #Die Event Chain über die das Event abgebrochen werden kann
    my $arg1 = shift;
    my $arg2 = shift;
    #do something
  }
}
```

Beim connecten der Signale, wir automatisch in der kompletten Hirarchie des Objekts nach der Definition eines Signals im PmsEvents Hash gesucht. Somit können auch Signale von übergeordneten Klassen verwendet werden, ohne diese noch einmal im Hash angeben zu müssen.

1.2.2 Events

Jedes Modul kann sich für bestimmte Server Events registrieren, es hinterlegt dabei eine Callback-Funktionen die beim auftreten dieses Events ausgeführt wird. Server Events sind hierbei nichts weiter als Signale die von der Application Klasse versendet werden, mit der Besonderheit, das sie immer ein Event Objekt als ersten und einzigen Parameter mit schicken.

Beispiel

```
sub createConnections{
  my $self = shift;
  #Weist den Server an das Event client_connect_success mit dem Callback zu verbinden
  $self->{m_parent}->connect(
    client_connect_success => $self->_callback()
  );
}

sub _callback{
  my $self = shift;
  return sub{
    my $eventChain = shift; #Die Event Chain über die das Event abgebrochen werden kann
    my $eventType  = shift; #Das Event Objekt selbst ,das diverse Informationen enthält

    #Eine Message an den User schicken
    $eventType->connection()->postMessage(
      Pms::Prot::Messages::serverMessage("default","Hallo vom Callback")
    );
  }
}
```

Folgende Events stehen zur Verfügung:

Name	Beschreibung
client_connect_request	Ein neuer Client versucht sich zu verbinden
client_connect_success	Ein neuer Client hat eine Verbindung aufgebaut
client_disconnect_success	Ein Client hat die Verbindung geschlossen
message_send_request	Ein Client versucht eine Message zu senden
message_send_success	Ein Client hat eine Message gesendet
join_channel_request	Ein Client möchte einen Channel betreten
join_channel_success	Ein Client hat einen Channel betreten
leave_channel_success	Ein Client hat einen Channel verlassen
create_channel_request	Ein Client möchte einen Channel erstellen
create_channel_success	Ein Client hat einen Channel erstellt
channel_close_success	Ein Channel wurde geschlossen
change_nick_request	Ein Client versucht seinen Nickname zu ändern
change_nick_success	Ein Client hat seinen Nickname geändert
change_topic_request	Ein Client versucht ein Channel Topic zu ändern
change_topic_success	Ein Client hat ein Channel Topic zu geänderd
execute_command_request	Ein Client versucht ein Command auszuführen

Jedes Event, das über ein "_request" Postfix verfügt, kann in der Callback Funktion unter Angabe eines Grundes zurückgewiesen werden, die weitere Verarbeitung wird somit abgebrochen und der User bekommt eine entsprechende Meldung darüber. Hiermit wird es nicht nur ermöglicht, auf Events zu reagieren sondern diese sogar von ausserhalb des Cores zu steuern. Diese Funktionalität wird zum Beispiel im Security Modul verwendet um Userrechte zu überprüfen, allerdings könnte man damit auch einfachere Wortfilter erstellen, die zB Messages mit Schimpfwörtern herausfiltern.

Zurückweisung eines Events im Callback

```
sub _callback{
  my $self = shift;
  return sub{
    my $eventChain = shift; #Die Event Chain über die das Event abgebrochen werden kann
    my $eventType  = shift; #Das Event Objekt selbst ,das diverse Informationen enthält

    #Setzt die Error Message die an den Client gesendet wird
    $eventType->reject("Eine beliebige Error Message");

    #Stoppt das Event
    $eventChain->stop_event;
  }
}
```

1.3 Externe Schnittstellen

1.3.1 Allgemein

Wie in der Einleitung schon kurz erwähnt, ist die komplette Kommunikation im Core modular aufgebaut. Hierfür sind zwei abstrakte Klassen im Core vorhanden, die die Implementationsdetails der jeweiligen Kommunikationsbackends "verstecken".

- Pms::Core::ConnectionProvider

  Der ConnectionProvider ist die Abstrakte Form eines Server-Sockets, wie man ihn von der normalen TCP Programmierung kennt: Er nimmt Verbindungsversuche an und informiert den Server über ein Signal (connectionAvailable) darüber, das neue Connections vorliegen. Hierbei wird komplett abstrahiert wie dieser Ablauf im Hintergrund funktioniert. Ob dies nun ein HTTP Requst oder ein normaler TCP-Socket ist, ist für den Core komplett irrelevant.

- Pms::Core::Connection

  Die Connection Klasse hat zwei Aufgaben:

  1. Sie abstrahiert die darunterliegende Kommunikation
  2. Sie identifiziert den verbundenen User und weiss dessen Nicknamen

  Am besten kann man sich die Connection Klasse als einen Socket vorstellen, sie empfängt und schickt Daten entweder synchron oder asynchron, je nachdem welche Funktion verwendet wurde. Auch hier ist das darunterliegende Protokoll irrelevant. Unter Zuhilfenahme einer SessionId und eines Mailservers könnte man sogar, eine Kommunikation per EMail über diese Klasse realisieren.

Die verwendeten Backends werden über die Konfigurationsdatei festgelegt und zur Laufzeit geladen und initialisiert. Standardmässig sollten sie im Packet Pms::Net untergebracht sein, können aber durch Verwendung des FQN in der Konfigurationsdatei von überall geladen werden (solange sie sich im Include Pfad befinden)

1.3.2 Backend Implementation Websocket

Die Referenzimplementation des Backends ist auf dem Modul AnyEvent::Handle und dem WebSocket Protokoll aufgebaut, das es dem Server ermöglicht HTML5-WebSocket Connections entgegenzunehmen, somit kann jeder mit einem HTML5-fähigen Browser, ohne etwas installieren zu müssen, eine Verbindung zum Chatserver aufbauen.

Note
Im Rahmen der Projektarbeit, wurde nur der Chromium Browser getestet.

1.3.2.1 Verwendete Fremdsoftware

- AnyEvent::Handle

```
Ein Teil des AnyEvent Frameworks, das es ermöglicht Socket Handles mit unserer asynchronen
Programmierung zu verbinden. Anstatt in einem blockierenden System-Call auf neue Daten
zu warten, führt der EventLoop unsere Callbacks aus wenn neue Daten ankommen.
```

- AnyEvent::Socket

```
Ein Teil des AnyEvent Frameworks, das es ermöglicht einen TCP-Server mit unserer  ↩
    asynchronen
Programmierung zu verbinden. Anstatt in einem blockierenden System-Call auf neue  ↩
    Verbindungen
zu warten, führt der EventLoop unsere Callbacks aus wenn neue Daten ankommen.
```

- Protocol::WebSocket (0.00906)

```
Ein aus dem CPAN installiertes Modul, implementiert das Websocket Protokoll.
Hierbei wird ein einfacher HTTP-Handshake an den Server gesendet:

GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13

Der Server antwortet dementsprechend und wechselt von der HTTP
in die Socket Kommunikation. Danach kann man die Verbindung wie eine
normale TCP Verbindung verwenden.

Weiterführende Informationen:
http://de.wikipedia.org/wiki/WebSocket
```

1.3.2.2 Line Protokoll

Da man, wie bei jeder Stream basierten Verbindung (wie zB Tcp oder Websocket), nicht weiss wann eine Nachricht komplett angekommen ist oder ob man noch auf Daten warten muss, braucht man ein Protokoll das einem hilft Anfang und Ende der Nachrichten zu erkennen. Hierbei haben wir uns für Netstring, ein einfaches Textbasiertes Protokoll entschieden. Eine Nachricht im Netstring Format hat diese Darstellung:

```
16 : Das ist ein Text ,
```

Die Zahl am Beginn der Nachricht, beschreibt wieviel **Zeichen** sich zwischen dem beiden Delimitern : und , befinden. Hierbei ist vollkommen egal welche Zeichen, sogar ein weiterer Netstring könnte eingebettet werden oder sogar binäre Daten. Letzteres ist aber aufgrund von Einschränkungen durch das WebSocket Protokoll nicht möglich, da dies nur UTF-8 Character zulässt. Nun ist es ein leichtes für den Server zu erkennen wann eine Nachricht endet und wann er noch warten muss:

FLOWCHART Netstring

Weitere Informationen zu Netstring, findet man in der offiziellen Dokumentation: http://cr.yp.to/proto/netstrings.txt

1.3.3 Chat und Message Protokoll

Damit Client und Server richtig miteinander kommunizieren können, haben wir ein Chatprotokoll festgelegt, und einen eigenen handgeschriebenen Parser dafür entwickelt. Die Implementierung befindet sich im Pms::Prot::Parser Modul.

Der Aufbau einer Message hat immer den gleichen Aufbau:

```
/command "Hallo ich bin ein String" 12345
```

Folgende Regeln müssen zutreffen:

- Eine Message startet immer mit einem /
- Nach dem / muss sich ein Token befinden (Kommandoname),
- Ein Token startet immer mit einem Buchstaben und kann nur aus Buchstaben und Zahlen bestehen
- Alle Argumente sind duch Leerzeichen getrennt
- Ein Argument kann entweder ein String oder eine Nummer sein
- Ein String startet entweder mit einem " oder einem ’ , das gleiche Zeichen muss verwendet werden um den String zu beenden
- Kommt das Escape Zeichen im String noch einmal vor muss es mit einem \ escaped werden.
- Eine Nummer kann mit folgenden Zeichen beginnen: +,-,. oder einer Nummer

Wenn der Parser keinen Fehler findet, wird er einen Hash zurückgeben der den Kommandonamen (das Token nach dem /) und die Argumente als Array beinhält. Dieses wird dann an das Server-Objekt(Pms::Application) zur weiteren Verarbeitung weitergegeben.

1.4 Das Server Object

Das Herzstück, das Pms::Application Modul verbindet die bisher beschriebenen Funktionalitäten miteinander. Es sorgt dafür, das die ConnectionProvider und Module geladen und initialisiert werden.

Der Server ist ausserdem dafür zuständig, die einzelnen Connections anzunehmen und zu verwalten, Channels zu erstellen und zu löschen und die Messages richtig zu verteilen.

1.4.1 Empfang und Abarbeitung von Messages

Nachdem, der Server von einer der bestehenden Connections ein dataAvailable Signal erhalten ,die Daten ausgelesen und sie wie in den Kapiteln "Line Protokoll" und "Chat und Message Protokoll" verarbeitet hat, liegt ein Hash mit folgendem Aufbau vor:

```
(
  name => "Kommandoname",
  args => [
    arg0,
    arg1,
    #,....,
    argn
  ]
);
```

Der Server sucht nun in einem internen Hash, ob ein Callback für dieses Kommando hinterlegt wurde, und führt es aus. Als Parameter für den Callback wird immer als erstes Argument das Connection Objekt übergeben, das das Kommando ausführt, danach das Array mit den Argumenten aus dem Hash.

1.4.1.1 User-Schnittstelle (Kommandos)

Der Core Server kommt von Haus aus mit einigen vordefinierten Kommandos:

- **/send "channel" "message"**
  Eine Chatmessage, die an einen Channel geschickt wird
- **/join "channel"**
  Einen Channel betreten
- **/leave "channel"**
  Einen Channel verlassen
- **/create "channel"**
  Einen neuen Channel erstellen
- **/list**
  Eine Liste von bestehenden Channeln abfragen
- **/nick "newname"**
  Den eigenen Usernamen ändern
- **/users**
  Eine Liste von Usern erhalten (internes Kommando)
- **/topic "channel" "topic"**
  Die Überschrift/ das Thema eines Channels ändern

1.4.2 Erweiterungs API für Module

Eine weitere wichtige Aufgabe, ist die Bereitstellung einer API für die Erweiterung durch Module. Dazu gehören nicht nur die Events, die durch die Module beinflusst werden können, sondern auch eine Möglichkeit eigene "Chat-Kommandos" zu registrieren, die vom User oder der Clientapplikation aufgerufen werden können. Ein Modul muss dazu nur ein Kommando mit einer dazugehörigen Callback-Funktion registrieren, den Rest macht das Server Objekt von alleine:

```
sub createCommands{
  my $self = shift;
  my $srv  = shift; #Das Server Objekt

  $srv->registerCommand("ping",$self->_pingCallback());
}

sub _pingCallback{
  my $self = shift;
  return sub{
    my $connection = shift;
    $connection->postMessage(Pms::Prot::Messages::serverMessage("default","Pong"));
  }
}
```

Führt der User jetzt das Kommando **/ping** aus, wird der Server im "default" [2] Channel mit "Pong" antworten.

Das verarbeiten und stoppen von Events wurde im Kapitel Events schon beschrieben. Eigene Events zu registrieren ist jedoch nicht möglich,allerdings kann über das **execute_command_request** Event die Ausführung eines von einem Modul registrierten Kommandos beinflusst werden.

[2] Der default Channel ist das Hauptfenster in dem der Serverprozess Nachrichten schickt

1.4.3 Laden der Module

Das Server Objekt, lädt die Module zur Laufzeit, welche Module der Server lädt, in welcher Reihenfolge und welche Abhängigkeiten die Module untereinander haben, werden in der Server Konfigurationsdatei hinterlegt. Initialisierung und Shutdown der Module muss in den Konstruktor und Destruktor Funktionen durchgeführt werden, ansonsten werden keine Anforderungen an die API der Module erstellt. Der Server übergibt als erstes Argument an den Konstruktor immer eine Referenz auf sich selbst und als zweites Argument den Konfigurationshash des Moduls, dieser wird aus der Konfigurationsdatei gelesen.

Beispiel-Modul

```
#!/usr/bin/perl -w

package Pms::Modules::Ping;

use strict;
use Pms::Application;
use Pms::Prot::Messages;

sub new{
  my $class = shift;
  my $self  = {};
  bless ($self, $class);

  $self->{m_parent} = shift;     ❶
  $self->{m_config} = shift;     ❷
  $self->createCommands();

  return $self;
}

sub DESTROY{
  my $self = shift;
  $self->shutdown();
}

sub createCommands{
  my $self = shift;
  my $srv  = shift; #Das Server Objekt

  $srv->registerCommand("ping",$self->_pingCallback()); ❸
}

sub _pingCallback{       ❹
  my $self = shift;
  return sub{
    my $connection = shift;
    $connection->postMessage(
      Pms::Prot::Messages::serverMessage("default","Pong")
    );
  }
}

sub shutdown{
  my $self = shift;
  warn "Shutting Down";
}

1;
```

❶ Das ServerObjekt ist immer der erste Parameter

❷ Der Konfigurationshash des Moduls

3. Registrierung des **ping** Kommandos im Server
4. Callback Funktion für das **ping** Kommando

Chapter 2

Security Module

2.1 Allgemein

Um eine sinnvolle Anbindung an eine Datenbank und die Chatfunktion um Rechte zu erweitern wurde das Security Modul entwickelt. Das Security Modul führt zwei Arten von Rechten ein:

1. Globale Rechte

   ```
   Diese Art von Rechten legt fest ob ein User z.B. einen Channel erstellen,
   oder einen Channel löschen kann (das löschen wurde nicht mehr implementiert).
   Also Aktionen ausserhalb der Channel.
   ```

2. Channel Rechte

   ```
   Hier wird festgelegt welche Rechte ein User in einem bestimmten Channel hat.
   Es gibt pro User und Channel ein eigenes Rechtepacket, oder falls nichts in
   der Datenbank hinterlegt ist wird ,je nach Art des Channels, ein Default Packet erstel
   ```

2.2 Verwendete Fremdsoftware

- AnyEvent::DBI

```
Ein aus dem CPAN installiertes Modul. Es ermöglicht eine Asynchrone Verbindung
zur Datenbank aufzubauen. Datenbankoperationen die sehr lange dauern,
können so nicht den Mainloop blockieren und der Server kann eingehende
Events schneller verarbeiten.
Da Perl echte Threads fehlen, startet das AynEvent::DBI Modul kleine Hilfsprozesse,
die die Queries entgegennehmen und ausführen, ist die Datenbankoperation fertig
wird im Hauptprogramm ein mitgegebenes Callback ausgeführt.
```

2.3 Funktionsweise

Das Security Modul nützt alle Möglichkeiten des Core Servers um ihn zu erweitern. Es registriert diverse Befehle und behandelt fast alle Events die der Server schickt. Nachdem ein User erfolgreiche eine Verbindung zum Server aufgebaut hat, bekommt er ein Standard Rechtepacket zugewiesen, das es ihm möglich macht eigene Channels zu erstellen. Hat er einen Channel erstellt, ist er dort Channel-Admin solange er den Channel nicht verlässt, er hat in diesem Channel alle Rechte. User die den Channel nur betreten, bekommen ein Standard Channel Rechtepacket, sie dürfen den Channel standardmässig betreten und darin sprechen.

Das Security Modul führt aber noch eine weitere Art von Channels ein, sogenannte persistente Channel. Diese sind in der Datenbank hinterlegt und werden beim starten des Servers automatisch erzeugt. Um in einen persistenten Channel betreten zu können benötigt ein User extra Rechte, hier wird kein standard Regelpacket erzeugt, hat der User keine Rechte für diesen Channel in der Datenbank ist es auch nicht möglich den Channel zu betreten.

2.4 User-Schnittstelle (Kommandos)

- **/identify "username" "password"**
  Identifiziert den User beim Security Modul, die globalen Rechte des Users werden aus der DB geladen
- **/giveOp "channel" "username"**
  Ein Channeladministrator, hat die Möglichkeit einem anderen User im Channel Administratorenrechte zu geben
- **/takeOp "channel" "username"**
  Einen Channeladministrator kann Admin Rechte eines anderen Users entfernen

Chapter 3

Administrations-Interface

3.1 Allgemein

Um die Module und deren Einstellungen ,wie zB die Rechte des Security Moduls, zu verwalten wurde ein Webinterface auf Basis von Html, Javascript und Perl-CGI Programmierung entwickelt. Als Datenbankbackend kommt natürlich MySQL zum Einsatz.

3.2 Aufbau des Source Codes

```
pms-admin
├── css                                 (Stylesheet Dateien)
├── img                                 (Vom Interface vewendete Images)
├── js                                  (Javascript Code der Grundkonfig)
│   ├── bootstrap.min.js                (Fremdsoftware)
│   ├── helper.js
│   ├── jquery-blockUI.js               (Fremdsoftware)
│   ├── jquery.js                       (Fremdsoftware)
│   ├── login.js
│   └── userview.js
├── Pms
│   ├── BaseModule.pm (Die Basisklasse für alle Module)
│   ├── MainView.pm (Grundgerüst der Oberfläche)
│   ├── Modules
│   │   └── Security
│   │       ├── js (Die Javascript Dateien des Security Moduls)
│   │       │   ├── channel.js
│   │       │   ├── channelRoles.js
│   │       │   ├── user.js
│   │       │   └── userRoles.js
│   │       ├── tmpl                    (Die HTML Templates des Moduls)
│   │       ├── Mod.pm                  (Die Implementation des Moduls selbst)
│   │       ├── Channel.pm              (Die View um Channels anzulegen und zu löschen)
│   │       ├── ChannelRoles.pm         (Die View um Channel-Rechte zu verwalten)
│   │       ├── User.pm                 (Die View um User anzulegen und zu löschen)
│   │       └── UserRoles.pm            (Die View um User-Rechte zu verwalten)
│   ├── Session.pm              (Alle Session bezogenen Funktionalitäten)
│   └── UserView.pm             (Modul, zur Administratoren-Verwaltung)
├── tmpl                        (Enthält die HTML-Templates des Basis Interfaces)
│   ├── addUser.tmpl
│   ├── base.tmpl
│   ├── index.tmpl
│   └── login.tmpl
│
├── index.pl            (siehe Abschnitt Application Entry Scripts)
├── login.pl            (siehe Abschnitt Application Entry Scripts)
├── module.pl           (siehe Abschnitt Application Entry Scripts)
├── PmsConfig.pm        (Die Konfigurationsdatei des Servers)
└── users.pl            (siehe Abschnitt Application Entry Scripts)

9 directories, 42 files
```

3.3 Fremdsoftware

- JSON (2.53) und JSON::XS (2.27)

  Aus dem CPAN nachinstallierte Module, sie stellen einen Parser und Serializer für JSON Dokumente bereit. JSON ist ein leichtgewichtiges Datentransferformat, das das Webinterface verwendet um Daten per AJAX vom Server abzuholen.

- HTML::Template

  Aus dem CPAN nachinstalliertes Modul, es ermöglicht die strikte Trennung von HTML

und Perl Code. Zu diesem Zweck wird der HTML Code in sogenannte Template Dateien ausgelagert, in denen bestimmte "Tags" hinterlegt sind. Diese "Tags" kann man durch dynamische Daten ersetzen.

- JQuery

  JQuery ist ein Javascript Framework, das es einfach macht Browserübergreifendes Javascript zu schreiben, in unserer Software kommmt es hautpsächlich bei AJAX-Datenabfragen und beim dymamischen ändern des DOM-Trees zum Einsatz.

- Twitter Bootstrap

  Bootstrap ist ein Html, Css und Javascript Framework, das vordefinierte Elemente liefert, um das Styling und Erstellen von HTML basierten Interfaces zu erleichtern. Die komplette Ui des Admin Interfaces wurde mit Bootstrap erstellt.

3.4 Implementation und Design

Auch das Admin Interface ist wie der Server modular aufgebaut. Server Module, die eine Konfiguration benötigen, die über die Fähigkeiten einer Konfigurationsdatei hinausgehen, müssen nur ein Modul/Plugin für das Admin Interface liefern und einige bestimmte API Vorgaben erfüllen.

3.4.1 Application Entry Scripts

Es gibt insgesamt nur 4 Skripte, die der HTTP-Server direkt ausführen kann, diese leiten die Requests dann an interne Module weiter oder bearbeiten sie direkt selbst:

- **login.pl**

  Ist für Login und das erstellen der Session zuständig, gibt
  es schon eine bestehende Session, wird der Client auf das index.pl
  Script umgeleitet.

- **index.pl**

  Zeigt die Startseite und das Menu an.

- **module.pl**

  Dieses Skript behandelt Requests an die Module -> s.a. Module Funktionsweise

- **users.pl**

  Ist dafür zuständig neue Admin User über die Oberfläche anzulegen und zu verwalten.

3.4.2 Funktionsweise von Modulen

Wie schon im vorherigen Abschnitt erwähnt, werden Requests an die Module von Application-Entry Skript module.pl gesteuert. Hierbei wird vorausgesetzt, das es ein Module gibt das folgende Voraussetzungen erfüllt:

- Ein Modul im Verzeichnis Pms/Modules/<PluginName> mit dem Namen Mod.pm

  `Für das Security Modul würde das bedeuten: Pms/Modules/Security/Mod.pm`

- Dieses Module implementiert die Basisklasse Pms::BaseModule
- Es wurden alle abstrakten Funktionen implementiert.

Für genauere Informationen wie ein Modul implementiert wird, beachten Sie bitte die API Dokumentation.

Es gibt zwei Arten von Requests an ein Modul: 1. Die Anforderung einer neuen HTML-Seite, also eine View 2. Ein Datenrequest das per AJAX geschickt wird.

Welche Art von Request durchgeführt wird und an welches Modul es geschickt wird, wird per GET Daten in der URL entschieden:

URL für eine View

```
http://hostname/module.pl?mod=<ModuleName>;view=<ViewName>
```

Wie man im Beispiel erkennen kann, wird über die **mod** Variable die Bezeichnung des Moduls an das module.pl Skript übergeben, dieses versucht dann zur Laufzeit das Modul zu laden und ruft die Funktion **renderContent($viewName)** auf. Diese Funktion sollte den Inhalt der HTML Seite zurückgeben, dieser Inhalt wird dann in den Datenbereich des Haupt-Templates eingebettet.

URL für einen Daten Request

```
http://hostname/module.pl?mod=<ModuleName>;action=<ActionName>
```

Die Url für einen Daten-Request ist im Aufbau der View-Request ähnlich, allerdings hat sie anstatt dem **view** Parameter, einen **action** Parameter. Das Skript module.pl lädt das Plugin in der gleichen Weise wie bei der View, führt hier jedoch die Funktion **dataRequest($actionName)** aus. Daten, die an die dataRequest Funktion selbst übergeben werden, müssen als JSON-Dokument per POST Request übergeben werden. In der aktuellen Konfiguration werden unbekannte GET-Parameter einfach ignoriert.