FVS Dokumentation

Release 1.0.0

Ritchie Flick

07. 03. 2013

Inhaltsverzeichnis

KAPITEL 1

Allgemeines

1.1 Grundlegende Informationen

FVS ist als Teil einer Abschlussarbeit *(Projet de fin d'étude – Formation du Technicien en Informatique)* der Klasse T3IFAN 2012/2013 in Wiltz, Luxemburg entstanden.

- **Author** Ritchie FLICK
- **Klasse** T3IFAN 2012/2013
- **Schule** LN - Lycée du Nord
- **Projektname** *Feuerwehrverwaltungssoftware - FVS*

1.2 Begriffserklärungen

- **FVS**: **Feuerwehrverwaltungssoftware** = Der Name dieses Projektes
- *Modul*: Ein Modul in *Python* ist im allgemeinen eine Datei welche nicht als eigenständiges Programm aufgerufen wird, sondern in ein Programm eingebunden wird. Es kann ausführbare Anweisungen, als auch Klassen- oder Funktiondefinitionen enthalten. [1]
- *Pakete*: Pakete werden dazu verwendet, den Namensraum um *Python Module* zu strukturieren, damit muss der Autor sich keine Sorgen um mögliche gleiche Modulnamen oder globale Variablennamen zu machen. [2]

[1] Weitere Informationen zu *Modulen* zu finden auf: http://tutorial.pocoo.org/modules.html
[2] Weitere Informationen zu *Paketen* zu finden auf: http://tutorial.pocoo.org/modules.html#pakete

1.3 Anforderungen

1.3.1 Betriebssysteme

FVS wurde getestet auf *Windows 7* [3] von *Microsoft* [4] und auf *Kubuntu 12.10* [5] von *Canonical* [6]. **FVS** ist somit kompatibel und lauffähig auf *Windows* als auch auf den meisten *Linux* Distributionen.

Bermerkung

Obwohl **FVS** theoretisch auch auf *Win 7* oder *Ubuntu* Touch Tablets/Laptops lauffähig ist, wurde dies jedoch noch nicht getestet und es kann keine Garantie auf eine problem freie Benutzung auf solche Geräte gegeben werden.

1.3.2 Datenbank

FVS ist nur kompatibel mit einer *MySQL* [7] Datenbank von *Oracle* [8]. Es wurden keine weiteren Datenbanken getestet, weder noch im Programm selbst eingebaut.

1.3.3 Python & PySide

FVS wurde mit *Python 3.2* und *PySide 1.1.1* programmiert. Es kann keine Garantie für die Komptabilität mit anderen Versionen gegeben werden. [9]

1.3.4 Qt & QML

FVS wurde mit *Qt 4.8* und *QML 1.1* programmiert. Es kann keine Garantie für die Komptabilität mit anderen Versionen gegeben werden. [10]

[3] http://windows.microsoft.com/en-us/windows7/products/home
[4] http://www.microsoft.com/
[5] http://www.ubuntu.com/
[6] http://www.canonical.com/
[7] http://www.mysql.com/
[8] http://www.oracle.com/index.html
[9] Einschränkungen sind nur für die unkompilierte Version von **FVS** zu erwarten. Die kompilierte Version von **FVS** hat keine externen Abhängigkeiten.
[10] Einschränkungen sind nur für die unkompilierte Version von **FVS** zu erwarten. Die kompilierte Version von **FVS** hat keine externen Abhängigkeiten.

KAPITEL 2

Einleitung

2.1 Aktuellste Programmversionen

Die aktuellsten Programmversionen für *Windows* und *Linux* sind zu finden auf:

https://bitbucket.org/Xenplex/fvs/downloads

2.2 Technische Dokumentation

Die aktuellste Version dieser Dokumentation ist zu finden auf:

http://158.64.96.130/~reserve/

Oder als Download unter: https://bitbucket.org/Xenplex/fvs/downloads

2.3 Benutzerhandbuch

Die aktuellste Version des Benutzerhandbuches ist zu finden auf:

https://bitbucket.org/Xenplex/fvs/downloads

2.4 Programmcode Repository

Die aktuellste Version des Projektes ist zu finden auf:

http://bitbucket.org/Xenplex/fvs

2.5 Aufgabestellung von FVS

Die Feuerwehrverwaltungssoftware oder kurz **FVS** ist ein Desktopprogramm, geschrieben in *QML* [1] und *Python PySide* [2], zur Verwaltung einer Feuerwehreinsatzstelle. **FVS** ist aus folgenden *Benutzerbereichen* aufgebaut:

- Administrator
- Kommandant
- Sekretär
- Inventarist
- Maschinist

FVS hat sich zum Ziel gesetzt die Aufgaben dieser Bereiche für die jeweiligen Mitglieder zu vereinfachen und somit die Übersicht und Zusammenarbeit zu verbessern.

Die Benutzer von **FVS** erhalten die Möglichkeit automatisch Meldungen zu erhalten, wann zum Beispiel die *TÜV* Kontrolle eines Fahrzeugs abgelaufen ist, Inventargegenstände zu verwalten und Berichte erstellen zu können.

2.6 Kurzinformationen über eine Feuerwehreinsatzstelle

Um die Aufgabestellung von **FVS** besser zu verstehen, soll hier eine kurze Übersicht über den allgemeinen Aufbau einer Feuerwehreinsatzstelle gegeben werden. [3]

Eine Feuerwehreinsatzstelle wird von Mitgliedern in unterschiedlichen Positionen und Aufgaben verwaltet.

Der *Kommandant* als Leiter der Einsatzstelle kümmert sich vor allem um die Verwaltung der einzelnen Mitglieder und soll eine Übersicht über alle Bereiche haben.

Der *Sekretär* kümmert sich um organisatorische Aufgaben, wie zum Beispiel der Aufbau des Übungsplans, der Führung von Protokollen während einer Versammlung und dem Ausfüllen der Einsatzberichte.

Der *Inventarist* kümmert sich um den gesamten Inventar einer Feuerwehreinsatzstelle und muss sich um mögliche beschädigte Inventargegenstände kümmern und den Überblick behalten wo die einzelnen Inventargegenstände verwendet werden, damit bei Bedarf die jeweiligen Gegenstände wieder gefunden werden können.

Der *Maschinist* kümmert sich um die Fahrzeuge einer Feuerwehreinsatzstelle und stellt sicher dass alle Fahrzeuge die benötigte Ausstattung besitzen und die einzelnen Kontrollen (z.Bsp: *TÜV*) durchgeführt wurden.

2.7 Release Skripte

2.7.1 Linux Release Skript

Das **Linux Release Skript** dient zur Automatisierung verschiedener Aufgaben auf *Linux* Systemen. Ein Ausschnitt aus der Dokumentation von **fvs_linux_release.sh** [4]:

> **fvs_linux_release.sh** ist ein *Bash* Skript zur Kompilation von **FVS - Feuerwehrverwaltungssoftware**, der Generierung seiner Dokumentation mit *Sphinx* und weiteren Funktionen.

[1] *Qt Meta Language*: Bestandteil von *Qt* und dient zum Aufbau von grafischen Benutzeroberflächen
[2] *Qt* Anbindungen für *Python*: http://qt-project.org/wiki/PySide
[3] Weitere Informationen zu finden auf: http://de.wikipedia.org/wiki/Feuerwehr
[4] http://en.wikipedia.org/wiki/Bash_(Unix_shell)

Das Skript lässt den Entwickler eine *fvs_release_konfig.cfg* Datei erstellen als Konfigurationsdatei, in dem die aktuelle Versionsnummer von **FVS** angegeben wird, der Name des Entwicklers und gegebenfalls in welches Verzeichnis das Projekt *exportiert* werden soll.

FVS wird mit der angegebenen Versionsnummer und Entwickler Name kompiliert und ins entsprechende **Release** Verzeichnis kopiert. Die Bibliotheken werden in ein verstecktes Verzeichnis kopiert, damit der Benutzer keine für ihn unnötigen Dateien sieht, um die Benutzer- freundlichkeit zu erhöhen.

Die Dokumentation von **FVS** wird mit der angegebenen Versionsnummer und dem Namen des Entwicklers neu generiert und ins aktuelle **Release** Verzeichnis von **FVS** kopiert.

Die benötigten *SQL* Skripte werden in das aktuelle **Release** Verzeichnis kopiert um dem Entwickler/Benutzer die Einrchtung der Datenbank und/oder das Entwickeln/Testen von **FVS** zu erleichtern.

Das **Release** Verzeichnis kann optional *exportiert* werden in ein gewünschte Verzeichnis. (z.Bsp.: *Dropbox*)

Des weiteren kann **FVS** mit *git* und der angegebenen Versionsnummer *getagt* und anschließend automatisch ins eingerichtete *git repository gepusht* werden. (z.Bsp.: *Github*, *Bitbucket*)

2.7.2 Windows Release Skript

Das **Windows Release Skript** dient zur Automatisierung verschiedener Aufgaben auf *Windows* Systemen. Ein Ausschnitt aus der Dokumentation von **fvs_win_release.ps1** [5]:

fvs_win_release.ps1 ist ein *Powershell* Skript zur Kompilation von **FVS - Feuerwehrverwaltungssoftware**, der Generierung seiner Dokumentation mit *Sphinx* und weiteren Funktionen.

Das Skript lässt den Entwickler eine *XML* Datei erstellen als Konfigurationsdatei, in dem die aktuelle Versionsnummer von **FVS** angegeben wird, der Name des Entwicklers und gegebenfalls in welches Verzeichnis das Projekt *exportiert* werden soll.

FVS wird mit der angegebenen Versionsnummer und Entwickler Name kompiliert und ins entsprechende **Release** Verzeichnis kopiert. Die Bibliotheken erhalten das Attribut *hidden*, damit der Endbenutzer nur die für ihn wichtigen Dateien sieht, zur Erhöhung Benutzerfreundlichkeit.

Die Dokumentation von **FVS** wird mit der angegebenen Versionsnummer und dem Namen des Entwicklers neu generiert und ins aktuelle **Release** Verzeichnis von **FVS** kopiert.

Die benötigten *SQL* Skripte werden in das aktuelle **Release** kopiert um dem Entwickler/Benutzer die Einrchtung der Datenbank und/oder das Entwickeln/Testen von **FVS** zu erleichtern.

Das **Release** Verzeichnis kann optional *exportiert* werden in ein gewünschte Verzeichnis. (z.Bsp.: *Dropbox*)

Des weiteren kann **FVS** mit *git* und der angegebenen Versionsnummer *getagt* und anschließend automatisch ins eingerichtete *git repository gepusht* werden. (z.Bsp.: *Github*, *Bitbucket*)

2.8 Erstellen von Berichten

FVS generiert seine Berichte als *HTML & CSS* Seiten. Dies ermöglicht dem Benutzer von **FVS** seine Berichte mit *CSS* Kenntnissen seinen Bedürfnissen anzupassen und wenn gewünscht auf seiner Webseite zu präsentieren.

Die Berichte können dann über die Print Funktion des Browsers auf Wunsch ausgedruckt werden.

Die Berichte werden in das Verzeichnis *Berichte/* abgelegt, in der auch die *fvs.css* Datei liegt, welche ein erfahrener Benutzer verwenden kann um die Berichte anzupassen.

[5] http://en.wikipedia.org/wiki/Windows_PowerShell

Die Berichte werden im folgendem Format abgespeichert: **bericht<benutzer_name><datum_erstellung>**

KAPITEL 3

Technik

3.1 Verwendete Technologien

FVS verwendet folgende Technologien:

- Python [1]
- Qt & QML [2]
- PySide [3]
- MySQL [4]

3.1.1 Python

Python ist eine höhere Programmier- bzw. Skriptsprache und *Open Source*. Es ist eine multiparadimatische und dynamische Programmiersprache und ist 1991 erstmals erschienen. Obwohl *Python* mehrere Programmierparadigma unterstützt, wird in **FVS** vorrangig die objektorientierte Programmierparadigma verwendet.

Python wurde als Programmiersprache gewählt da sie einfach zu lernen, eine klare Syntax und viele Standard Bibliotheken von Haus liefert. Desweiteren wird *Python* ständig weiter entwickelt und verbessert und besitzt eine aktive Community. *Python* ist außerdem platformunabhängig, was die Verwendung von **FVS** auf den meisten Betriebssystemen möglich macht.

3.1.2 Qt & QML

Qt (aussgesprochen eng. **cute**) ist eine plattformunabhängige Bibliothek/Framework für grafische Benutzeroberflächen. *Qt* wird meistens in Verbindung mit der Programmiersprache *C++* verwendet, jedoch kann es auch mit anderen Programmiersprachen, über sogenannte Sprachanbindungen [5] [6], verwendet werden.

[1] http://python.org/
[2] http://qt-project.org/
[3] http://qt-project.org/wiki/Category:LanguageBindings::PySide
[4] http://www.mysql.com/
[5] http://de.wikipedia.org/wiki/Sprachanbindung
[6] http://en.wikipedia.org/wiki/Wrapper_library

Wenn das Produkt welches mit *Qt* entwickelt wird, nicht unter einen freien Lizenz steht, kann wahlweise eine kommerzielle Lizenz erworben werden. Wird jedoch eine freie Lizenz für das Projekt verwendet, kann *Qt* außerdem unter einen freien Lizenz verwendet werden.

QML ist eine Entwicklung von *Qt* zur Entwicklung von grafischen Benutzeroberflächen, welche vor allem *Touch-Screen* freundlich sind. Weiterhin ermöglich *QML* sehr viele Möglichkeiten zum Aufbau der grafischen Benutzeroberfläche. Die GUI lässt sich beinahe wie eine HTML&CSS Seite frei gestalten und kann somit genau so viele visuelle Effekte enthalten.

Qt und *QML* wurden ausgewählt, da es stark weiterentwickelt wird, es mit einer freien Lizenz verwendet werden kann und für spätere Versionen offizieller Support für *Android*, *iPhone* geplant ist (es gibt auch inoffizielle Ports [7]), was die Benutzer **FVS** noch effektiver verwenden lässt. *QML* wird außerdem bei *Ubuntu Touch* [8] von Haus aus unterstützt.

3.1.3 PySide

PySide ist eine Sprachanbindung für *Python* und *Qt*. *PySide* wurde im Gegenstatz zu *PyQt* [9] verwendet, da *PySide* das verwenden einer freien Lizenz erlaubt.

Der Grund wieso *PySide* überhaupt gewählt wurde, ist in den ersten beiden Punkten für *Python* und *Qt* erklärt.

3.1.4 MySQL

FVS benötigt für seine Arbeit eine Datenbank und *MySQL* ist eine der am weitesten verbreiteten *Open Source* Datenbanken.

3.2 Verwendete Bibliotheken

Im folgenden sollen Bibliotheken, welche in **FVS** verwendet wurden, jedoch nicht Teil der Standardbibliothek einer Sprache sind, aufgeslitet werden:

- PyMySQL: Ein purer *Python MySQL* Klient. Wird in **FVS** verwendet für die Verbindung mit der *MySQL* Datenbank. [10]

3.3 Verwendete Code Ausschnitte

Liste mit allen Code Ausschnitten welche über *copy/paste* in **FVS** eingebunden wurden:

- Javascript Funktion zum Kontrollieren eines Datums: http://www.qodo.co.uk/blog/javascript-checking-if-a-date-is-valid/
- Funktion für die **Memo QML_Komponente**, kopiert von http://doc-snapshot.qt-project.org/4.8/qml-textedit.html

3.4 Verwendete Software/Programme

Während der Arbeit an **FVS** wurden verschiedene Programme verwendet:

[7] http://thp.io/2011/pyside-android/
[8] http://developer.ubuntu.com/get-started/gomobile/
[9] http://www.riverbankcomputing.com/software/pyqt/intro
[10] https://github.com/petehunt/PyMySQL

- *Sublime Text 2* ist ein moderner, einfacher Text Editor für Programmierer. [11] Wurde zur Erstellung der *Python* Dateien verwendet.
- *Qt Creator* ist eine *IDE*, speziell für das *Qt Framework*. [12] Wurde zur Erstellung der *QML* Dateien verwendet.
- Zur Verwaltung des Programmcodes von **FVS** wurde *git* [13] als *Versionsverwaltung* [14] und *bitbucket* [15] als Host des Programmcodes verwendet.
- *MySQL Workbench* [16] zur Erstellung/Bearbeitung/Verwaltung der *MySQL* Datenbank
- *Umbrello UML Modeller* [17] ist ein *KDE* [18] zur Erstellung von *UML* Diagrammen für den Aufbau der Datenbank und dem allgemeinen Aufbau von **FVS**.
- *ReText* [19] zum Schreiben der *ReStructuredText* [20] Datein für die Dokumentation von **FVS**.
- *cxFreeze* [21] ist eine Zusammenstellung von Skripten zum *Einfrieren* von Python Anwendungen in ausführbare Dateien.
- *Shutter 0.89* [22] zum Erstellen und Bearbeiten der Screenshots für das Benutzerhandbuch

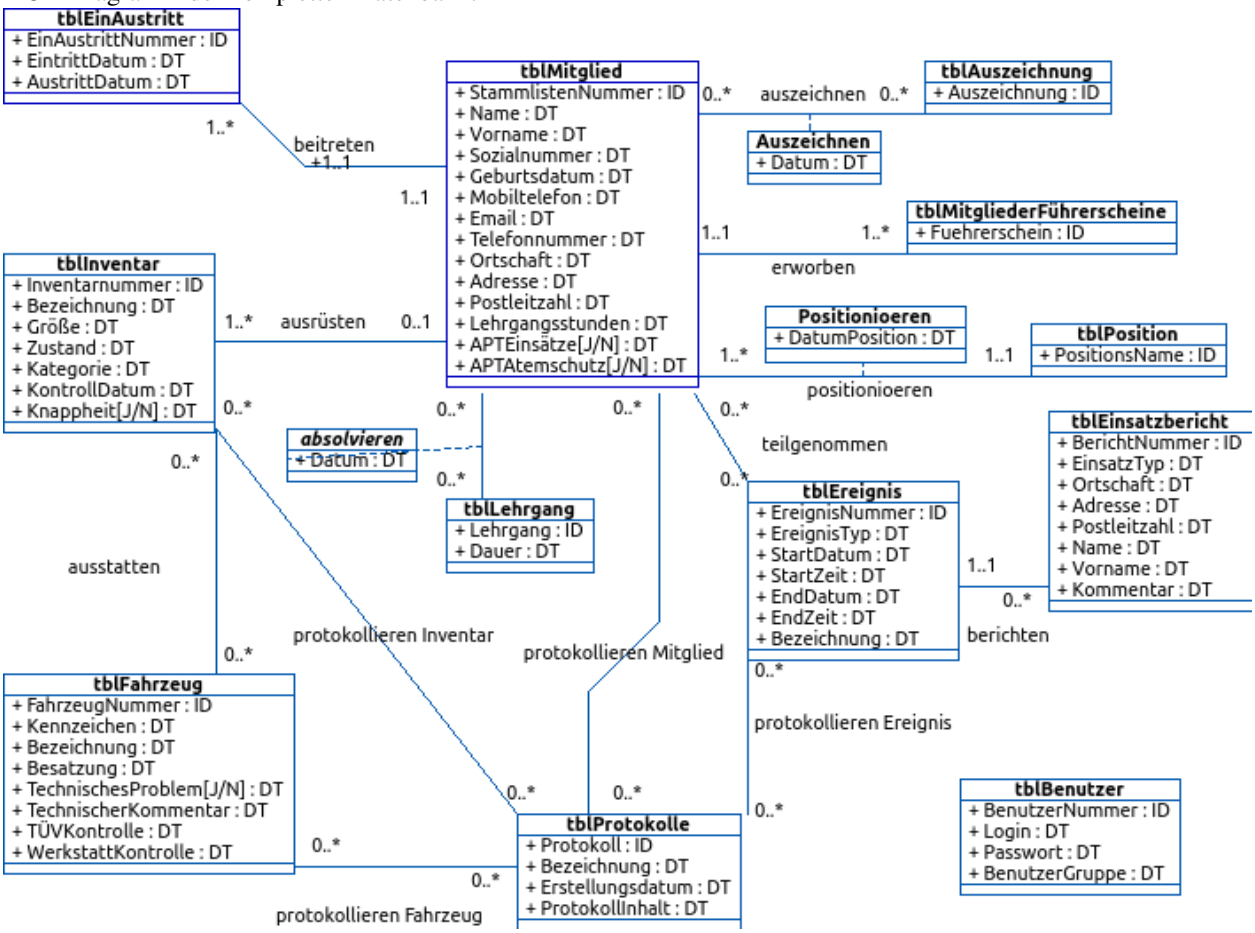
[11] http://www.sublimetext.com/
[12] http://qt-project.org/wiki/Category:Tools::QtCreator
[13] http://git-scm.com/
[14] http://de.wikipedia.org/wiki/Versionsverwaltung
[15] https://bitbucket.org/
[16] http://www.mysql.com/products/workbench/
[17] http://uml.sourceforge.net/
[18] http://www.kde.org/
[19] http://sourceforge.net/p/retext/home/ReText/
[20] http://docutils.sourceforge.net/rst.html
[21] http://cx-freeze.sourceforge.net/
[22] http://shutter-project.org/

Projekt Aufbau

4.1 Datenbank Aufbau

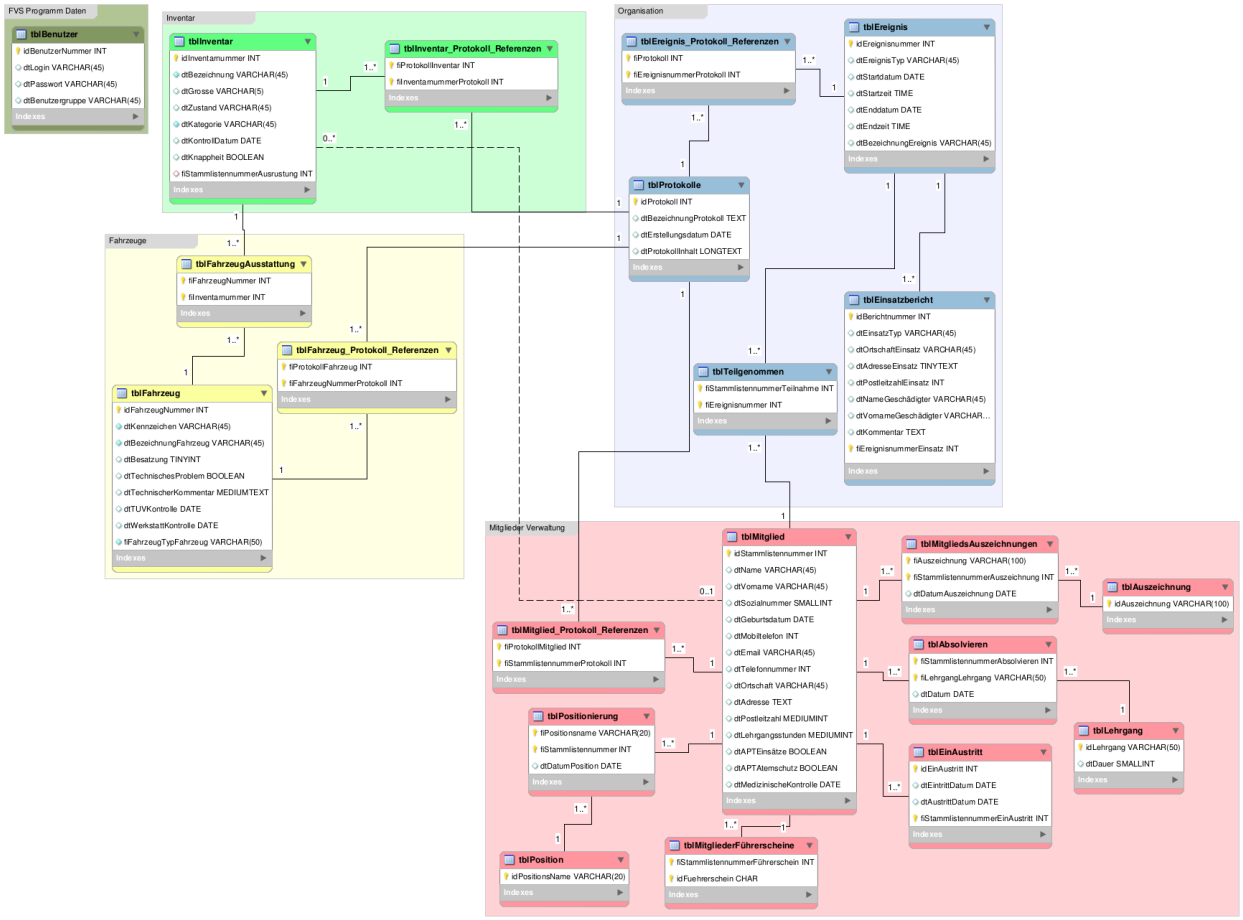
4.1.1 MCD

MCD Diagramm der kompletten Datenbank:



4.1.2 MySQL Workbench EER Modell

Die komplette Datenbank als *EER Modell* [1] generiert mit *MySQL Workbench*.



4.1.3 Erklärungen zu der einzelnen Tabellen

tblMitglied

tblMitglied enthält alle Informationen über ein Mitglied.

tblEinAustritt

tblEinAustritt enthält alle Informationen über einen Ein- bzw Austritt aus der Feuerwehr eines Mitglieds. Jedes Mitglied kann mehrmals aus der Feuerwehr Ein- bzw Austreten.

tblAuszeichnung

tblAuszeichnung enthält alle möglichen Auszeichnungen welche ein Mitglied während seinen Dienstjahren in der Feuerwehr erhalten kann.

[1] http://en.wikipedia.org/wiki/Enhanced_entity%E2%80%93relationship_model

tblMitgliederFührerscheine

tblMitgliederFührerscheine enthält alle möglichen Führerscheine welche ein Mitglied besitzen kann.

tblPosition

tblPosition enthält alle Positionen welche von den Mitglieder eingenommen werden können.

tblEreignis

tblEreignis enthält alle Ereignisse welche für eine Feuerwehr auftreten können. Zu Eregnissen zählen:

- Übungen
- Medizinische Kontrollen
- Fest
- Einsätze

tblLehrgang

tblLehrgang enthält alle möglichen Lehrgänge welche ein Mitglied absolvieren kann und wie lange ein Lehrgang dauert.

tblEinsatzbericht

tblEinsatzbericht wird als Erweiterung der Tabelle **tblEreignis** verwendet und enthält alle weiteren Informationen welche für einen Einsatz aufgezeichnet werden müssen.

tblInventar

tblInventar enthält alle Inventargegenstände einer Feuerwehr. Dazu zählen:

- Kleider
- Kommunikationsgeräte
- Erste Hilfe Ausrüstung
- Wasserführende Armaturen
- Schläuche
- Atemschutzgeräte
- Maschinen (Lüfter, Motorsäge, ...)

tblFahrzeug

tblFahrzeug enthält alle Informationen welche über ein Fahrzeug benötigt werden.

tblProtokolle

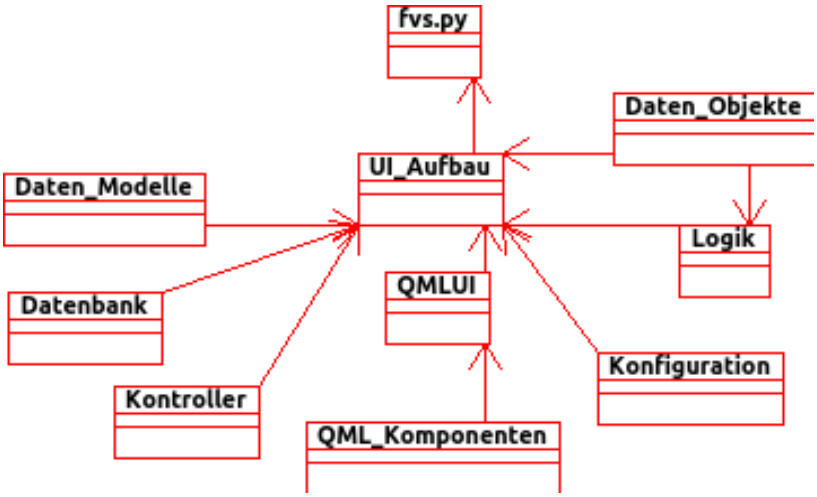
Enthält alle möglichen Protokolle welche erstellt worden sind. Ein Protokoll hat Verbindungen zu den Tabellen:

- **tblMitglied**
- **tblEreignis**
- **tblFahrzeug**
- **tblInventar**

Da innerhalb eines Protokoll *Referenzen* auf einzelne Mitglieder, Ereignisse, Fahrzeuge, Inventargegenstände gesetzt werden können.

4.2 Programm Aufbau

4.2.1 Struktur



Die **fvs.py** ist die Hauptdatei und von ihr aus wird **FVS** gestartet. Im Verzeichnis **UI_Aufbau** sind die Hauptdateien der einzelnen Bereiche. Sie importieren die für sie benötigten *Python Module* und sie bauen die grafische Benutzeroberfläche aus **QMLUI** auf.

QML_Komponenten enthält einzelne Komponenten der **QMLUI** (Buttons, Listen, Labels, ...), aus welchen die Oberfläche aufgebaut wird.

Weitere Informationen zu den einzelnen Dateien/Verzeichnisse ist zu finden im Teil der *Programmcode Dokumentation*

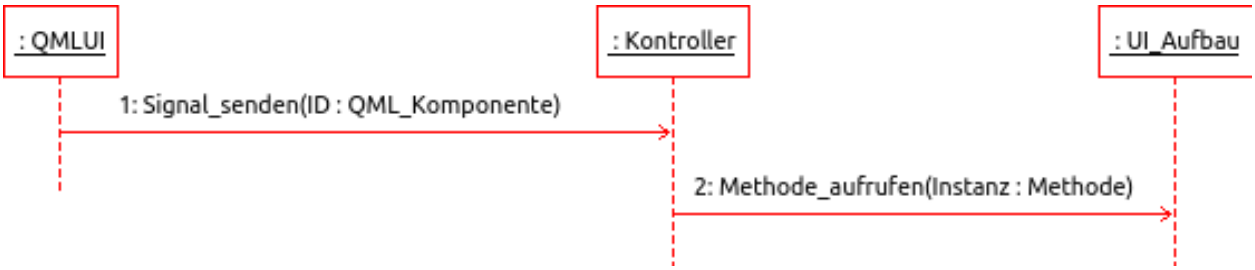
4.2.2 Kontroller

Die Kontroller kümmern sich um das *empfangen* der Signale aus der **QMLUI** und dem Aufrufen der jeweiligen Methode, welche zur ID des Senders passen.

Legende

- **QMLUI**: *XY* QML Komponent aus der grafischen Oberfläche
- **Kontroller**: *XY* Kontroller zuständig für *XY* QML Komponent
- **UI_Aufbau**: *XY* Modul welches die **QMLUI** aufgebaut hat

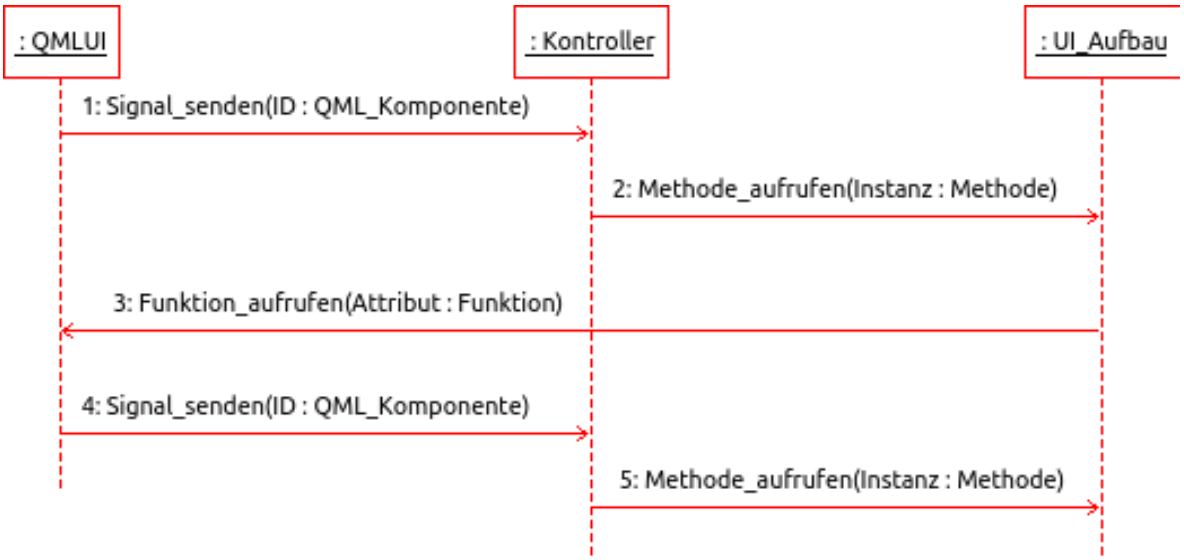
Einfacher Kontroller



Operationen Kontroller

Der Operationen Kontroller wird benötigt, da die **QMLUI** ein *QtCore.Signal()* an den Kontroller schickt, wenn jedoch zum Beispiel die Daten aus der grafischen Benutzeroberfläche ausgelesen werden soll (Beispiel aus Eingabefeldern), so muss diese Operation erst durch den *Python* Unterbau gestartet werden.

Nach dem Ablauf *2: Methode_aufrufen* geht ein Signal zurück an die **QMLUI**, welches diese veranlasst die Informationen in Variablen zu schreiben. Bei diesem Vorgang tritt folgendes Problem auf: Wenn der *Python* Unterbau nicht auf ein Signal der **QMLUI** warten würde, würde es die Variablen auslesen, bevor die **QMLUI** diese neue gesetzt hat.



Bemerkung

Die Lösung der Kontroller ist suboptimal, jedoch konnte bislang keine bessere Lösung gefunden werden, welche mit dem momentanen Aufbau & Struktur von **FVS** kompatibel ist.

4.3 Benutzerbereiche

FVS ist in 5 Benutzerbereiche aufgeteilt:

- Admin
- Inventarist
- Kommandant

- Maschinist
- Sekretär

4.3.1 Admin

Der *Administrator* hat keinen direkten Zugang zu den anderen Benutzerbereichen, jedoch hat er die Möglichkeit die Tabellen der Datenbank zu kontrollieren/optimieren/reparieren. [2]

4.3.2 Inventarist

Der *Inventarist* kümmert sich um die Verwaltung des Inventars der Feuerwehr. Der Inventarist kontrolliert den Bestand der Gegenstände und deren Zustand.

4.3.3 Kommandant

Der *Kommandant* kümmert sich um die Verwaltung der Mitglieder. Außerdem hat der *Kommandant* als Leiter der Feuerwehr zugang zu allen anderen Benutzerbereichen mit Ausnahme des *Admin* Bereichs.

4.3.4 Maschinist

Der *Maschinist* kümmert sich um die Fahrzeuge der Feuerwehr und kontrolliert regelmäßig deren Ausstattung und die Datumen derer Kontrollen.

4.3.5 Sekretär

Der *Sekretär* kümmert sich um die Verwaltung der Übungen, Einsatzberichten und die Protokollführung, in welchem Referenzen über Fahrzeuge, Mitglieder, Ereignisse oder Inventargegenstände gesetzt werden können, um so besser die Übersicht zu behalten, von was in einem Protokoll die Rede ist.

4.4 Login Informationen

Benutzername	Passwort	Benutzerbereich
admin	admin	Administrator
inventarist	inventarist	Inventarist
kommandant	kommandant	Inventarist, Kommandant Maschinist, Sekretär
maschinist	maschinist	Maschinist
sekretaer	sekretaer	Sekretär

[2] Die verwendete Version der Datenbank und Storage Engines sind nicht kompatibel mit diesen Funktionen

KAPITEL 5

Sphinx/

Das Verzeichnis *Sphinx/* enthält alle Dateien welche Sphinx [1] benötigt um die Dokumentation für **FVS** generieren zu können.

5.1 Makefile

Die *Makefile* Datei dient zum erstellen der **FVS** Dokumentation auf *Linux* Systemen:

```
SPHINXOPTS    =
SPHINXBUILD   = sphinx-build -D copyright="$(name)" -D release="$(version)"
PAPER         =
BUILDDIR      = ../Dokumentation

PAPEROPT_a4     = -D latex_paper_size=a4
PAPEROPT_letter = -D latex_paper_size=letter
ALLSPHINXOPTS   = -d $(BUILDDIR)/doctrees $(PAPEROPT_$(PAPER)) $(SPHINXOPTS) source
I18NSPHINXOPTS  = $(PAPEROPT_$(PAPER)) $(SPHINXOPTS) source

.PHONY: html latex latexpdf

html:
    $(SPHINXBUILD) -b html $(ALLSPHINXOPTS) $(BUILDDIR)/html
    @echo
    @echo "Erstellung abgeschlossen. HTML Seiten zu finden in $(BUILDDIR)/html"

latex:
    $(SPHINXBUILD) -b latex $(ALLSPHINXOPTS) $(BUILDDIR)/latex
    @echo
    @echo "Erstellung abgeschlossen; LaTeX Dateien in $(BUILDDIR)/latex."

latexpdf:
    $(SPHINXBUILD) -b latex $(ALLSPHINXOPTS) $(BUILDDIR)/latex
    @echo "LaTeX Dateien werden gesendet an pdflatex..."
    $(MAKE) -C $(BUILDDIR)/latex all-pdf
    @echo "pdflatex abgeschlossen; PDF Datei in $(BUILDDIR)/latex."
```

[1] Sphinx ist ein Software-Dokumentationswerkzeug welche aus reStructuredText Dateien Dokumentationen als HTML Webseiten, PDF Dokumente und weitere mögliche Formate umwandelt. Weitere Informationen zu finden auf: http://sphinx-doc.org/

5.2 make.bat

Die *Batch* Datei zum erstellen der **FVS** Dokumentation auf *Windows* Systemen:

```
if "%SPHINXBUILD%" == "" (
    set SPHINXBUILD=sphinx-build -D copyright=%2 -D release=%3
)
set BUILDDIR=..\Dokumentation
set ALLSPHINXOPTS=-d %BUILDDIR%/doctrees %SPHINXOPTS% source
set I18NSPHINXOPTS=%SPHINXOPTS% source

if "%1" == "html" (
    %SPHINXBUILD% -b html %ALLSPHINXOPTS% %BUILDDIR%/html
    if errorlevel 1 exit /b 1
    echo.
    echo.Erstellung abgeschlossen. HTML Seiten zu finden in %BUILDDIR%/html
    goto end
)

if "%1" == "text" (
    %SPHINXBUILD% -b text %ALLSPHINXOPTS% %BUILDDIR%/text
    if errorlevel 1 exit /b 1
    echo.
    echo.Erstellung abgeschlossen. Text Dateien zu finden in %BUILDDIR%/text
    goto end
)

:end
```

5.3 Sphinx/source/

Enthält alle Dateien welche den Aufbau der Dokumentation festlegen und weitere Dokumentation zu verschiedenen Aspekten von **FVS**.

Programmcode Dokumentation

6.1 src/

Im *Source* Verzeichnis (*src/*) von **FVS** stehen die Hauptdateien welche zur Ausführung von **FVS** benötigt werden und Verzeichnisse (für *Python Packete*) zu den einzelnen Modulen für **FVS**.

6.1.1 fvs.py

fvs.py ist die Ausführungsdatei von **FVS** und dient zum Starten des Programmes. Das Loginfenster wird geladen und aufgerufen.

6.1.2 setup.py

setup.py ist eine Konfigurationsdatei für cx-freeze durch welche die benötigten Einstellungen und Meta-Daten zur Kompilation von **FVS** konfiguriert werden.

6.1.3 Weitere Informationen

Die einzelnen Klassen/Methoden zum Aufbau der **QMLUI** befinden sich in *src/UI_Aufbau*, die Dateien zur Zusammenstellung der grafischen Oberfläche befinden sich in *src/QMLUI* und die einzelnen Komponenten der **QMLUI** befinden sich in *src/QML_Komponenten*.

6.2 Daten_Modelle

Daten_Modelle enthält Dateien zur Modelisierung von Daten Objekten, nach dem *MVC* Prinzip.

6.2.1 list_modell

class Daten_Modelle.list_modell.**ListenModell**(*datensatz*)
Modellklasse für einen Datensatz mit Listenstruktur

Beschreibung

ListenModell ist die Modellklasse für Datensätze, welche als Liste dargestellt werden sollen.

Parameter

•datensatz: Der Datensatz welcher ins **ListenModell** modelisiert werden soll.

data(*index*, *role=PySide.QtCore.Qt.ItemDataRole.DisplayRole*)
Muss in jeder Modellklasse für die View Klasse implementiert sein

rowCount(*parent=<PySide.QtCore.QModelIndex(-1, -1, 0x0, QObject(0x0)) at 0x4464710>*)
*Muss in jeder Modellklasse für die *View* Klasse implementiert sein*

6.2.2 verpacken_modelisieren

Beschreibung

Enthält *Funktionen* welche zum "verpacken" von Datensätzen in ein **QObject** benötigt werden und das anschließende einbinden in die entsprechende *Modell* Klasse.

Daten_Modelle.verpacken_modelisieren.**list_verpackung_modelisieren**(*datensatz*)
Verpackung & Modelisierung eines Datensatzes

Beschreibung

Das übergebene Daten Objekt wird als erstes durch die Funktion **verpacken** (*datensatz*) in ein **QObject** verpackt und anschließend ins Daten Modell für eine **ListenAnsicht** (*ListView*) modelisiert.

Parameter

•datensatz: Daten Objekt welches verpackt und modelisiert werden soll

Rückgabewerte

•datenliste: Verpacktes Daten Objekt in ein **ListenAnsicht** Modell modelisiert.

Daten_Modelle.verpacken_modelisieren.**verpacken**(*datensatz*)
Der übergebene Parameter wird verpackt

Beschreibung

Für die Daten Modelle müssen die Daten Objekte in ein **QObject** verpackt werden damit *Qt* und die **QMLUI** damit arbeiten können.

Parameter

•datensatz: Daten Objekt welches in ein **QObject** verpackt werden soll

Rückgabewerte

•verpackte_daten: Datensatz verpackt als ein **QObject**

6.2.3 verpackung

class Daten_Modelle.verpackung.**DatenVerpackung**(*datensatz*)
Klasse zum Verpacken eines Python Objektes in ein **QObject**

Beschreibung

Klasse welche als *Verpackung* für ein **QObject** verwendet wird. Der Aufbau der Klasse ist vorgegeben und wird von *Qt* so benötigt.

Parameter

•datensatz: Datensatz welcher *verpackt* werden soll

6.3 Daten Objekte

Daten Objekte enthält Python Module, welche zur Erstellung von Daten Objekten verwendet werden.

6.3.1 Fahrzeug Objekt

class Daten_Objekte.fahrzeug_objekt.**Fahrzeug**(*schlussel*, *kennzeichen*, *bezeichnung*, *besatzung*, *technisches_problem*, *kommentar*, *tuv*, *werkstatt*, *typ*)

Erstellung eines Fahrzeug Objektes

Parameter

- schlussel: Die ID des Fahrzeuges
- kennzeichen: Das Kennzeichen des Fahrzeuges
- bezeichnung: Die Bezeichnung des Fahrzeuges
- besatzung: Die Besatzung des Fahrzeuges
- technisches_problem: Boolean Wert ob ein technisches Problem vorliegt oder nicht
- kommentar: Kommentar zu einem technischem Problem
- tuv: Das Datum der letzten TÜV Kontrolle
- werkstatt: Das Datum der letzten Werkstatt Kontrolle
- typ: Der Typ des Fahrzeuges

class Daten_Objekte.fahrzeug_objekt.**FahrzeugTypListe**(*typ*)

Erstellung eines Fahrzeug Typ Liste Objektes

Parameter

- typ: Fahrzeug Typ

6.3.2 Inventarliste Objekt

class Daten_Objekte.inventarliste_objekt.**Groesse**(*groesse*)

Erstellung eines Größe Objektes

Parameter

- groesse: Eine Größen Angabe

class Daten_Objekte.inventarliste_objekt.**Inventarliste**(*schlussel*, *bezeichnung*, *kategorie*, *knappheit=None*, *anzahl=None*, *groesse=None*, *zustand=None*, *kontrolle=None*, *mitglied=None*, *kontrolle_anzeigen=False*)

Erstellung eines Inventarliste Objektes

Parameter

- schlussel: Die ID des Inventargegenstandes
- bezeichnung: Die Bezeichnung des Inventargegenstandes
- kategorie: Die Kategorie des Inventargegenstandes

- knappheit: Ist vom jeweiligen Inventargegenstand noch genügend Vorhanden? (Bsp.: in einem Benzinkanister)
- anzahl: Wie oft kommt ein bestimmter Gegenstand vorkommt
- groesse: Die Größe des Gegenstandes
- zustand: Der aktuelle Zustand des Inventargegenstandes
- kontrolle: Das letzte Kontrolldatum des Inventargegenstandes
- mitglied: Welchem Mitglied ist der Inventargegenstand zugewiesen
- kontrolle_anzeigen: Soll das Datum der letzten Kontrolle in der String Representation angezeigt werden?

class Daten_Objekte.inventarliste_objekt.**Kategorie**(*kategorie*)
Erstellung eines Kategorie Objektes

Parameter

- kategorie: Name einer Kategorie

class Daten_Objekte.inventarliste_objekt.**Kleidung**(*art*)
Erstellung eines Kleidung Objektes

Parameter

- art: Art der Kleidung (Jacke, Uniform,...)

class Daten_Objekte.inventarliste_objekt.**Zustand**(*zustand*)
Erstellung eines Zustand Objektes

Parameter

- zustand: Eine Zustand Angabe

6.3.3 Meldungen Objekt

class Daten_Objekte.meldungen_objekt.**Meldungen**(*meldung*)
Erstellung eines Meldungen Objektes

Parameter

- meldung: Die Meldung welche gesetzt werden soll

6.3.4 Schlauch Objekt

class Daten_Objekte.schlauche_objekt.**SchlauchTypListe**(*typ*)
Erstellung einer SchlauchTypListe Objektes

Parameter

- typ: Ein Schlauch Typ

6.3.5 Mitglied Objekt

class Daten_Objekte.mitglied_objekt.**Mitglied**(*schlussel*, *name*, *vorname*, *sozialnummer=None*, *bday=None*, *mobiltelefon=None*, *email=None*, *telefonnummer=None*, *ortschaft=None*, *adresse=None*, *postleitzahl=None*, *lehrgangsstunden=None*, *mediKontrolle=None*, *apteinsatz=None*, *aptatemschutz=None*)

Erstellung eines Mitglied Objektes

Parameter

- schlussel: Stammlistennummer des Mitglieds
- name: Der Familienname des Mitglieds
- vorname: Der Vorname des Mitglieds
- sozialnummer: Die Sozialversicherungsnummer des Mitglieds
- bday: Der Geburtsdag des Mitglieds
- mobiltelefon: Die Handy Nummer des Mitglieds
- ermail: Die Email Adresse des Mitglieds
- telefonnummer: Die Festnetz des Mitglieds
- ortschaft: Die Ortschaft in der das Mitglied lebt
- adresse: Die Adresse an der das Mitglied lebt
- postleitzahl: Die Postleitzahl des Mitglieds
- lehrgangsstunden: Die restlichen Lehrgangsstunden welche das Mitglied noch für weitere Lehrgänge zur Verfügung hat
- mediKontrolle: Das Datum der letzten medizinischen Kontrolle
- apteinsatz: Ja/Nein Angabe ob das Mitglied die medizinische Erlaubnis hat an Einsätzen teilzunehmen oder nicht
- aptatemschutz: Ja/Nein Angabe ob das Mitglied die medizinische Erlaubnis hat den schweren Atemschutz zu tragen oder nicht

6.3.6 Ereignis Objekt

class Daten_Objekte.ereignis_objekt.**Ereignis**(*schlussel*, *typ=None*, *startdatum=None*, *startzeit=None*, *enddatum=None*, *endzeit=None*, *bezeichnung=None*)

Erstellung eines Ereignis Objektes

Parameter

- schlussel: Die ID des Ereignisses
- typ: Der Typ des Ereignisses
- startdatum: Datum an dem das Ereignis anfing
- startzeit: Uhrzeit an dem das Ereignis anfing
- enddatum: Datum an dem das Ereignis aufhörte

•endzeit: Uhrzeut an dem das Ereignis aufhörte

•bezeichnung: Die Bezeichnung für das Ereignis

class Daten_Objekte.ereignis_objekt.**EreignisTyp**(*typ*)
Erstellung eines EreignisTyp Objektes

Parameter

•typ: Der Typ des Ereignisses

6.3.7 Protokoll Objekt

class Daten_Objekte.protokoll_objekt.**Protokoll**(*schlussel*, *bezeichnung*, *datum*, *inhalt*)
Erstellung eines Protokoll Objektes

Parameter

•schlussel: Die ID des Protokolls

•bezeichnung: Die Bezeichnung des Protokolls

•datum: Das Erstellungsdatum des Protokolls

•inhalt: Der Inhalt des Protokolls

6.3.8 Referenz Objekt

class Daten_Objekte.referenz_objekt.**Referenz**(*schlussel*, *bezeichnung*, *referenzTabelle*)
Erstellung eines Referenz Objektes

Parameter

•schlussel: Die ID der zu referenzierenden Eingabe

•bezeichnung: Die Bezeichnung unter der die zu referenzierende Eingabe zu erkennen ist

•referenzTabelle: Gibt an aus welcher Tabelle die Referenz stammt

6.3.9 Lehrgang Objekt

class Daten_Objekte.lehrgang_objekt.**Lehrgang**(*schlussel*)
Erstellung eines Lehrgang Objekt

Parameter

•schlussel: Die ID/Name des Lehrgangs

6.3.10 Auszeichnung Objekt

class Daten_Objekte.auszeichnung_objekt.**Auszeichnung**(*auszeichnung*)
Erstellung eines Auszeichnung Objektes

Parameter

•auszeichnung: Die ID/Name der Auszeichnung

6.3.11 Führerschein Objekt

class Daten_Objekte.fuhrerschein_objekt.**Fuhrerschein**(*schlussel*)
Erstellung eines Führerschein Objekt

Parameter

•schlussel: Die ID/Name des Lehrgangs

6.3.12 DB Tabellen Objekt

class Daten_Objekte.db_tabellen_objekt.**DBTabelle**(*tabelle*)
Erstellung eines DBTabelle Objektes

Parameter

•tabelle: Der Name der Tabelle

6.4 Datenbank

Alle benötigten Dateien für die *MySQL* Datenbank befinden sich in diesem Verzeichnis.

- BeispielDaten.sql: *SQL* Skript, welches bei Bedarf (zu Entwicklungszwecken oder zu Testzwecken) die Datenbank mit Beispieldaten füllt.
- FVSDatenbank.sql: *SQL* Skript welches die ganze Datenbank mit allen Tabellen und wichtigen Standarddaten.

6.4.1 Datenbank Verbindung

class Datenbank.db_verbindung.**DB_Verbindung**(*db_parameter*)
Datenbankverbindung

Beschreibung

Die Klasse **DB_Verbindung** baut eine Verbindung zur Datenbank auf, dessen Verbindungs Parameter bei der Initialisation übergeben wurden. Instanzvariablen für die *Verbindung* und für den *Cursor* werden generiert.

Alle weiteren Grundfunktionen, welche für die Arbeit mit einer Datenbank benötigt werden, sind als Methoden implementiert.

Parameter

•db_parameter: Enthält Liste mit den Verbindungsparametern welche benötigt werden um eine Verbindung zur Datenbank aufzubauen:

–db_parameter[0]: IP Adresse des Servers

–db_parameter[1]: Port Nummer auf die der Server hört

–db_parameter[2]: Der Name der zu verwendeten Datenbank

–db_parameter[3]: Der Benutzername der Datenbank

–db_parameter[4]: Das Passwort des Benutzers

commit()
Ausstehende Transaktionen werden in die DB geschrieben

execute(*sql_anweisung*, *commit=False*)
Ausführung einer SQL Anweisung

Beschreibung

Die übergebene SQL Anweisung wird ausgeführt und falls der nötige Parameter gesetzt wurde, sofort *committed*.

Parameter

- sql_anweisung: Die auszuführende SQL Anweisung als String
- commit: Soll die Anweisung sofort übernommen werden?

Rückgabewerte

- resultate: Enthält gegebenfalls die Resultate welche die SQL Anweisung zurückgibt

letzter_erstellter_schlussel()
Gibt den letzten erstellten Schlüssel der DB zurück

Beschreibung

Methode welche es erlaubt den letzen erstellen Schlüssel durch *AUTO INCREMENT* aus der Datenbank zu erhalten.

Rückgabewerte

-schlussel: Letzter erstellter *AUTO INCREMENT* Schlüssel der Datenbank

rollback()
Ausstehende Transaktionen werden verworfen

6.4.2 Admin Datenbank Verbindung

class Datenbank.admin_db_verbindung.**Admin_DB_Verbindung**(*db_parameter*)
Basisklassen: Datenbank.db_verbindung.DB_Verbindung

Datenbankverbindung

Beschreibung

Die Klasse **Admin_DB_Verbindung** erbt von der Klasse **DB_Verbindung**.

Weitere Methoden speziell für den *Administrator* werden hier implementiert.

Parameter

- db_parameter: Enthält Liste mit den Verbindungsparametern welche benötigt werden um eine Verbindung zur Datenbank aufzubauen
  - db_parameter[0]: IP Adresse des Servers
  - db_parameter[1]: Port Nummer auf die der Server hört
  - db_parameter[2]: Der Name der zu verwendeten Datenbank
  - db_parameter[3]: Der Benutzername der Datenbank
  - db_parameter[4]: Das Passwort des Benutzers

liste_aller_tabellen()
Eine Liste aller Tabellen in der Datenbank erzeugen

Beschreibung

Es wird eine Liste mit allen Tabellen welche in der Datenbank existieren erzeugt mit dem eingebautenem SQL Befehl *SHOW TABLES*.

Rückgabewerte

•tabellen_liste: Eine Liste aller Tabellen in der Datenbank

tabelle_kontrolle(*tabelle*)
Fehler in der angegebenen Tabelle werden zurückgegeben

Beschreibung

Die angegebene Tabelle wird durch den eingebauten SQL Befehl *CHECKT TABLE* auf Fehler untersucht.

Parameter

•tabelle: Die Tabelle welche mit dem SQL Statement *CHECK TABLE* kontrolliert weden soll.

Rückgabewerte

•kontroll_resultat: Das Resultat der SQL Anfrage

tabelle_optimieren(*tabelle*)
Die angegebene Tabelle wird optimiert

Beschreibung

Die angegebene Tabelle wird mit einem eingebautenem SQL Befehl *OPTIMIZE TABLE* optimiert.

Parameter

•tabelle: Die Tabelle welche optimiert werden soll.

tabelle_reparieren(*tabelle*)
Die angegebene Tabelle wird repariert

Beschreibung

Die angegebene Tabelle wird mit dem eingebautenem SQL Befehl *OPTIMIZE TABLE* repariert.

Parameter

•tabelle: Die Tabelle welche mit dem SQL Statement repariert werden soll.

6.4.3 SQL Anweisungen

Beschreibung

sql_anweisungen.py enthält Funktionen mit welchen beliebige SQL Anweisungen generiert werden können.

Datenbank.sql_anweisungen.**aktualisieren**(*tabelle*, *aktualisierung*, *filtern*)
Der angegebene Eintrag wird mit den jeweiligen Werten aktualisiert

Beschreibung

Funktion mit welcher der angegebene Eintrag oder Einträge aktualisiert werden.

Parameter

•tabelle: Tabelle in welcher ein Eintrag aktualisiert werden soll

•aktualisierung: Die gewünschte Aktualisierung/ der gewünschte neue Wert

•filtern: Der gewünschte Filter.

Datenbank.sql_anweisungen.**entfernen**(*tabelle*, *filtern=None*)
Angegebener Eintrag wird aus der entsprechenden Tabelle gelöscht

Beschreibung

Löscht den angegebenen Eintrag aus der jeweiligen Tabelle.

Parameter

- tabelle: Die Tabelle aus welcher ein Eintrag gelöscht werden soll
- filtern: Wird gesetzt falls nur ein bestimmter oder mehrer Einträge nach einer gewissen Angabe gelöscht werden sollen.

Datenbank.sql_anweisungen.**hinzufugen**(*tabelle*, *felder=''*, *werte=''*)
Ein neuer Eintrag wird in die angegebene Tabelle eingefügt

Beschreibung

Ein neuer Eintrag wird in der angegebenen Tabelle mit den übergebenen Daten (falls gegeben) erstellt. Wurden keine Werte übergeben, wird ein leerer neuer Eintrag erstellt.

Parameter

- tabelle: Tabelle in welcher der neue Eintrag erstellt werden soll
- felder: Die gewünschten Kolonnen welche Werte erhalten sollen
- werte: Die gewünschten Werten mit welchen der neue Eintrag erstellt werden soll

Datenbank.sql_anweisungen.**null_test**(*tabelle*, *null_feld*, *felder*)
Sucht alle Einträge in welchen eine bestimmte Zelle **NULL** *ist*

Beschreibung

Funktion mit welcher erkannt werden kann, welche Felder in einer Tabelle keinen Inhalt (also *NULL*) enthalten.

Parameter

- tabelle: Tabelle welche kontrolliert werden soll
- null_feld: Kolonne welches auf *NULL* Feld kontrolliert werden soll
- felder: Kolonnen welche man erhalten will (z.Bsp.: Schlüsselkolonne)

Datenbank.sql_anweisungen.**sql**(*tabelle*, *select='*'*, *filtern=None*, *gruppieren=None*, *sortieren=None*)
Funktion zur Zusammenstellung einer SQL SELECT Anweisung

Beschreibung

Standard *SQL* Anfrage, welche über Parameterübergaben ausgebaut werden kann.

Wenn keine Parameter übergeben wurden, wird eine *Default* **sql_anweisung** erstellt und als Rückgabewert zurückgegeben.

Wenn Parameter übergeben wurden, wird abgefragt welche Parameter gesetzt wurden und die **sql_anweisung** wird gegebenfalls ausgebaut.

Parameter

- select: Parameter zum setzen der eigentlichen SELECT Anweisung.
- tabelle: Die gewünschte Tabelle
- filtern: Wenn gesetzt, wird eine *WHERE* Anweisung mit dem übergebenem Inhalt erstellt.
- gruppieren: Wenn gesetzt, wird eine *GROUP BY* Anweisung mit dem übergebenem Inhalt erstellt.

•sortieren: Wenn gesetzt, wird eine *ORDER BY* Anweisung mit dem übergebenem Inhalt erstellt.

6.5 Konfiguration

FVS arbeitet mit einer Konfigurationsdatei *konfig.cfg* um die Daten zur Datenbankverbindung abzuspeichern und wieder aufrufen zu können.

Konfiguration.konfig.**laden_konfig**()
Die Konfiguration wird aus konfig.cfg geladen

Rückgabewerte

•einstellungen:

–dbAdresse: Die IP Adresse der Datenbank (Bsp. 127.0.0.1)

–dbPort: Die Port Nummer auf dem die Datenbank läuft

–dbName: Der Name der Datenbank

–dbBenutzer: Der Benutzer der Datenbank

–dbPasswort: Das Passwort des Benutzers

Konfiguration.konfig.**speichern_konfig**(*dbAdresse*, *dbPort*, *dbName*, *dbBenutzer*, *dbPasswort*)
Abspeichern der Konfiguration in konfig.cfg

Parameter

•dbAdresse: Die IP Adresse der Datenbank (Bsp. 127.0.0.1)

•dbPort: Die Port Nummer auf dem die Datenbank läuft

•dbName: Der Name der Datenbank

•dbBenutzer: Der Benutzer der Datenbank

•dbPasswort: Das Passwort des Benutzers

Rückgabewerte

•False: Wird zurückgegeben falls das Abspeichern fehlgeschlagen ist

6.6 Kontroller

Die *Kontroller* dienen dazu die verschiedenen *Signale* der **QMLUI** zu empfangen (*Slots*) und die gewünschten Funktionen aufzurufen.

6.6.1 Operationen

class Kontroller.operationen.**Kontroller**(*basisklasse*, *basisklasse_funktionen*)
Kontroller für Programminterne Signale

Beschreibung

Operationen ist ein Kontroller welche die Signale innerhalb von **FVS** verwaltet. Die Signale werden vom Programm für das Programm gesendet und sind unabhängig des Benutzers. Dieser *Kontroller* wird verwendet um möglichen Problemen zwischen der **QMLUI** und des *Python* Unterbaus zu umgehen (Bsp.: *Python* soll

Variablen von der **QMLUI** einlesen, jedoch ist die **QMLUI** langsamer als *Python* und hat die Variablen noch nicht gesetzt)

operations_signal(*operationID*)
Aufgerufen wenn ein Programminternes Signal gesendet wurde

Beschreibung

Die gewünschte Funktion der Basisklasse wird aufgerufen, abhängig davon von welches Signal gesendet wurde.

6.6.2 Button

class Kontroller.button.**Kontroller**(*basisklasse*, *basisklasse_funktionen*)
Kontroller für die Button Komponente der QMLUI

button_gedruckt(*buttonID*)
Aufgerufen wenn ein Button gedrückt wurde

Beschreibung

Die gewünschte *Funktion* der *Basisklasse* wird aufgerufen, abhängig davon von welchem **Button** das *Signal* ausgegangen ist.

6.6.3 Listen

class Kontroller.listen.**Kontroller**(*basisklasse*, *basisklasse_funktionen*)
Kontroller für eine ListenAnsicht Komponente der QMLUI

eintrag_ausgewaehlt(*packung*, *listeID*)
Aufgerufen wenn Eintrag in der Fahrzeugliste ausgewählt wurde

Beschreibung

Die gewünschte Funktion der Basisklasse wird aufgerufen, abhängig davon von welcher Liste das Signal ausgegangen ist. *packung* wird von der **QMLUI** übergeben und stellt den Inhalt (als **QObject** verpackt) des momentan ausgewählten Eintrages dar.

6.6.4 JaNeinAuswahl

class Kontroller.jaNeinAuswahl.**Kontroller**(*basisklasse*, *basisklasse_funktionen*)
Kontroller für die JaNeinAuswahl Komponente der QMLUI

status_geaendert(*auswahlID*, *status*)
Aufgerufen wenn der Status der JaNeinAuswahl getoggled wurde

Beschreibung Die gewünschte *Funktion* der *Basisklasse* wird aufgerufen, abhängig davon von welcher **JaNeinAuswahl** das *Signal* ausgegangen ist.

Parameter

- auswahlID: Die *ID* der **JaNeinAuswahl**
- status: Der momentane *Status* der **JaNeinAuswahl**

6.7 Logik

Sämtliche *Logik* Module für **FVS** welche verschiedene *Operationen* mit den Daten aus der Datenbank durchführen.

6.7.1 Fahrzeug Kontrollen

Hier sind Funktionen implementiert, welche verwendet werden um Kontrollen und Tests auf Fahrzeuge durchzuführen.

Logik.fahrzeug.**problemTest**(*problem_angabe*)
Gibt an ob ein Problem besteht oder nicht

Beschreibung

Kontrolliert ob für ein Fahrzeug ein technisches Problem besteht oder nicht.

Parameter

•problem_angabe: *Boolean* Wert ob ein technisches Problem besteht oder nicht

Rückgabewerte

•0: Es besteht kein technisches Problem

•1: Es besteht ein technisches Problem

Logik.fahrzeug.**tuvTest**(*tuvdatum*)
Testet TÜV Test

Beschreibung

Funktion welche testet ob der *TÜV* eines Fahrzeuges bereits abgelaufen ist oder nicht.

Parameter

•tuvdatum: Das Datum des letzten *TÜV*

Rückgabewerte

•0: Der *TÜV* ist nicht abgelaufen

•1: Der *TÜV* ist abgelaufen

Logik.fahrzeug.**werkTest**(*werkdatum*)
Testet Werkstattkontrolle Ablaufdatum

Beschreibung

Funktion welche testet ob die *Werkstattkontrolle* bereits abgelaufen ist oder nicht.

Parameter

•werkdatum: Das Datum der letzten *Werkstattkontrolle*

Rückgabewerte

•0: Die *Werkstattkontrolle* ist nicht abgelaufen

•1: Die *Werkstattkontrolle* ist abgelaufen

6.7.2 Inventar Logik

Logik.inventar.**nicht_verwendete_inventargegenstaende**(*db_verbindung*)
Liste aller nicht verwendeten Inventargegenständen

Beschreibung

Führt *SQL* Anfragen in verschiedenen Tabellen durch um festzustellen welche *Inventargegenstände* noch nicht verwendet werden.

Parameter

•db_verbindung: Das Objekt der momentanen Verbindung zur Datenbank

Rückgabewerte

•datensatz: Liste aller bisher noch nicht verwendeten Inventargegenständen

6.7.3 Bericht Erstellung

Logik.bericht_erstellung.**html_bericht**(*daten*, *benutzer*, *datei_name=None*)
Erstellt einen HTML Bericht

Beschreibung

Funktion welche die übergebenen Daten in einen *HTML* Datei als Bericht einbindet und anschließend diesen Bericht im *Standard Browser* des Benutzers anzeigt.

Parameter

•daten: Die Daten aus welchen der HTML Bericht erstellt werden soll

•benutzer: Der Benutzer welcher den Bericht erstellt hat (wird verwendet umd die Dateien später voneinander zu unterscheiden)

6.7.4 Protokoll Logik

Logik.protokoll_logik.**referenzenListe**(*db_verbindung*, *schlussel*)
Zusammenstellung der Referenzen eines Protokolles

Beschreibung

Alle Referenzen welche in einem Protokoll gesetzt wurden, werden ausgelesen und als Liste zurückgegeben.

Parameter

•db_verbindung: Das Objekt der momentanen Verbindung zur Datenbank

•schlussel: Die ID des Protokolls für welche die Referenzen ausgelesen werden sollen

Rückgabewerte

•datensatz: Liste aller Referenzen eines Protokolls

6.7.5 Mitglied Logik

Hier sind Funktionen implementiert welche verwendet werden um Kontrollen und Tests für verschiedene Mitglieder auszuführen.

Logik.mitglied_logik.**medizinischeKontrolle**(*mediDatum*)
Überprüfung der medizinischen Kontrolle

Beschreibung

Funktion welche testet ob die *medizinische Kontrolle* eines Mitglieds bereits abgelaufen ist oder nicht.

Parameter

•mediDatum: Das Datum der letzten medizinischen Kontrolle

Rückgabewerte

•None: Die *medizinische Kontrolle* ist nicht abgelaufen

•1: Die *medizinische Kontrolle* ist abgelaufen

Logik.mitglied_logik.**restliche_stunden**(*stunden*)
Überprüft ob ein Mitglied noch freie Stunden hat

Beschreibung

Funktion welche kontrolliert ob ein Mitglied noch genügend freie Stunden hat um an Lehrgängen teilzunehmen.

Parameter

•stunden: Die Anzahl an restlichen Stunden

Rückgabewerte

•None: Das Mitglied hat noch genügend Stunden

•1: Das Mitglied hat nicht mehr genügend Stunden

6.8 Berichte

6.8.1 Erstellung von Berichten

In diesem Verzeichnis werden die erstellten HTML Berichte von **FVS** gespeichert. Die Berichte werden wie folgt abgespeichert: **bericht<benutzer_name><datum_erstellung>**. Dies gewährleistet, dass die einzelnen Berichte der verschiedenen Benutzer voneinander unterschieden werden können.

Die Berichte sind als *HTML5 & CSS3* [1] Dateien erstellt worden und sind unter allen Browser lauffähig, welche sich an die Vorgaben des w3c [2] halten.

6.8.2 fvs.css

Hierbei handelt es sich um die *CSS* Datei, welche die Berichte verwenden. Es ist eine Standard *Vorlage* erstellt worden, welche jedoch von den Benutzern ganz nach ihren Bedürfnissen angepasst werden kann.

6.9 QMLUI

Dateien welche die *grafische Benutzeroberfläche* darstellen für **FVS**. Die **QMLUI** Dateien bauen ihre Oberflächen aus den einzelnen Komponenten aus **QML_Komponenten** auf.

[1] http://www.w3schools.com/html/html5_intro.asp
[2] http://www.w3.org/

6.9.1 Admin

Die grafische Benutzeroberfläche für den Administrator.

6.9.2 Inventarist

Die grafische Benutzeroberfläche für den Inventaristen.

6.9.3 Kommandant

Die grafische Benutzeroberfläche für den Kommandanten.

6.9.4 Login

Die grafische Benutzeroberfläche für die Benutzeranmeldung.

6.9.5 Maschinist

Die grafische Benutzeroberfläche für den Maschinisten.

6.9.6 Sekretaer

Die grafische Benutzeroberfläche für den Sekretär.

6.10 QML Komponenten

Verschiedene spezifische *QML Komponenten* welche von der **QMLUI** verwendet werden um die *GUI* aufzubauen.

Die *QML Komponenten* mussten manuell designt/erstellt werden, da die verwendete Version von *Qt* und *QML* (siehe *Qt & QML*) keine Standard Komponenten mitlieferte.

Die zuständigen *Kontroller* werden außerdem in den einzelnen Komponenten definiert.

6.10.1 AuswahlListe

Die **AuswahlListe.qml** definiert ein *QML Komponent* welches eine Standard *ComboBox* darstellen soll.

6.10.2 Button

Ein Standard *Button* für die **QMLUI**.

6.10.3 InfoText

Ein Standard *Label* für die **QMLUI**.

6.10.4 JaNeinAuswahl

Eine Standard *CheckBox* für die **QMLUI**.

6.10.5 ListenAnsicht

Eine *ListView* für die **QMLUI**.

6.10.6 ListenContainer

Eine Standard *ListBox* für die **QMLUI**.

6.10.7 Memo

Ein Standard *Memo* Feld für die **QMLUI**.

6.10.8 Overlay

Eine Komponente speziell für **FVS** entwickelt. Es stellt ein *Dropdown* Menü dar. Innerhalb ist ein **ListenContainer** integriert, welche die einzelnen Meldungen auflistet. Das **Overlay** gleitet mit einer einfachen Animation ins Programmfenster hinein.

6.10.9 TextEingabe

Eine Standard *EditBox* für die **QMLUI**.

6.11 UI Aufbau

Die **UI_Aufbau** *Python Module* bauen die **QMLUI** auf und setzen alle Informationen und stellen alle weiteren Funktionen/Methoden zum Gebrauch der **QMLUI** bereit. Es handelt sich um die Hauptdatein von **FVS**.

6.11.1 Inventarist

class `UI_Aufbau.inventarist.`**`QMLUI`**

Hauptklasse der grafischen Benutzeroberfläche des Inventaristen

Beschreibung

Die **QMLUI** (grafische Benutzeroberfläche) des Inventaristen wird aufgebaut, die Verbindung zur Datenbank wird aufgebaut und alle nötigen Signale/Slots und Funktionen zum arbeiten mit der **QMLUI** werden implementiert.

`aktuel_schlauch_liste_entf``()`

Die aktuelle Liste von Schläuchen wird entfernt

`aktueller_schlauch_info_setzen`(*schlauch*)

Der Zustand des aktuellen Schlauches wird gesetzt

Parameter

•schlauch: Verpackter Datensatz eines Schlauchs

atemschutz_abspeichern()
Die Daten eines Atemschutzgerätes werden abgespeichert

atemschutz_aktualisieren()
Signal zum abfragen der Daten aus der QMLUI

atemschutz_beschaedigt_bericht()
Berichterstellung über beschädigte Atemschutzgeräte

atemschutz_entfernen()
Löschen eines Atemschutzgerätes

atemschutz_info_setzen(*atemschutz*)
Die Informationen des momentanen Atemschutzgerätes wird gesetzt

atemschutz_kontrolle_bericht()
Berichterstellung über die Kontrollen der Atemschutzgeräte

atemschutz_liste_setzen(*status=''*)
Die Verwaltung zum Atemschutz wird aufgebaut

Parameter

•status: Gibt an ob die Liste nach beschädigten Geräten gefiltert werden soll oder nicht

auswahl_listen_aufbauen()
Die verschiedenen AuswahlListe Komponenten aufbauen

einzeln_schlauch_entfernen()
Der aktuelle Schlauch wird entfernt

kleider_aktualisieren()
Die Daten der Kleidung werden abgefragt

kleider_entfernen()
Ein Kleidungsgegenstand wird entfernt

kleider_info_setzen(*kleidung*)
Die Daten der Kleidung wird gesetzt

kleider_neu()
Ein neuer Kleidungsgegenstand wird erstellt

kleider_verwaltung()
Die Verwaltung der Kleider wird aufgerufen

kleidung_abspeichern()
Die Daten der Kleidung werden abgespeichert

kommunikation_abspeichern()
Die Daten eines Kommunikationsgerätes werden abgespeichert

kommunikation_aktualisieren()
Die Daten eines Kommunikationsgerätes werden abgefragt

kommunikation_entfernen()
Ein Kommunikationsgerät wird entfernt

kommunikation_info_setzen(*kommunikation*)
Die Daten eines Kommunikationsgerätes werden gesetzt

Parameter

•kommunikation: Verpackter Datensatz eines Kommunikationsgerätes

kommunikation_neu()
Ein neues Kommunikationsgerät wird erstellt

kommunikation_verwaltung()
Die Kommunikationsverwaltung wird aufgerufen

neu_atemschutz()
Erstellung eines neuen Atemschutzgerätes

neu_schlauch_anzahl_abfragen()
Die Anzahl an neuen Schläuchen wird abgefragt

schlauch_bericht_erstellen()
Ein Bericht über alle beschädigten Schläuche wird erstellt

schlauch_filter()
Filtern der Schlauch Liste

Beschreibung

Abhängig davon welche **JaNeinAuswahl** ausgewählt sind:

•chkTypA

•chkTypB

•chkTypC

•chkTypD

•chkZustand,

wird die Liste der verschiedenen Schläuche nach dieser Auswahl gefiltert.

schlauch_gegenstaende_liste()
Die Schlauch Verwaltung wird aufgerufen

schlauch_typ_auswahliste_aufbauen()
AuswahlListe von Schlauch Typs wird aufgebaut

schlauche_hinzufugen()
Neue Schläuche werden der DB hinzugefügt

sonstige_gegenstaende_liste()
Liste sonstiger Inventargegenstände wird generiert

Beschreibung

Eine Liste aller Inventargegenständen welche nicht in eine der Hauptkategorien passen wird generiert.

sonstige_informationen_setzen(*gegenstand*)
Informationen eines sonstitgen Gegenstandes werden gesetzt

Parameter

•gegenstand: Verpackter Datensatz eines sonstigen Gegenstandes

sonstiges_abspeichern()
Die Info eines sonstigen Inventargegenstandes werden abgefragt

sonstiges_entfernen()
Ein sonstiger Inventargegenstand wird entfernt

sonstiges_in_db()
Die Info eines sonst. Gegenstandes werden in die DB gespeichert

Beschreibung

Der Python Unterbau geht die Daten des zuletzt ausgewählten sonstigen Inventargegenstandes aus der **QMLUI** auslesen und speichert diese Daten in die Datenbank ab.

sonstiges_neu()
Ein neuer sonstiger Inventargegenstand wird erstellt

typA_wechsel(*status*)
Wird aufgerufen wenn der Status von chkTypA wechselt

Parameter

- status: Der aktuelle Status von **chkTypA**

typB_wechsel(*status*)
Wird aufgerufen wenn der Status von chkTypB wechselt

Parameter

- status: Der aktuelle Status von **chkTypB**

typC_wechsel(*status*)
Wird aufgerufen wenn der Status von chkTypC wechselt

Parameter

- status: Der aktuelle Status von **chkTypC**

typD_wechsel(*status*)
Wird aufgerufen wenn der Status von chkTypD wechselt

Parameter

- status: Der aktuelle Status von **chkTypD**

zum_hauptmenu()
Der Benutzer geht zurück zum Hauptmenü

zustand_wechsel(*status*)
Wird aufgerufen wenn der Status von chkZustand wechselt

Parameter

- status: Der aktuelle Status von **chkZustand**

6.11.2 Maschinist

class UI_Aufbau.maschinist.**QMLUI**
Hauptklasse der grafischen Benutzeroberfläche des Maschinisten

Beschreibung

Die **QMLUI** (grafische Benutzeroberfläche) des Maschinisten wird aufgebaut, die Verbindung zur Datenbank wird aufgebaut und alle nötigen Signale/Slots und Funktionen zum arbeiten mit der **QMLUI** werden implementiert.

aus_ausstattung_entfernen()
Inventargegenstand wird aus der Fahrzeugausstattung entfernt

ausstattung_anzeigenVerstecken()
Die Ausstattung wird angezeigt oder versteckt

ausstattung_schlussel_einlesen(*ausstattungs_gegenstand*)
Der Schlüssel des momentan ausgewählten Ausstattungegenstandes

Parameter

•ausstattungs_gegenstand: Verpackter Datensatz eines Ausstattung Gegenstandes

ausstattung_verwaltung_aufrufen()
Die Verwaltung der Fahrzeugausstattung wird aufgerufen

ausstattung_verwaltung_stoppen()
Die Verwaltung der Fahrzeugausstattung wird gestoppt

bericht_erstellen()
Ein Bericht wird aus der Meldungsliste erstellt

fahrzeug_entfernen()
Ausgewähltes Fahrzeug wird aus der DB entfernt

fahrzeug_info(*fahrzeug_daten*)
Fahrzeug Informationen werden gesetzt

Parameter

•fahrzeug_daten: Verpackter Datensatz eines Fahrzeug Objektes

fahrzeug_infoFelder_freischaltenSperren()
Felder der Fahrzeuginformationen werden freigeschaltet

fahrzeug_info_abspeichern()
Signal an die QMLUI dass Fahrzeuginformationen abgepseichern

fahrzeugliste_aufbauen()
Fahrzeugliste wird aufgebaut und mit Daten gefüllt

in_db_speichern()
Informationen aus der QMLUI in die DB abgespeichert

inventargegenstand_schlussel_einlesen(*inventar_eintrag*)
Der momentan ausgewählte Inventargegenstand wird gesetzt

Parameter

•inventar_eintrag: Verpackter Datensatz eines Inventargegenstandes

neues_fahrzeug_erstellen()
Ein neues Fahrzeug wird erstellt

overlay_meldungen_aufbau()
Die Meldungen für das Overlay werden aufgebaut

Beschreibung

Methode welche alle benötigten/vorgesehenen Kontrollen/Tests durchführt und diese ins Overlay schreibt.

zur_ausstattung_hinzufugen()
Ein Inventargegenstand wird zur Fahrzeugausstattung hinzugefügt

6.11.3 Sekretär

class UI_Aufbau.sekretaer.**QMLUI**
Hauptklasse der grafischen Benutzeroberfläche des Sekretärs

Beschreibung

Die **QMLUI** (grafische Benutzeroberfläche) des Sekretärs wird aufgebaut, die Verbindung zur Datenbank wird aufgebaut und alle nötigen Signale/Slots und Funktionen zum arbeiten mit der **QMLUI** werden implementiert.

abspeichern_ereignis()
Die Informationen des aktuellen Ereignis werden abgespeichert

Beschreibung

Abhängig davon in welchem State sich die **QMLUI** befindet, werden die zusätzlichen Informationen für einen Einsatz mit abgespeichert.

auswahlSchlusselEreignis(*packung*)
Der Schlüssel des ausgewählten Ereignisses wird gesetzt

Beschreibung

Der Schlüssel des zuletzt ausgewählten Ereignisses wird gesetzt.

Parameter

- packung: **datensatz** des ausgewählten Ereignisses in der Auswahliste

auswahlSchlusselFahrzeug(*packung*)
Der Schlüssel des ausgewählten Fahrzeuges wird gesetzt

Parameter

- packung: Verpackter Datensatz eines Inventargegenstandes

auswahlSchlusselInventar(*packung*)
Der Schlüssel des ausgewählten Inventargegenstandes wird gesetzt

Parameter

- packung: Verpackter Datensatz eines Inventargegenstandes

auswahlSchlusselMitglied(*packung*)
Der Schlüssel des ausgewählten Mitgliedes wird gesetzt

Parameter

- packung: Verpackter Datensatz eines Inventargegenstandes

einsatzInfo(*packung*)
Die Informationen eines Einsatzes werden gesetzt

Parameter

- packung: Verpackter Datensatz eines Ereignisses

einsatz_verwaltung()
Die Einsatzverwaltung wird aufgerufen

entfernen_ereignis()
Das aktuell ausgewählte Ereignis entfernen

entfernen_protokoll()
Ausgewähltes Protokoll wird aus der Datenbank gelöscht

ereignisInfo(*packung*, *einsatz=False*)
Die Informationen eines Ereignisses werden gesetzt

Parameter

- packung: Verpackter Datensatz eines Ereignisses
- einsatz: Wenn *einsatz* auf True steht, werden weitere Informationen eines Einsatzes ausgelesen

ereignis_hinzu()
Ein neues Ereignis wird als Referenz hinzugefügt

ereignis_in_db()
Die Informationen eines Ereignisses werden in die DB gespeichert

Beschreibung

Der Python Unterbau geht die Daten des ausgewählten Ereignisses aus der **QMLUI** auslesen und speichert diese Daten in die Datenbank ab.

ereignis_verwaltung(*einsatz=False*)
Das Menü zur Vewaltung von Ereignissen wird aufgerufen

Parameter

- einsatz: Wenn *einsatz* gesetzt ist, werden nur Ereignisse welche als Einsatz markiert wurden angezeigt

fahrzeug_hinzu()
Ein neues Fahrzeug wird als Referenz hinzugefügt

inventar_hinzu()
Ein neuer Inventargegenstand wird als Referenz hinzugefügt

mitglied_hinzu()
Ein neues Mitglied wird als Referenz hinzugefügt

neu_ereignis()
Ein neues Ereignis wird erstellt

neu_protokoll()
Ein neues Protokoll wird in der Datenbank erstellt

protokoll_bericht()
Ein Bericht des aktuellen Protokolls wird erstellt

Beschreibung

Ein Bericht wird aus dem aktuellen Protokoll erstellt. Als erstes wird der eigentliche Inhalt des Protokolls erstellt und anschließend wird die Liste aller Referenzen ausgelesen und die Informationen der Referenzen werden gesetzt.

protokoll_info_db()
Die Informationen eines Protokolls werden in die DB gespeichert

Beschreibung

Der Python Unterbau geht die Daten des ausgewählten Protokolls aus der **QMLUI** auslesen und speichert diese Daten in die Datenbank ab.

protokoll_info_setzen(*protokoll*)
Die Daten des ausgewählten Protokolls werden gesetzt

Beschreibung

Die Informationen eines Protokolls werden gesetzt und der gesamte **Datensatz** des jeweiligen Protokolls wird in eine Klassenvariable gespeichert, zur späteren weiter verwendung.

Parameter

- protokoll: Der **datensatz** des in der Liste ausgewählten Protokolls

protokoll_speichern()
Der Vorgang zum Abspeichern der Protokolldaten wird vorbereitet

protokoll_verwaltung(*state_wechseln=True*)
Das Menü zur Protokollverwaltung wird aufgebaut

Beschreibung

Das Menü zur Verwaltung der Protokolle wird aufgebaut. Die Hauptauswahl für Protokolle wird aufgebaut und Listen von

- Fahrzeugen
- Mitglieder
- Inventargegenstände
- Ereignisse

werden aufgebaut, welche in *AuswahlListen* (ComboBox) gesetzt werden, mit welchen Referenzen gesetzt werden können.

Parameter

- state_wechseln: *Boolean* Angabe ob nur die einzelnen Listen neu erstellt werden sollen oder auch der *State* der **QMLUI** geändert werden soll.

referenzSchlussel(*packung*)
Der Schlussel der letzten ausgewählten Referenz wird gesetzt

Parameter

- packung: Verpackter Datensatz eines Referenz

referenz_entfernen()
Die momentan ausgewählte Referenz wird gelöscht

teilnehmerAuswahl(*packung*)
Der Schlüssel des für einen neuen Teilnehmer wird gesetzt

Parameter

- packung: Verpackter Datensatz eines Mitglieds

teilnehmerSchlussel(*packung*)
Der Schlüssel des ausgewählten Teilnehmers wird gesetzt

Parameter

- packung: Verpackter Datensatz eines Mitglieds

teilnehmer_entfernen()
Der ausgewählte Teilnehmer wird entfernt

teilnehmer_hinzu()
Der ausgewählte Teilnehmer wird der Liste hinzugefügt

6.11.4 Kommandant

class UI_Aufbau.kommandant.**QMLUI**
Hauptklasse der grafischen Benutzeroberfläche des Kommandanten

Beschreibung

Die **QMLUI** (*grafische Benutzeroberfläche) des Kommandanten wird aufgebaut, die Verbindung zur Datenbank wird aufgebaut und alle nötigen Signale/Slots und Funktionen zum arbeiten mit der **QMLUI** werden implementiert.

ausrustung_bearbeiten()
Das Menü zum bearbeiten der Ausrüstung eines Mitglieds

Beschreibung

Das Menü zur Verwaltung der persönlichen Ausrüstung eines Mitglieds wird gesetzt und aufgerufen. Eine Liste mit allen nicht verwendeten Inventargegenständen wird angezeigt und die momentane Ausrüstung eines Mitglieds wird angezeigt.

ausrustung_entfernen()
Ein Ausrüstungsgegenstand wird von einem Mitglied entfernt

ausrustung_hinzu()
Ein Ausrüstungsgegenstand wird dem Mitglied hinzugefügt

ausrustung_schlussel_setzen(*packung*)
Der Schlüssel des ausgewählten Ausrüstungsgegenstandes setzen

Parameter

- packung: Verpackter Datensatz eines Ausrüstungsgegenstandes

auszeichnung_auswahl_setzen(*packung*)
Schlüssel der ausgewählten Auszeichnung in der Auswahliste setzen

Parameter

- packung: Verpackter Datensatz einer Auswahl einer Auszeichnung

auszeichnung_entfernen()
Eine Auszeichnung wird aus der Liste des Mitglieds entfernt

Beschreibung

Die letzte ausgewählte Auszeichnung aus der Liste der bereits bestehenden Auszeichnungen des Mitglieds wird entfernt.

auszeichnung_hinzu()
Eine neue Auszeichnung wird einem Mitglied hinzugefügt

Beschreibung

Die letzte ausgewählte Auszeichnung aus der Auszeichnungen Auswahlliste wird den bereits bestehenden Auszeichnungen des Mitglieds hinzugefügt.

auszeichnungen_schlussel_setzen(*packung*)
Schlüssel der ausgewählten Auszeichnung wird gesetzt

Parameter

- packung: Verpackter Datensatz einer Auszeichnung

entfernen_mitglied()
Ein Mitglied soll gelöscht werden

Beschreibung

Das zuletzt ausgewählte Mitglied wird aus der Datenbank gelöscht.

fuhrerschein_auswahl_setzen(*packung*)
Schlüssel des ausgewählten Führerschein in der Auswahliste setzen

Parameter

- packung: Verpackter Datensatz einer Auswahl eines Führerscheins

fuhrerschein_entfernen()
Ein Führerschein wird aus der Liste des Mitglieds entfernt

Beschreibung

Der letzte ausgewählte Führerschein aus der Liste der bereits bestehenden Führerscheine des Mitglieds wird entfernt.

fuhrerschein_hinzu()
Ein neuer Führerschein wird einem Mitglied hinzugefügt

Beschreibung

Der letzte ausgewählte Führerschein aus der Führerschein Auswahlliste wird den bereits bestehenden Führerscheinen des Mitglieds hinzugefügt.

fuhrerscheine_schlussel_setzen(*packung*)
Schlüssel des ausgewählten Führerscheins wird gesetzt

Parameter

- packung: Verpackter Datensatz eines Führerscheins

inventar_schlussel_setzen(*packung*)
Der Schlüssel des ausgewählten Inventargegenstandes wird gesetzt

Parameter

- packung: Verpackter Datensatz eines Inventargegenstandes

inventarist_bereich_aufrufen()
Der Bereich des Inventaristen wird aufgerufen

kommandant_bereich_aufbau(*state_wechseln=True*)
Die Informationen für den Bereich des Kommandanten werden gesetzt

Beschreibung

Eine Liste aller Mitglieder welche in der Datenbank abgespeichert sind wird erstellt und gesetzt und die einzelnen Listen aller Lehrgänge, Führerscheine und Auszeichnungen für die Auswahllisten werden erstellt und gesetzt.

Parameter

- state_wechseln: *Boolean* Angabe ob nur die einzelnen Listen neu erstellt werden sollen oder auch der *State* der **QMLUI** geändert werden soll.

lehrgaenge_schlussel_setzen(*packung*)
Schlüssel des ausgewählten Lehrgangs wird gesetzt

Parameter

- packung: Verpackter Datensatz eines Lehrgangs

lehrgang_auswahl_setzen(*packung*)
Schlüssel des ausgewählten Lehrgangs in der Auswahliste setzen

Parameter

- packung: Verpackter Datensatz einer Auswahl eines Lehrgangs

lehrgang_entfernen()
Ein Lehrgang wird aus der Liste des Mitglieds entfernt

Beschreibung

Der letzte ausgewählte Lehrgang aus der Liste der bereits bestehenden Lehrgängen des Mitglieds wird entfernt.

lehrgang_hinzu()
Ein neuer absolvierter Lehrgang einem Mitglied zuweisen

Beschreibung

Der letzte ausgewählte Lehrgang aus der Lehrgang Auswahlliste wird den bereits bestehenden Lehrgängen des Mitglieds hinzugefügt.

maschinist_bereich_aufrufen()
Der Bereich des Maschinisten wird aufgerufen

mitglied_aktualisieren()
Die Informationen eines Mitglieds werden abgefragt zum Speichern

Beschreibung

Die **QMLUI** erhält die Anweisung die Informationen des momentan ausgewählten Mitglieds dem Python Unterbau zu übergeben.

mitglied_info_anzeigen()
Die Mitglied Informationen wieder anzeigen

mitglied_info_in_db_speichern()
Die Informationen eines Mitglieds werden in die DB gespeichert

Beschreibung

Der Python Unterbau geht die Daten des zuletzt ausgewählten Mitglieds aus der **QMLUI** auslesen und speichert diese Daten in die Datenbank ab.

mitglied_info_setzen(*mitglied*)
Die Informationen des ausgewäglten Mitglieds werden gesetzt

Parameter

- mitglied: Verpackter Datensatz eines Mitglieds

neues_mitglied()
Ein neues Mitglied soll erstellt werden

Beschreibung

In der Datenbank wird ein neues Mitglied erstellt, bei welchem die 3 wichtigsten Kolonnen einen Standard Wert erhalten.

overlay_bericht()
Ein Bericht wird aus der Meldungsliste erstellt

overlay_meldungen_aufbau()
Die Meldungen für das Overlay werden aufgebaut

Beschreibung

Methode welche alle benötigten/vorgesehenen Kontrollen/Tests durchführt und diese ins Overlay schreibt.

sekretaer_bereich_aufrufen()
Der Bereich des Sekretärs wird aufgerufen

6.11.5 Admin

class UI_Aufbau.admin.**QMLUI**
Hauptklasse der grafischen Benutzeroberfläche des Admins

Beschreibung

Die **QMLUI** (grafische Benutzeroberfläche) des Admins wird aufgebaut, die Verbindung zur Datenbank wird aufgebaut und alle nötigen Signale/Slots und Funktionen zum arbeiten mit der **QMLUI** werden implementiert.

admin_bereich_aufbau()
Der Admin Bereich wird aufgebaut

Beschreibung

Eine Liste mit allen Tabellen in der Datenbank wird erstellt.

tabelle_kontrollieren()
Ausgewählte Tabelle wird auf Fehler kontrolliert

tabelle_optimieren()
Ausgewählte Tabelle wird optimiert

tabelle_reparieren()
Ausgewählte Tabelle wird repariert

tabellen_name_setzen(*packung*)
Der Name der ausgewählten Tabelle wird gesetzt

Python-Modulindex

d

f

k

l

u

Stichwortverzeichnis

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z