PyAPplus64

Release 1.1.2

Thomas Tuerk

13.11.2023

Inhaltsverzeichnis

KAPITEL 1

Beschreibung

Das Paket *PyAPplus64* enthält eine Sammlung von Python Tools für die Interaktion mit dem ERP-System APplus 6.4. Es sollte auch für andere APplus Versionen nützlich sein.

Zielgruppe sind APplus-Administratoren und Anpassungs-Entwickler. *PyAPplus64* erlaubt u.a.

- **einfacher Zugriff auf SOAP-Schnittstelle des APP-Servers**
  - damit Zugriff auf SysConfig
  - Zugriff auf Tools wie z.B. *nextNumber* für Erzeugung der nächsten Nummer für ein Business-Object
  - ...
- **Zugriff auf APplus DB per direktem DB-Zugriff und mittels SOAP**
  - automatischer Aufruf von *completeSQL*, um per AppServer SQL-Statements um z.B. Mandanten erweitern zu lassen
  - Tools für einfache Benutzung von *useXML*, d.h. für das Einfügen, Löschen und Ändern von Datensätzen mit Hilfe des APP-Servers. Genau wie bei Änderungen an Datensätzen über die Web-Oberfläche und im Gegensatz zum direkten Zugriff über die Datenbank werden dabei evtl. zusätzliche Checks ausgeführt, bestimmte Felder automatisch gesetzt oder bestimmte Aktionen angestoßen.
- **das Duplizieren von Datensätzen**
  - zu kopierende Felder aus XML-Definitionen werden ausgewertet
  - Abhängige Objekte können einfach ebenfalls mit-kopiert werden
  - Änderungen wie beispielsweise Nummer des Objektes möglich
  - Unterstützung für Kopieren dyn. Attribute
  - Anlage neuer Objekte mittels APP-Server
  - Datensätze können zwischen Systemen kopiert und auch in Dateien gespeichert werden
  - Beispiel: Kopieren von Artikeln mit Arbeitsplan und Stückliste zwischen Deploy- und Prod-System
- **einfaches Erstellen von Excel-Reports aus SQL-Abfragen**
  - mittels Pandas und XlsxWriter

  - einfache Wrapper um andere Libraries, spart aber Zeit

- …

In *PyAPplus64* wurden die Features (vielleicht leicht verallgemeinert) implementiert, die ich für konkrete Aufgabenstellungen benötigte. Ich gehe davon aus, dass im Laufe der Zeit weitere Features hinzukommen.

1.1 Warnung

PyAPplus64 erlaubt den schreibenden Zugriff auf die APplus Datenbank und beliebige Aufrufe von SOAP-Methoden. Unsachgemäße Nutzung kann Ihre Daten zerstören. Benutzen Sie *PyAPplus64* daher bitte vorsichtig. Zudem kann ich leider nicht garantieren, dass *PyAPplus64* fehlerfrei ist.

KAPITEL 2

typische Anwendungsfälle

2.1 einfache Admin-Aufgaben

Selten auftretende Admin-Aufgaben lassen sich gut mittels Python-Scripten automatisieren. Es ist sehr einfach möglich, auf die DB, aber auch auf SOAP-Schnittstelle der APP-Serverse zuzugreifen. Zudem ist rudimentärer Zugriff auf ASMX-Seiten implementiert. Ich habe dies vor allem für Wartungsarbeiten an Anpassungstabellen genutzt, die für eigene Erweiterungen entwickelt wurden.

Als triviales Beispiel sucht folgender Code alle *DOCUMENTS* Einträge in Artikeln (angezeigt als *Bild* in *Artikel-Rec.aspx*), für die Datei, auf die verwiesen wird, nicht im Dateisystem existiert. Diese fehlenden Dateien werden ausgegeben und das Feld *DOCUMENTS* gelöscht. Das Löschen erfolgt dabei über *useXML*, so dass die Felder *UPDDATE* und *UPDUSER* korrekt gesetzt werden.

```
import pathlib
import PyAPplus64
import applus_configs
from typing import Optional


def main(confFile: pathlib.Path, updateDB: bool, docDir: Optional[str] = None) ->␣
↪None:
    server = PyAPplus64.applus.applusFromConfigFile(confFile)

    if docDir is None:
        docDir = str(server.scripttool.getInstallPathWebServer().joinpath("DocLib"))

    sql = PyAPplus64.sql_utils.SqlStatementSelect("ARTIKEL")
    sql.addFields("ID", "ARTIKEL", "DOCUMENTS")
    sql.where.addConditionFieldStringNotEmpty("DOCUMENTS")

    for row in server.dbQueryAll(sql):
        doc = pathlib.Path(docDir + row.DOCUMENTS)
        if not doc.exists():
            print("Bild '{}' für Artikel '{}' nicht gefunden".format(doc, row.
↪ARTIKEL))

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
            if updateDB:
                upd = server.mkUseXMLRowUpdate("ARTIKEL", row.ID)
                upd.addField("DOCUMENTS", None)
                upd.update()


if __name__ == "__main__":
    main(applus_configs.serverConfYamlTest, False)
```

Man kann alle Python Bibliotheken nutzen. Als Erweiterung wäre es in obigem Script zum Beispiel einfach möglich, alle BMP-Bilder zu suchen, nach PNG zu konvertieren und den DB-Eintrag anzupassen.

2.2 Ad-hoc Reports

APplus erlaubt es mittels Jasper-Reports, flexible und schöne Reports zu erzeugen. Dies funktioniert gut und ist für regelmäßig benutzte Reports sehr sinnvoll.

Für ad-hoc Reports, die nur sehr selten oder sogar nur einmal benutzt werden, ist die Erstellung eines Jasper-Reports und die Einbindung in APplus jedoch zu zeitaufwändig. Teilweise genügen die Ergebnisse einer SQL-Abfrage, die direkt im MS SQL Server abgesetzt werden kann. Wird es etwas komplizierter oder sollen die Ergebnisse noch etwas verarbeitet werden, bietet sich evtl. Python an.

Folgendes Script erzeugt zum Beispiel eine Excel-Tabelle, mit einer Übersicht, welche Materialen wie oft für Artikel benutzt werden:

```
import PyAPplus64
import applus_configs
import pathlib


def main(confFile: pathlib.Path, outfile: str) -> None:
    server = PyAPplus64.applus.applusFromConfigFile(confFile)

    # Einfache SQL-Anfrage
    sql1 = ("select Material, count(*) as Anzahl from ARTIKEL "
            "group by MATERIAL having MATERIAL is not null "
            "order by Anzahl desc")
    df1 = PyAPplus64.pandas.pandasReadSql(server, sql1)

    # Sql Select-Statements können auch über SqlStatementSelect zusammengebaut
    # werden. Die ist bei vielen, komplizierten Bedingungen teilweise hilfreich.
    sql2 = PyAPplus64.SqlStatementSelect("ARTIKEL")
    sql2.addFields("Material", "count(*) as Anzahl")
    sql2.addGroupBy("MATERIAL")
    sql2.having.addConditionFieldIsNotNull("MATERIAL")
    sql2.order = "Anzahl desc"
    df2 = PyAPplus64.pandas.pandasReadSql(server, sql2)

    # Ausgabe als Excel mit 2 Blättern
    PyAPplus64.pandas.exportToExcel(outfile, [(df1, "Materialien"), (df2,
↪"Materialien 2")], addTable=True)


if __name__ == "__main__":
    main(applus_configs.serverConfYamlTest, "myout.xlsx")
```

Dieses kurze Script nutzt Standard-Pandas Methoden zur Erzeugung der Excel-Datei. Allerdings sind diese Methoden in den Aufrufen von *pandasReadSql* und *exportToExcel* gekapselt, so dass der Aufruf sehr einfach ist. Zudem ist es sehr einfach, die Verbindung zur Datenbank und zum APP-Server mittels der YAML-Konfigdatei herzustellen. Bei diesem Aufruf kann optional ein Nutzer und eine Umgebung übergeben werden, die die Standard-Werte aus der YAML-Datei überschreiben. *pandasReadSql* nutzt intern *completeSQL*, so dass der für die Umgebung korrekte Mandant automatisch verwendet wird.

2.3 Anbindung eigener Tools

Ursprünglich wurde *PyAPplus64* für die Anbindung einer APplus-Anpassung geschrieben. Diese Anpassung ist als Windows-Service auf einem eigenen Rechner installiert und überwacht dort ein bestimmtes Verzeichnis. Bei Änderungen an Dateien in diesem Verzeichnis (Hinzufügen, Ändern, Löschen) werden die Dateien verarbeitet und die Ergebnisse an APplus gemeldet. Dafür werden DB-Operationen aber auch SOAP-Calls benutzt. Ebenso wird auf die SysConf zugegriffen.

Viele solcher Anpassungen lassen sich gut und sinnvoll im App-Server einrichten und als Job regelmäßig aufrufen. Im konkreten Fall wird jedoch für die Verarbeitung der Dateien viel Rechenzeit benötigt. Dies würde dadurch verschlimmert, dass die Dateien nicht auf der gleichen Maschine wie der App-Server liegen und somit viele, relativ langsame Dateizugriffe über das Netzwerk nötig wären. Hinzu kommt, dass die Verarbeitung der Dateien dank existierender Bibliotheken in Python einfacher ist.

PyAPplus64 kann für solche Anpassungen eine interessante Alternative zur Implementierung im App-Server oder zur Entwicklung eines Tools ohne spezielle Bibliotheken sein. Nach Initialisierung des Servers:

```
import PyAPplus64

server = PyAPplus64.applus.applusFromConfigFile("my-applus-server.yaml")
```

bietet *P2APplus64* wie oben demonstriert einfachen lesenden und schreibenden Zugriff auf die DB. Hierbei werden automatisch die für die Umgebung nötigen Mandanten zu den SQL Statements hinzugefügt. Zudem ist einfach ein Zugriff auf die Sysconf möglich:

```
print (server.sysconf.getString("STAMM", "MYLAND"))
print (server.sysconf.getList("STAMM", "EULAENDER"))
```

Dank der Bibliothek *zeep* ist es auch sehr einfach möglich, auf beliebige SOAP-Methoden zuzugreifen. Beispielsweise kann auf die Sys-Config auch händisch, d.h. durch direkten Aufruf einer SOAP-Methode des APP-Servers zugegriffen werden:

```
client = server.getAppClient("p2system", "SysConf");
print (client.service.getString("STAMM", "MYLAND"))
```

KAPITEL 3

Abhängigkeiten

3.1 pyodbc

Für die Datenbankverbindung wird `pyodbc` (`python -m pip install pyodbc`) verwendet. Der passende ODBC Treiber, MS SQL Server 2012 Native Client, wird zusätzlich benötigt. Dieser kann von Microsoft bezogen werden.

3.2 zeep

Die Soap-Library `zeep` wird benutzt (`python -m pip install zeep`).

3.3 requests-negotiate-sspi

Die Authentifzierungsmethode Negotiate Wird für Zugriffe auf ASMX-Seiten benutzt (`python -m pip install requests-negotiate-sspi`). Leider ist dies nur unter Windows verfügbar. Alle anderen Funktionen können aber auch ohne dieses Paket benutzt werden.

3.4 PyYaml

Die Library `pyyaml` wird für Config-Dateien benutzt (`python -m pip install pyyaml`).

3.5 Sphinx

Diese Dokumentation ist mit Sphinx geschrieben. `python -m pip install sphinx`. Dokumentation ist im Unterverzeichnis *docs* zu finden. Sie kann mittels `make.bat html` erzeugt werden, dies ruft intern `sphinx-build -M html source build` auf. Die Dokumentation der Python-API sollte evtl. vorher mittels `sphinx-apidoc -T -f ../src/PyAPplus64 -o source/generated` erzeugt oder aktualisiert werden. Evtl. können 2 Aufrufe von `make.bat html` sinnvoll sein, falls sich die Struktur der Dokumentation ändert. Diese Aufrufe werden von `builddocs.sh` automatisiert.

Die erzeugte Doku findet sich im Verzeichnis `build/html`.

3.6 Pandas / SqlAlchemy / xlsxwriter

Sollen Excel-Dateien mit Pandas erzeugt, werden, so muss Pandas, SqlAlchemy und xlsxwriter installiert sein (*python -m pip install pandas sqlalchemy xlsxwriter*).

3.7 PySimpleGUI und andere

Einige Beispiele benutzen PySimpleGUI (`python -m pip install pysimplegui`) sowie teilweise spezielle Bibliotheken etwa zum Pretty-Printing von SQL (`python -m pip install sqlparse sqlfmt`). Dies sind aber Abhängigkeiten von Beispielen, nicht der Bibliothek selbst.

KAPITEL 4

Beispiele

Im Verzeichnis `examples` finden sich Python Dateien, die die Verwendung von *PyAPplus64* demonstrieren.

4.1 Config-Dateien

Viele Scripte teilen sich Einstellungen. Beispielsweise greifen fast alle Scripte irgendwie auf APplus zu und benötigen Informationen, mit welchem APP-Server, welchem Web-Server und welcher Datenbank sie sich verbinden sollen. Solche Informationen, insbesondere die Passwörter, werden nicht in jedem Script gespeichert, sondern nur in den Config-Dateien. Es bietet sich wohl meist an, 3 Konfigdateien zu erstellen, je eine für das Deploy-, das Test- und das Prod-System. Ein Beispiel ist im Unterverzeichnis `examples/applus-server.yaml` zu finden.

```
# Einstellung für die Verbindung mit dem APP-, Web- und DB-Server.
# Viele der Einstellungen sind im APplus Manager zu finden

appserver : {
  server : "some-server",
  port : 2037,
  user : "asol.projects",
  env  : "default-umgebung" # hier wirklich Umgebung, nicht Mandant verwenden
}
webserver : {
  baseurl : "http://some-server/APplusProd6/",
  user : null, # oft "ASOL.Projects", wenn nicht gesetzt, wird aktueller Windows-
↪Nutzer verwendet
  userDomain : null, # Domain für ASOL.PROJECTS
  password : null # das Passwort
}
dbserver : {
  server : "some-server",
  db : "APplusProd6",
  user : "SA",
  password : "your-db-password"
}
```

Damit nicht in jedem Script immer wieder neu die Konfig-Dateien ausgewählt werden müssen, werden die Konfigs für das Prod-, Test- und Deploy-System in `examples/applus_configs.py` hinterlegt. Diese Datei wird in allen

Scripten importiert, so dass das Config-Verzeichnis und die darin enthaltenen Configs einfach zur Verfügung stehen. Zudem werden in dieser Datei auch alle verwendeten Kombinationen aus System und Umgebung hinterlegt. So kann in Scripten auch eine Auswahl des Systems implementiert werden.

```
import pathlib
from PyAPplus64.applus import APplusServerConfigDescription

basedir = basedir = pathlib.Path(__file__) # Adapt to your needs
configdir = basedir.joinpath("config")

serverConfYamlDeploy = configdir.joinpath("applus-server-deploy.yaml")
serverConfYamlTest = configdir.joinpath("applus-server-test.yaml")
serverConfYamlProd = configdir.joinpath("applus-server-prod.yaml")


serverConfDescProdEnv1 = APplusServerConfigDescription("Prod/Env1",␣
↪serverConfYamlProd, env="Env1")
serverConfDescProdEnv2 = APplusServerConfigDescription("Prod/Env2",␣
↪serverConfYamlProd, env="Env2")
serverConfDescTestEnv1 = APplusServerConfigDescription("Test/Env1",␣
↪serverConfYamlTest, env="Env1")
serverConfDescTestEnv2 = APplusServerConfigDescription("Test/Env2",␣
↪serverConfYamlTest, env="Env2")
serverConfDescDeploy = APplusServerConfigDescription("Deploy", serverConfYamlDeploy)

serverConfDescs = [
  serverConfDescProdEnv1,
  serverConfDescProdEnv2,
  serverConfDescTestEnv1,
  serverConfDescTestEnv2,
  serverConfDescDeploy
]
```

4.2 read_settings.py

Einfaches Beispiel für Auslesen der SysConf und bestimmter Einstellungen.

```
# Einfaches Script, das verschiedene Werte des Servers ausliest.
# Dies sind SysConfig-Einstellungen, aber auch der aktuelle Mandant,
# Systemnamen, ...

import pathlib
import PyAPplus64
import applus_configs
from typing import Optional, Union


def main(confFile: Union[str, pathlib.Path], user: Optional[str] = None, env:␣
↪Optional[str] = None) -> None:
    server = PyAPplus64.applusFromConfigFile(confFile, user=user, env=env)

    print("\n\nSysConf Lookups:")

    print("  Default Auftragsart:", server.sysconf.getString("STAMM",
↪"DEFAULTAUFTRAGSART"))
    print("  Auftragsarten:")
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    arten = server.sysconf.getList("STAMM", "AUFTRAGSART", sep='\n')
    if not arten:
        arten = []
    for a in arten:
        print("    - " + a)

    print("  Firmen-Nr. automatisch vergeben:", server.sysconf.getBoolean("STAMM",
↪"FIRMAAUTOMATIK"))
    print("  Anzahl Artikelstellen:", server.sysconf.getInt("STAMM",
↪"ARTKLASSIFNRLAENGE"))

    print("\n\nScriptTool:")

    print("  CurrentDate:", server.scripttool.getCurrentDate())
    print("  CurrentTime:", server.scripttool.getCurrentTime())
    print("  CurrentDateTime:", server.scripttool.getCurrentDateTime())
    print("  LoginName:", server.scripttool.getLoginName())
    print("  UserName:", server.scripttool.getUserName())
    print("  UserFullName:", server.scripttool.getUserFullName())
    print("  SystemName:", server.scripttool.getSystemName())
    print("  Mandant:", server.scripttool.getMandant())
    print("  MandantName:", server.scripttool.getMandantName())
    print("  InstallPath:", server.scripttool.getInstallPath())
    print("  InstallPathAppServer:", server.scripttool.getInstallPathAppServer())
    print("  InstallPathWebServer:", server.scripttool.getInstallPathWebServer())
    print("  ServerInfo - Version:", server.scripttool.getServerInfo().find("version
↪").text)

    client = server.getWebClient("dbenv/dbenv.asmx")
    print("WEB Environment:", client.service.getEnvironment())

if __name__ == "__main__":
    main(applus_configs.serverConfYamlTest)
```

4.3 check_dokumente.py

Einfaches Beispiel für lesenden und schreibenden Zugriff auf APplus Datenbank.

```
import pathlib
import PyAPplus64
import applus_configs
from typing import Optional


def main(confFile: pathlib.Path, updateDB: bool, docDir: Optional[str] = None) ->␣
↪None:
    server = PyAPplus64.applus.applusFromConfigFile(confFile)

    if docDir is None:
        docDir = str(server.scripttool.getInstallPathWebServer().joinpath("DocLib"))

    sql = PyAPplus64.sql_utils.SqlStatementSelect("ARTIKEL")
    sql.addFields("ID", "ARTIKEL", "DOCUMENTS")
    sql.where.addConditionFieldStringNotEmpty("DOCUMENTS")
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    for row in server.dbQueryAll(sql):
        doc = pathlib.Path(docDir + row.DOCUMENTS)
        if not doc.exists():
            print("Bild '{}' für Artikel '{}' nicht gefunden".format(doc, row.
↪ARTIKEL))

            if updateDB:
                upd = server.mkUseXMLRowUpdate("ARTIKEL", row.ID)
                upd.addField("DOCUMENTS", None)
                upd.update()


if __name__ == "__main__":
    main(applus_configs.serverConfYamlTest, False)
```

4.4 adhoc_report.py

Sehr einfaches Beispiel zur Erstellung einer Excel-Tabelle aus einer SQL-Abfrage.

```
import PyAPplus64
import applus_configs
import pathlib


def main(confFile: pathlib.Path, outfile: str) -> None:
    server = PyAPplus64.applus.applusFromConfigFile(confFile)

    # Einfache SQL-Anfrage
    sql1 = ("select Material, count(*) as Anzahl from ARTIKEL "
            "group by MATERIAL having MATERIAL is not null "
            "order by Anzahl desc")
    df1 = PyAPplus64.pandas.pandasReadSql(server, sql1)

    # Sql Select-Statements können auch über SqlStatementSelect zusammengebaut
    # werden. Die ist bei vielen, komplizierten Bedingungen teilweise hilfreich.
    sql2 = PyAPplus64.SqlStatementSelect("ARTIKEL")
    sql2.addFields("Material", "count(*) as Anzahl")
    sql2.addGroupBy("MATERIAL")
    sql2.having.addConditionFieldIsNotNull("MATERIAL")
    sql2.order = "Anzahl desc"
    df2 = PyAPplus64.pandas.pandasReadSql(server, sql2)

    # Ausgabe als Excel mit 2 Blättern
    PyAPplus64.pandas.exportToExcel(outfile, [(df1, "Materialien"), (df2,
↪"Materialien 2")], addTable=True)


if __name__ == "__main__":
    main(applus_configs.serverConfYamlTest, "myout.xlsx")
```

4.5 mengenabweichung.py

Etwas komplizierteres Beispiel zur Erstellung einer Excel-Datei aus SQL-Abfragen.

```
# Erzeugt Excel-Tabellen mit Werkstattaufträgen und Werkstattauftragspositionen mit Mengenabweichungen

import datetime
import PyAPplus64
import applus_configs
import pandas as pd  # type: ignore
import pathlib
from typing import Tuple, Union, Optional


def ladeAlleWerkstattauftragMengenabweichungen(
        server: PyAPplus64.APplusServer,
        cond: Union[PyAPplus64.SqlCondition, str, None] = None) -> pd.DataFrame:
    sql = PyAPplus64.sql_utils.SqlStatementSelect("WAUFTRAG w")
    sql.addLeftJoin("personal p", "w.UPDUSER = p.PERSONAL")

    sql.addFieldsTable("w", "ID", "BAUFTRAG", "POSITION")
    sql.addFields("(w.MENGE-w.MENGE_IST) as MENGENABWEICHUNG")
    sql.addFieldsTable("w", "MENGE", "MENGE_IST",
                       "APLAN as ARTIKEL", "NAME as ARTIKELNAME")
    sql.addFields("w.UPDDATE", "p.NAME as UPDNAME")

    sql.where.addConditionFieldGe("w.STATUS", 5)
    sql.where.addCondition("abs(w.MENGE-w.MENGE_IST) > 0.001")
    sql.where.addCondition(cond)
    sql.order = "w.UPDDATE"
    dfOrg = PyAPplus64.pandas.pandasReadSql(server, sql)

    # Add Links
    df = dfOrg.copy()
    df = df.drop(columns=["ID"])
    # df = df[['POSITION', 'BAUFTRAG', 'MENGE']] # reorder / filter columns

    df['POSITION'] = PyAPplus64.pandas.mkHyperlinkDataframeColumn(
                        dfOrg,
                        lambda r: r.POSITION,
                        lambda r: server.makeWebLinkWauftrag(
                            bauftrag=r.BAUFTRAG, accessid=r.ID))
    df['BAUFTRAG'] = PyAPplus64.pandas.mkHyperlinkDataframeColumn(
                        dfOrg,
                        lambda r: r.BAUFTRAG,
                        lambda r: server.makeWebLinkBauftrag(bauftrag=r.BAUFTRAG))

    colNames = {
        "BAUFTRAG": "Betriebsauftrag",
        "POSITION": "Pos",
        "MENGENABWEICHUNG": "Mengenabweichung",
        "MENGE": "Menge",
        "MENGE_IST": "Menge-Ist",
        "ARTIKEL": "Artikel",
        "ARTIKELNAME": "Artikel-Name",
        "UPDDATE": "geändert am",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        "UPDNAME": "geändert von"
    }
    df.rename(columns=colNames, inplace=True)

    return df


def ladeAlleWerkstattauftragPosMengenabweichungen(
        server: PyAPplus64.APplusServer,
        cond: Union[PyAPplus64.SqlCondition, str, None] = None) -> pd.DataFrame:
    sql = PyAPplus64.sql_utils.SqlStatementSelect("WAUFTRAGPOS w")
    sql.addLeftJoin("personal p", "w.UPDUSER = p.PERSONAL")

    sql.addFieldsTable("w", "ID", "BAUFTRAG", "POSITION", "AG")
    sql.addFields("(w.MENGE-w.MENGE_IST) as MENGENABWEICHUNG")
    sql.addFieldsTable("w", "MENGE", "MENGE_IST", "APLAN as ARTIKEL")
    sql.addFields("w.UPDDATE", "p.NAME as UPDNAME")

    sql.where.addConditionFieldEq("w.STATUS", 4)
    sql.where.addCondition("abs(w.MENGE-w.MENGE_IST) > 0.001")
    sql.where.addCondition(cond)
    sql.order = "w.UPDDATE"

    dfOrg = PyAPplus64.pandas.pandasReadSql(server, sql)

    # Add Links
    df = dfOrg.copy()
    df = df.drop(columns=["ID"])
    df['POSITION'] = PyAPplus64.pandas.mkHyperlinkDataframeColumn(
                        dfOrg,
                        lambda r: r.POSITION,
                        lambda r: server.makeWebLinkWauftrag(
                            bauftrag=r.BAUFTRAG, accessid=r.ID))
    df['BAUFTRAG'] = PyAPplus64.pandas.mkHyperlinkDataframeColumn(
                        dfOrg,
                        lambda r: r.BAUFTRAG,
                        lambda r: server.makeWebLinkBauftrag(bauftrag=r.BAUFTRAG))
    df['AG'] = PyAPplus64.pandas.mkHyperlinkDataframeColumn(
                        dfOrg,
                        lambda r: r.AG,
                        lambda r: server.makeWebLinkWauftragPos(
                            bauftrag=r.BAUFTRAG, position=r.POSITION, accessid=r.ID))

    # Demo zum Hinzufügen einer berechneten Spalte
    # df['BAUFPOSAG'] = PyAPplus64.pandas.mkDataframeColumn(dfOrg,
    #                     lambda r: "{}.{} AG {}".format(r.BAUFTRAG, r.POSITION, r.
↪AG))

    # Rename Columns
    colNames = {
        "BAUFTRAG": "Betriebsauftrag",
        "POSITION": "Pos",
        "AG": "AG",
        "MENGENABWEICHUNG": "Mengenabweichung",
        "MENGE": "Menge",
        "MENGE_IST": "Menge-Ist",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        "ARTIKEL": "Artikel",
        "UPDDATE": "geändert am",
        "UPDNAME": "geändert von"
    }
    df.rename(columns=colNames, inplace=True)
    return df


def computeInYearMonthCond(field: str, year: Optional[int] = None,
                           month: Optional[int] = None) -> Optional[PyAPplus64.SqlCondition]:
    if not (year is None):
        if month is None:
            return PyAPplus64.sql_utils.SqlConditionDateTimeFieldInYear(field, year)
        else:
            return PyAPplus64.sql_utils.SqlConditionDateTimeFieldInMonth(field, year, month)
    else:
        return None


def computeFileName(year: Optional[int] = None, month: Optional[int] = None) -> str:
    if year is None:
        return 'mengenabweichungen-all.xlsx'
    else:
        if month is None:
            return 'mengenabweichungen-{:04d}.xlsx'.format(year)
        else:
            return 'mengenabweichungen-{:04d}-{:02d}.xlsx'.format(year, month)


def _exportInternal(server: PyAPplus64.APplusServer, fn: str,
                    cond: Union[PyAPplus64.SqlCondition, str, None]) -> int:
    df1 = ladeAlleWerkstattauftragMengenabweichungen(server, cond)
    df2 = ladeAlleWerkstattauftragPosMengenabweichungen(server, cond)
    print("erzeuge " + fn)
    PyAPplus64.pandas.exportToExcel(fn, [(df1, "WAuftrag"), (df2, "WAuftrag-Pos")], addTable=True)
    return len(df1.index) + len(df2.index)


def exportVonBis(server: PyAPplus64.APplusServer, fn: str,
                 von: Optional[datetime.datetime], bis: Optional[datetime.datetime]) -> int:
    cond = PyAPplus64.sql_utils.SqlConditionDateTimeFieldInRange("w.UPDDATE", von, bis)
    return _exportInternal(server, fn, cond)


def exportYearMonth(server: PyAPplus64.APplusServer,
                    year: Optional[int] = None, month: Optional[int] = None) -> int:
    cond = computeInYearMonthCond("w.UPDDATE", year=year, month=month)
    fn = computeFileName(year=year, month=month)
    return _exportInternal(server, fn, cond)


```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
def computePreviousMonthYear(cyear: int, cmonth: int) -> Tuple[int, int]:
    if cmonth == 1:
        return (cyear-1, 12)
    else:
        return (cyear, cmonth-1)


def computeNextMonthYear(cyear: int, cmonth: int) -> Tuple[int, int]:
    if cmonth == 12:
        return (cyear+1, 1)
    else:
        return (cyear, cmonth+1)


def main(confFile: Union[str, pathlib.Path], user: Optional[str] = None, env:␣
↪Optional[str] = None) -> None:
    server = PyAPplus64.applusFromConfigFile(confFile, user=user, env=env)

    now = datetime.date.today()
    (cmonth, cyear) = (now.month, now.year)
    (pyear, pmonth) = computePreviousMonthYear(cyear, cmonth)

    # Ausgaben
    exportYearMonth(server, cyear, cmonth)  # Aktueller Monat
    exportYearMonth(server, pyear, pmonth)  # Vorheriger Monat
    # export(cyear) # aktuelles Jahr
    # export(cyear-1) # letztes Jahr
    # export() # alles


if __name__ == "__main__":
    main(applus_configs.serverConfYamlTest)
```

4.6 mengenabweichung_gui.py

Beispiel für eine sehr einfache GUI, die die Eingabe einfacher Parameter erlaubt. Die GUI wird um die Erzeugung von Excel-Dateien mit Mengenabweichungen gebaut.

```
import PySimpleGUI as sg  # type: ignore
import mengenabweichung
import datetime
import PyAPplus64
import applus_configs
import pathlib
from typing import Tuple, Optional, Union


def parseDate(dateS: str) -> Tuple[Optional[datetime.datetime], bool]:
    if dateS is None or dateS == '':
        return (None, True)
    else:
        try:
            return (datetime.datetime.strptime(dateS, '%d.%m.%Y'), True)
        except:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
            sg.popup_error("Fehler beim Parsen des Datums '{}'".format(dateS))
            return (None, False)


def createFile(server: PyAPplus64.APplusServer, fileS: str, vonS: str, bisS: str) ->␣
↪None:
    (von, vonOK) = parseDate(vonS)
    if not vonOK:
        return

    (bis, bisOK) = parseDate(bisS)
    if not bisOK:
        return

    if (fileS is None) or fileS == '':
        sg.popup_error("Es wurde keine Ausgabedatei ausgewählt.")
        return
    else:
        file = pathlib.Path(fileS)

    c = mengenabweichung.exportVonBis(server, file.as_posix(), von, bis)
    sg.popup_ok("{} Datensätze erfolgreich in Datei '{}' geschrieben.".format(c,␣
↪file))


def main(confFile: Union[str, pathlib.Path], user: Optional[str] = None, env:␣
↪Optional[str] = None) -> None:
    server = PyAPplus64.applusFromConfigFile(confFile, user=user, env=env)

    layout = [
        [sg.Text(('Bitte geben Sie an, für welchen Zeitraum die '
                  'Mengenabweichungen ausgegeben werden sollen:'))],
        [sg.Text('Von (einschließlich)', size=(15, 1)), sg.InputText(key='Von'),
         sg.CalendarButton("Kalender", close_when_date_chosen=True,
                           target="Von", format='%d.%m.%Y')],
        [sg.Text('Bis (ausschließlich)', size=(15, 1)), sg.InputText(key='Bis'),
         sg.CalendarButton("Kalender", close_when_date_chosen=True,
                           target="Bis", format='%d.%m.%Y')],
        [sg.Text('Ausgabedatei', size=(15, 1)), sg.InputText(key='File'),
         sg.FileSaveAs(button_text="wählen",
                       target="File",
                       file_types=(('Excel Files', '*.xlsx'),),
                       default_extension=".xlsx")],
        [sg.Button("Aktueller Monat"), sg.Button("Letzter Monat"),
         sg.Button("Aktuelles Jahr"), sg.Button("Letztes Jahr")],
        [sg.Button("Speichern"), sg.Button("Beenden")]
    ]

    systemName = server.scripttool.getSystemName() + "/" + server.scripttool.
↪getMandant()
    window = sg.Window("Mengenabweichung " + systemName, layout)
    now = datetime.date.today()
    (cmonth, cyear) = (now.month, now.year)
    (pyear, pmonth) = mengenabweichung.computePreviousMonthYear(cyear, cmonth)
    (nyear, nmonth) = mengenabweichung.computeNextMonthYear(cyear, cmonth)

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    while True:
        event, values = window.read()
        if event == sg.WIN_CLOSED or event == 'Beenden':
            break
        if event == 'Aktueller Monat':
            window['Von'].update(value="01.{:02d}.{:04d}".format(cmonth, cyear))
            window['Bis'].update(value="01.{:02d}.{:04d}".format(nmonth, nyear))
        if event == 'Letzter Monat':
            window['Von'].update(value="01.{:02d}.{:04d}".format(pmonth, pyear))
            window['Bis'].update(value="01.{:02d}.{:04d}".format(cmonth, cyear))
        if event == 'Aktuelles Jahr':
            window['Von'].update(value="01.01.{:04d}".format(cyear))
            window['Bis'].update(value="01.01.{:04d}".format(cyear+1))
        if event == 'Letztes Jahr':
            window['Von'].update(value="01.01.{:04d}".format(cyear-1))
            window['Bis'].update(value="01.01.{:04d}".format(cyear))
        if event == 'Speichern':
            try:
                createFile(server, values.get('File', None),
                           values.get('Von', None), values.get('Bis', None))
            except Exception as e:
                sg.popup_error_with_traceback("Beim Erzeugen der Excel-Datei trat ein␣
↪Fehler auf:", e)

    window.close()


if __name__ == "__main__":
    main(applus_configs.serverConfYamlProd)
```

4.7 complete_sql.pyw

Beispiel, wie ein einfacher APP-Server Aufruf über eine GUI zur Verfügung gestellt und mittels Python-Bibliotheken erweitert werden kann. Zudem wird demonstriert, wie eine Auswahl verschiedenere Systeme und Umgebungen realisiert werden kann.

```
import PySimpleGUI as sg  # type: ignore
import PyAPplus64
import applus_configs
import pathlib
from typing import Tuple, Optional, Union

try:
    import sqlparse
except:
    pass

try:
    import sqlfmt.api
except:
    pass

def prettyPrintSQL(format, sql):
    try:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        if format == "sqlfmt":
            mode = sqlfmt.api.Mode(dialect_name="ClickHouse")
            sqlPretty = sqlfmt.api.format_string(sql, mode)
            return sqlPretty.replace("N '", "N'") # fix String Constants
        elif format == "sqlparse-2":
            return sqlparse.format(sql, reindent=True, keyword_case='upper')
        elif format == "sqlparse":
            return sqlparse.format(sql, reindent_aligned=True, keyword_case='upper')
        else:
            return sql
    except e:
        print (str(e))
        return sql

def main() -> None:
    monospaceFont = ("Courier New", 12)
    sysenvs = applus_configs.serverConfDescs[:];
    sysenvs.append("-");
    layout = [
        [sg.Button("Vervollständigen"), sg.Button("aus Clipboard", key="import"), sg.Button("nach Clipboard", key="export"), sg.Button("zurücksetzen", key="clear"), sg.Button("Beenden"),
         sg.Text('System/Umgebung:'), sg.Combo(sysenvs, default_value="-", key="sysenv", readonly=True), sg.Text('Formatierung:'), sg.Combo(["-", "sqlfmt", "sqlparse", "sqlparse-2"], default_value="sqlparse", key="formatieren", readonly=True)
         ],
        [sg.Text('Eingabe-SQL')],
        [sg.Multiline(key='input', size=(150, 20), font=monospaceFont)],
        [sg.Text('Ausgabe-SQL')],
        [sg.Multiline(key='output', size=(150, 20), font=monospaceFont, horizontal_scroll=True)]
    ]

    # server = PyAPplus64.applusFromConfigFile(confFile, user=user, env=env)
    # systemName = server.scripttool.getSystemName() + "/" + server.scripttool.getMandant()
    window = sg.Window("Complete SQL", layout)
    oldSys = None
    while True:
        event, values = window.read()
        if event == sg.WIN_CLOSED or event == 'Beenden':
            break
        elif event == 'clear':
             window['input'].update("")
             window['output'].update("")
        elif event == 'import':
            try:
                window['input'].update(window.TKroot.clipboard_get())
            except:
                window['input'].update("")
            window['output'].update("")
        elif event == 'export':
            try:
                window.TKroot.clipboard_clear()
                window.TKroot.clipboard_append(window['output'].get())
            except:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
                pass
        elif event == 'Vervollständigen':
            sqlIn = window['input'].get()
            try:
                if sqlIn:
                    sys = window['sysenv'].get()
                    if sys != oldSys:
                        oldSys = sys
                        if sys and sys != "-":
                            server = sys.connect()
                        else:
                            server = None
                    if server:
                        sqlOut = server.completeSQL(sqlIn)
                    else:
                        sqlOut = sqlIn
                    sqlOut = prettyPrintSQL(window['formatieren'].get(), sqlOut)
                else:
                    sqlOut = ""
            except Exception as e:
                sqlOut = "ERROR: " + str(e)
                sg.popup_error_with_traceback("Fehler bei Vervollständigung", e)
            window['output'].update(value=sqlOut)

    window.close()


if __name__ == "__main__":
    main()
```

4.8 importViewUDF.py

Folgende Scripte erlauben den einfachen Import von DB-Anpass-Dateien, Views und UDFs über den Windows-Explorer. Werden Verknüpfungen zu den Scripten `importViewUDFDeploy.pyw` und `importViewUDFTest.pyw` in `%appdata%\Microsoft\Windows\SendTo` abgelegt, so können eine oder mehrerer solcher Dateien mittels _Kontextmenü (Rechtsklick) - Senden **an**_ an APplus zur Verarbeitung übergeben werden. Dabei ist es wichtig, dass sich die Dateien im für den jeweiligen Typ passenden Verzeichnis befinden.

```
import PySimpleGUI as sg  # type: ignore
import pathlib
import PyAPplus64
from PyAPplus64 import applus
from PyAPplus64 import sql_utils
import applus_configs
import traceback
import pathlib
import sys

def importViewsUDFs(server, views, udfs, dbanpass):
  res = ""
  try:
    if views or udfs:
      for env in server.scripttool.getAllEnvironments():
        res = res + server.importUdfsAndViews(env, views, udfs);
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        res = res + "\n\n";

    for xml in dbanpass:
      res = res + "Verarbeite " + xml + "\n"
      xmlRes = server.updateDatabase(xml);
      if (xmlRes == ""): xmlRes = "OK";
      res = res + xmlRes
      res = res + "\n\n"

    sg.popup_scrolled("Importiere", res)
  except:
    sg.popup_error("Fehler", traceback.format_exc())

def importIntoSystem(server, system):
  try:
    if (len(sys.argv) < 2):
      sg.popup_error("Keine Datei zum Import übergeben")
      return

    views = []
    udfs = []
    dbanpass = []
    errors = ""


    for i in range (1, len(sys.argv)):
      arg = pathlib.Path(sys.argv[i])
      if arg == server.scripttool.getInstallPathAppServer().joinpath("Database", "View
↪", arg.stem + ".sql"):
        views.append(arg.stem)
      elif arg == server.scripttool.getInstallPathAppServer().joinpath("Database",
↪"UDF", arg.stem + ".sql"):
        udfs.append(arg.stem)
      elif arg == server.scripttool.getInstallPathAppServer().joinpath("DBChange",␣
↪arg.stem + ".xml"):
        dbanpass.append(arg.stem + ".xml")
      else:
        errors = errors + "  - " + str(arg) + "\n";

    if len(errors) > 0:
      msg = "Folgende Dateien sind keine View, UDF oder DB-Anpass-Dateien des
↪"+system+"-Systems:\n" + errors;
      sg.popup_error("Fehler", msg);
    if views or udfs or dbanpass:
      importViewsUDFs(server, views, udfs, dbanpass)

  except:
    sg.popup_error("Fehler", traceback.format_exc())

if __name__ == "__main__":
  server = PyAPplus64.applusFromConfigFile(applus_configs.serverConfYamlDeploy)
  importIntoSystem(server, "Deploy");
```

Wrapper für Deploy-System:

```
import importViewUDF
import applus_configs
import PyAPplus64

if __name__ == "__main__":
  server = PyAPplus64.applusFromConfigFile(applus_configs.serverConfYamlDeploy)
  importViewUDF.importIntoSystem(server, "Deploy")
```

Wrapper für Test-System:

```
import importViewUDF
import applus_configs
import PyAPplus64

if __name__ == "__main__":
  server = PyAPplus64.applusFromConfigFile(applus_configs.serverConfYamlTest)
  importViewUDF.importIntoSystem(server, "Test")
```

4.9 copy_artikel.py

Beispiel, wie Artikel inklusive Arbeitsplan und Stückliste dupliziert werden kann.

```
import pathlib
import PyAPplus64
import applus_configs
import logging
import yaml
from typing import Optional


def main(confFile: pathlib.Path, artikel: str, artikelNeu: Optional[str] = None) ->␣
↪None:
    # Server verbinden
    server = PyAPplus64.applus.applusFromConfigFile(confFile)

    # DuplicateBusinessObject für Artikel erstellen
    dArt = PyAPplus64.duplicate.loadDBDuplicateArtikel(server, artikel)

    # DuplicateBusinessObject zur Demonstration in YAML konvertieren und zurück
    dArtYaml = yaml.dump(dArt)
    print(dArtYaml)
    dArt2 = yaml.load(dArtYaml, Loader=yaml.UnsafeLoader)

    # Neue Artikel-Nummer bestimmen und DuplicateBusinessObject in DB schreiben
    # Man könnte hier genauso gut einen anderen Server verwenden
    if (artikelNeu is None):
        artikelNeu = server.nextNumber("Artikel")

    if not (dArt is None):
        dArt.setFields({"artikel": artikelNeu})
        res = dArt.insert(server)
        print(res)


if __name__ == "__main__":
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
33      # Logger Einrichten
34      logging.basicConfig(level=logging.INFO)
35      # logger = logging.getLogger("PyAPplus64.applus_db");
36      # logger.setLevel(logging.ERROR)
37
38      main(applus_configs.serverConfYamlTest, "my-artikel", artikelNeu="my-artikel-copy
    ↪")
```

KAPITEL 5

PyAPplus64 package

5.1 Submodules

5.2 PyAPplus64.applus module

class PyAPplus64.applus.**APplusServer**(*db_settings:* PyAPplus64.applus_db.APplusDBSettings, *server_settings:* PyAPplus64.applus_server.APplusServerSettings)

Bases: object

Verbindung zu einem APplus DB und App Server mit Hilfsfunktionen für den komfortablen Zugriff.

Parameter

- **db_settings** (*APplusDBSettings*) – die Einstellungen für die Verbindung mit der Datenbank
- **server_settings** (*APplusAppServerSettings*) – die Einstellungen für die Verbindung mit dem APplus App Server
- **web_settings** (*APplusWebServerSettings*) – die Einstellungen für die Verbindung mit dem APplus Web Server

property client_adaptdb: zeep.client.Client

property client_nummer: zeep.client.Client

property client_table: zeep.client.Client

property client_xml: zeep.client.Client

completeSQL(*sql: Union[*PyAPplus64.sql_utils.SqlStatementSelect*, str]*, *raw: bool = False*) → str

Vervollständigt das SQL-Statement. Es wird z.B. der Mandant hinzugefügt.

Parameter

- **sql** (*sql_utils.SqlStatement*) – das SQL Statement
- **raw** (*boolean*) – soll completeSQL ausgeführt werden? Falls True, wird die Eingabe zurückgeliefert

Rückgabe das vervollständigte SQL-Statement

Rückgabetyp str

dbExecute(*sql: Union[*PyAPplus64.sql_utils.SqlStatementSelect*, str]*, **args: Any*, *raw: bool = False*) → Any

Führt ein SQL Statement (z.B. update oder insert) aus. Das SQL wird zunächst vom Server angepasst, so dass z.B. Mandanteninformation hinzugefügt werden.

dbQuery(*sql: Union[*PyAPplus64.sql_utils.SqlStatementSelect*, str]*, *f: Callable[[pyodbc.Row], None]*, **args: Any*, *raw: bool = False*) → None

Führt eine SQL Query aus und führt für jede Zeile die übergeben Funktion aus. Das SQL wird zunächst vom Server angepasst, so dass z.B. Mandanteninformation hinzugefügt werden.

dbQueryAll(*sql: Union[*PyAPplus64.sql_utils.SqlStatementSelect*, str]*, **args: Any*, *raw: bool = False*, *apply: Optional[Callable[[pyodbc.Row], Any]] = None*) → Any

Führt eine SQL Query aus und liefert alle Zeilen zurück. Das SQL wird zunächst vom Server angepasst, so dass z.B. Mandanteninformation hinzugefügt werden.

dbQuerySingleRow(*sql: Union[*PyAPplus64.sql_utils.SqlStatementSelect*, str]*, **args: Any*, *raw: bool = False*) → Optional[pyodbc.Row]

Führt eine SQL Query aus, die maximal eine Zeile zurückliefern soll. Diese Zeile wird geliefert.

dbQuerySingleRowDict(*sql: Union[*PyAPplus64.sql_utils.SqlStatementSelect*, str]*, **args: Any*, *raw: bool = False*) → Optional[Dict[str, Any]]

Führt eine SQL Query aus, die maximal eine Zeile zurückliefern soll. Diese Zeile wird als Dictionary geliefert.

dbQuerySingleValue(*sql: Union[*PyAPplus64.sql_utils.SqlStatementSelect*, str]*, **args: Any*, *raw: bool = False*) → Any

Führt eine SQL Query aus, die maximal einen Wert zurückliefern soll. Dieser Wert oder None wird geliefert.

dbQuerySingleValues(*sql: Union[*PyAPplus64.sql_utils.SqlStatementSelect*, str]*, **args: Any*, *raw: bool = False*) → Sequence[Any]

Führt eine SQL Query aus, die nur eine Spalte zurückliefern soll.

db_settings: *PyAPplus64.applus_db.APplusDBSettings*

Die Einstellungen für die Datenbankverbindung

execUseXMLRowDelete(*table: str*, *id: int*) → None

getAppClient(*package: str*, *name: str*) → zeep.client.Client

Erzeugt einen zeep - Client für den APP-Server. Mittels dieses Clients kann eines WSDL Schnittstelle des APP-Servers angesprochen werden. Wird als *package* „p2core“ und als *name* „Table“ verwendet und der resultierende client „client“ genannt, dann kann z.B. mittels „client.service.getCompleteSQL(sql)“ vom AppServer ein Vervollständigen des SQLs angefordert werden.

Parameter

- **package** (*string*) – das Packet, z.B. „p2core“
- **name** – der Name im Packet, z.B. „Table“

Rückgabe den Client

Rückgabetyp Client

getDBConnection() → pyodbc.Connection

Liefert eine pyodbc-Connection zur APplus DB. Diese muss genutzt werden, wenn mehrere Operationen in einer Transaktion genutzt werden sollen. Ansonsten sind die Hilfsmethoden wie *APplusServer.dbQuery()* zu bevorzugen. Diese Connection kann in Verbindung mit den Funktionen aus *PyAPplus64.applus_db* genutzt werden. Die Verbindung sollte nach Benutzung wieder freigegeben oder geschlossen werden.

getTableFields(*table: str*, *isComputed: Optional[bool] = None*) → Set[str]

Liefert die Namen aller Felder einer Tabelle.

Parameter

- **table** – Name der Tabelle
- **isComputed** – wenn gesetzt, werden nur die Felder geliefert, die berechnet werden oder nicht berechnet werden

Rückgabe Liste aller Feld-Namen

Rückgabetyp {str}

getUniqueFieldsOfTable(*table: str*) → Dict[str, List[str]]
Liefert alle Spalten einer Tabelle, die eindeutig sein müssen. Diese werden als Dictionary gruppiert nach Index-Namen geliefert. Jeder Eintrag enthält eine Liste von Feldern, die zusammen eindeutig sein müssen.

getWebClient(*url: str*) → zeep.client.Client
Erzeugt einen zeep - Client für den Web-Server. Mittels dieses Clients kann die von einer ASMX-Seite zur Verfügung gestellte Schnittstelle angesprochen werden. Als parameter wird die relative URL der ASMX-Seite erwartet. Die Base-URL automatisch ergänzt. Ein Beispiel für eine solche relative URL ist „masterdata/artikel.asmx“.

ACHTUNG: Als Umgebung wird die Umgebung des sich anmeldenden Nutzers verwendet. Sowohl Nutzer als auch Umgebung können sich von den für App-Clients verwendeten Werten unterscheiden. Wenn möglich, sollte ein App-Client verwendet werden.

Parameter **url** – die relative URL der ASMX Seite, z.B. „masterdata/artikel.asmx“

Rückgabe den Client

Rückgabetyp Client

importUdfsAndViews()
Importiert bestimmte Views und UDFs :param environment: die Umgebung, in die Importiert werden soll :type environment: string :param views: Views, die importiert werden sollen :type views: [string] :param udfs: Views, die importiert werden sollen :type udfs: [string] :return: Infos zur Ausführung :rtype: str

isDBTableKnown(*table: str*) → bool
Prüft, ob eine Tabelle im System bekannt ist

job: *PyAPplus64.applus_job.APplusJob*
erlaubt Arbeiten mit Jobs

makeWebLink(*base: str*, ***kwargs: Any*) → str

makeWebLinkAuftrag(***kwargs: Any*) → str

makeWebLinkBauftrag(***kwargs: Any*) → str

makeWebLinkVKRahmen(***kwargs: Any*) → str

makeWebLinkWarenaugang(***kwargs: Any*) → str

makeWebLinkWauftrag(***kwargs: Any*) → str

makeWebLinkWauftragPos(***kwargs: Any*) → str

mkUseXMLRowDelete(*table: str*, *id: int*) → *PyAPplus64.applus_usexml.UseXmlRowDelete*

mkUseXMLRowInsert(*table: str*) → *PyAPplus64.applus_usexml.UseXmlRowInsert*
Erzeugt ein Objekt zum Einfügen eines neuen DB-Eintrags.

Parameter **table** (*str*) – DB-Tabelle in die eingefügt werden soll

Rückgabe das XmlRow-Objekt

Rückgabetyp *applus_usexml.UseXmlRowInsert*

mkUseXMLRowInsertOrUpdate(*table: str*) → *PyAPplus64.applus_usexml.UseXmlRowInsertOrUpdate*
Erzeugt ein Objekt zum Einfügen oder Updaten eines DB-Eintrags.

Parameter **table** (*string*) – DB-Tabelle in die eingefügt werden soll

Rückgabe das XmlRow-Objekt

Rückgabetyp *applus_usexml.UseXmlRowInsertOrUpdate*

mkUseXMLRowUpdate(*table: str*, *id: int*) → *PyAPplus64.applus_usexml.UseXmlRowUpdate*

nextNumber(*obj: str*) → str
Erstellt eine neue Nummer für das Objekt und legt diese Nummer zurück.

reconnectDB() → None

releaseDBConnection(*conn: pyodbc.Connection*) → None
Gibt eine DB-Connection zur Wiederverwendung frei

scripttool: *PyAPplus64.applus_scripttool.APplusScriptTool*
erlaubt den einfachen Zugriff auf Funktionen des ScriptTools

server_conn: *PyAPplus64.applus_server.APplusServerConnection*
erlaubt den Zugriff auf den AppServer

server_settings: *PyAPplus64.applus_server.APplusServerSettings*
Einstellung für die Verbindung zum APP- und Webserver

sysconf: *PyAPplus64.applus_sysconf.APplusSysConf*
erlaubt den Zugriff auf die Sysconfig

updateDatabase(*file: str*) → str
Führt eine DBAnpass-xml Datei aus. :param file: DB-Anpass Datei, die ausgeführt werden soll :type file: string :return: Infos zur Ausführung :rtype: str

useXML(*xml: str*) → Any
Ruft `p2core.xml.usexml` auf. Wird meist durch ein `UseXMLRow-Objekt` aufgerufen.

class PyAPplus64.applus.**APplusServerConfigDescription**(*descr: str*, *yamlfile: FileDescriptorOrPath*, *user: Optional[str] = None*, *env: Optional[str] = None*)

Bases: `object`

Beschreibung einer Configuration bestehend aus Config-Datei, Nutzer und Umgebung.

Parameter

- **descr** (*str*) – Beschreibung als String, nur für Ausgabe gedacht
- **yamlfile** (*'FileDescriptorOrPath'*) – die Datei
- **user** (*Optional[str]*) – der Nutzer
- **env** (*Optional[str]*) – die Umgebung

connect() → *PyAPplus64.applus.APplusServer*

PyAPplus64.applus.**applusFromConfig**(*yamlString: str*, *user: Optional[str] = None*, *env: Optional[str] = None*) → *PyAPplus64.applus.APplusServer*
Läd Einstellungen aus einer Config-Datei und erzeugt daraus ein APplus-Objekt

PyAPplus64.applus.**applusFromConfigDict**(*yamlDict: Dict[str, Any]*, *user: Optional[str] = None*, *env: Optional[str] = None*) → *PyAPplus64.applus.APplusServer*
Läd Einstellungen aus einer Config und erzeugt daraus ein APplus-Objekt

PyAPplus64.applus.**applusFromConfigFile**(*yamlfile: FileDescriptorOrPath*, *user: Optional[str] = None*, *env: Optional[str] = None*) → *PyAPplus64.applus.APplusServer*
Läd Einstellungen aus einer Config-Datei und erzeugt daraus ein APplus-Objekt

5.3 PyAPplus64.applus_db module

class PyAPplus64.applus_db.**APplusDBSettings**(*server: str*, *database: str*, *user: str*, *password: str*)
Bases: object

Einstellungen, mit welcher DB sich verbunden werden soll.

connect() → pyodbc.Connection
Stellt eine neue Verbindung her und liefert diese zurück.

getConnectionString() → str
Liefert den ODBC Connection-String für die Verbindung. :return: den Connection-String

class PyAPplus64.applus_db.**DBTableIDs**
Bases: object

Klasse, die Mengen von IDs gruppiert nach Tabellen speichert

add(*table: str*, **ids: int*) → None
fügt Eintrag hinzu

Parameter

- **table** (*str*) – die Tabelle
- **id** – die ID

getTable(*table: str*) → Set[int]
Liefert die Menge der IDs für eine bestimmte Tabelle.

Parameter **table** (*str*) – die Tabelle

Rückgabe die IDs

PyAPplus64.applus_db.**getUniqueFieldsOfTable**(*cnxn: pyodbc.Connection*, *table: str*) → Dict[str, List[str]]
Liefert alle Spalten einer Tabelle, die eindeutig sein müssen. Diese werden als Dictionary gruppiert nach Index-Namen geliefert. Jeder Eintrag enthält eine Liste von Feldern, die zusammen eindeutig sein müssen.

PyAPplus64.applus_db.**rawExecute**(*cnxn: pyodbc.Connection*, *sql: Union[PyAPplus64.sql_utils.SqlStatementSelect, str]*, **args: Any*) → Any
Führt ein SQL Statement direkt aus

PyAPplus64.applus_db.**rawQuery**(*cnxn: pyodbc.Connection*, *sql: Union[PyAPplus64.sql_utils.SqlStatementSelect, str]*, *f: Callable[[pyodbc.Row], None]*, **args: Any*) → None
Führt eine SQL Query direkt aus und führt für jede Zeile die übergeben Funktion aus.

PyAPplus64.applus_db.**rawQueryAll**(*cnxn: pyodbc.Connection*, *sql: Union[PyAPplus64.sql_utils.SqlStatementSelect, str]*, **args: Any*, *apply: Optional[Callable[[pyodbc.Row], Any]] = None*) → Sequence[Any]
Führt eine SQL Query direkt aus und liefert alle Zeilen zurück. Wenn apply gesetzt ist, wird die Funktion auf jeder Zeile ausgeführt und das Ergebnis ausgeben, die nicht None sind.

PyAPplus64.applus_db.**rawQuerySingleRow**(*cnxn: pyodbc.Connection*, *sql: Union[PyAPplus64.sql_utils.SqlStatementSelect, str]*, **args: Any*) → Optional[pyodbc.Row]
Führt eine SQL Query direkt aus, die maximal eine Zeile zurückliefern soll. Diese Zeile wird geliefert.

PyAPplus64.applus_db.**rawQuerySingleValue**(*cnxn: pyodbc.Connection*, *sql: Union[PyAPplus64.sql_utils.SqlStatementSelect, str]*, **args: Any*) → Any
Führt eine SQL Query direkt aus, die maximal einen Wert zurückliefern soll. Dieser Wert oder None wird geliefert.

PyAPplus64.applus_db.**row_to_dict**(*row: pyodbc.Row*) → Dict[str, Any]
Konvertiert eine Zeile in ein Dictionary

5.4 PyAPplus64.applus_job module

class PyAPplus64.applus_job.**APplusJob**(*server: APplusServer*)
Bases: object

Zugriff auf Jobs

Parameter **server** (APplusServer) – die Verbindung zum Server

property client: zeep.client.Client

createSOAPJob(*bez: str*) → str
Erzeugt einen neuen SOAP Job mit der gegebenen Bezeichnung und liefert die neue JobID. :param bez: die Bezeichnung des neuen Jobs :type bez: str :return: die neue JobID :rtype: str

finish(*jobId: str*, *status: int*, *resurl: str*) → bool
Beendet den Job :param jobId: die ID des Jobs :type jobId: str :param status: der Status 2 (OK), 3 (Fehler) :type status: int :param resurl: die neue resurl des Jobs :type resurl: str

getInfo(*jobId: str*) → str
Liefert die Info eines Jobs :param jobId: die ID des Jobs :type jobId: str :return: die Info :rtype: str

getPtURL(*jobId: str*) → str
Liefert die PtURL eines Jobs :param jobId: die ID des Jobs :type jobId: str :return: die Pt-URL :rtype: str

getResult(*jobId: str*) → str
Liefert das Result eines Jobs :param jobId: die ID des Jobs :type jobId: str :return: das Result :rtype: str

getResultURL(*jobId: str*) → str
Liefert die ResultURL eines Jobs :param jobId: die ID des Jobs :type jobId: str :return: die Result-URL :rtype: str

getResultURLString(*jobId: str*) → Optional[str]
Liefert die ResultURL eines Jobs, wobei ein evtl. Präfix „retstring://“ entfernt wird und alle anderen Werte durch None ersetzt werden. :param jobId: die ID des Jobs :type jobId: str :return: die Result-URL als String :rtype: str

getStatus(*jobId: str*) → str
Liefert Informationen zum Job :param jobId: die ID des Jobs :type jobId: str :return: die Infos :rtype: str

kill(*jobId: str*) → None
Startet einen Job :param jobId: die ID des Jobs :type jobId: str

restart(*jobId: str*) → str
Startet einen Job neu :param jobId: die ID des Jobs :type jobId: str :return: die URL des Jobs :rtype: str

setInfo(*jobId: str*, *info: str*) → bool
Setzt die Informationen zu dem Job :param jobId: die ID des Jobs :type jobId: str :param info: die neuen Infos :type info: str

setPosition(*jobId: str*, *pos: int*, *max: int*) → bool
Schrittfunktion :param jobId: die ID des Jobs :type jobId: str :param pos: Position :type pos: int :param max: Anzahl Schritte in Anzeige :type max: int

setPtURL(*jobId: str*, *pturl: str*) → None
Setzt die ResultURL eines Jobs :param jobId: die ID des Jobs :type jobId: str :param pturl: die neue PtURL :type pturl: str

setResult(*jobId: str*, *res: str*) → None
Setzt das Result eines Jobs :param jobId: die ID des Jobs :type jobId: str :param res: das neue Result :type res: str

setResultURL(*jobId: str*, *resurl: str*) → None
Setzt die ResultURL eines Jobs :param jobId: die ID des Jobs :type jobId: str :param resurl: die neue Result-URL :type resurl: str

start(*jobId: str*) → bool
Startet einen Job :param jobId: die ID des Jobs :type jobId: str

5.5 PyAPplus64.applus_scripttool module

class PyAPplus64.applus_scripttool.**APplusScriptTool**(*server:* PyAPplus64.applus.APplusServer)
Bases: object

Zugriff auf AppServer ScriptTool

Parameter **server** (APplusServerConnection) – die Verbindung zum Server

property client: zeep.client.Client

getAllEnvironments()
Liefert alle Umgebungen

Rückgabe die gefundenen Umgebungen

Rückgabetyp [str]

getCurrentDate() → str

getCurrentDateTime() → str

getCurrentTime() → str

getInstallPath() → str
Liefert den Installionspfad des Appservers

getInstallPathAppServer() → pathlib.Path
Liefert den Installionspfad des Appservers als PathLib-Path

getInstallPathWebServer() → pathlib.Path
Liefert den Installionspfad des Webservers als PathLib-Path

getLoginName() → str

getMandant() → str
Liefert den aktuellen Mandanten

getMandantName() → str
Liefert den Namen des aktuellen Mandanten

getServerInfo() → Optional[lxml.etree.Element]
Liefert Informationen zum Server als ein XML Dokument.

Rückgabe das gefundene und geparste XML-Dokument

Rückgabetyp ET.Element

getServerInfoString() → str
Liefert Informationen zum Server als String. Dieser String repräsentiert ein XML Dokument.

Rückgabe das XML-Dokument als String

Rückgabetyp str

getSystemName() → str

getUserFullName() → str

getUserName() → str

getXMLDefinition(*obj: str*, *mandant: str = ''*, *checkFileExists: bool = False*) → Optional[lxml.etree.Element]

Läd die XML-Definition als String vom APPServer. und parst das XML in ein minidom-Dokument.

Parameter

- **obj** (*str*) – das Objekt, dessen Definition zu laden ist, „Artikel“ läd z.B. „ArtikelDefinition.xml“
- **mandant** (*str optional*) – der Mandant, dessen XML-Doku geladen werden soll, wenn „“ wird der Standard-Mandant verwendet

Rückgabe das gefundene und geparste XML-Dokument

Rückgabetyp ET.Element

getXMLDefinitionObj(*obj: str*, *mandant: str = ''*) → Optional[*PyAPplus64.applus_scripttool.XMLDefinition*]

Benutzt getXMLDefinitionObj und liefert den Top-Level „Object“ Knoten zurück, falls zusätzlich ein MD5 Knoten existiert, also falls das Dokument wirklich vom Dateisystem geladen werden konnte. Ansonten wird None geliefert.

Parameter

- **obj** (*str*) – das Objekt, dess Definition zu laden ist, „Artikel“ läd z.B. „ArtikelDefinition.xml“
- **mandant** (*str optional*) – der Mandant, dessen XML-Doku geladen werden soll, wenn „“ wird der Standard-Mandant verwendet

Rückgabe das gefundene und geparste XML-Dokument

Rückgabetyp Optional[*XMLDefinition*]

getXMLDefinitionString(*obj: str*, *mandant: str = ''*) → str

Läd die XML-Defintion als String vom APPServer. Auch wenn kein XML-Dokument im Dateisystem gefunden wird, wird ein String zurückgeliefert, der einen leeren Top-„Object“ Knoten enthält. Für gefundene XML-Dokumente gibt es zusätzlich einen Top-„MD5“-Knoten.

Parameter

- **obj** (*str*) – das Objekt, dessen Definition zu laden ist, „Artikel“ läd z.B. „ArtikelDefinition.xml“
- **mandant** (*str optional*) – der Mandant, dessen XML-Doku geladen werden soll, wenn „“ wird der Standard-Mandant verwendet

Rückgabe das gefundene XML-Dokument als String

Rückgabetyp str

class PyAPplus64.applus_scripttool.**XMLDefinition**(*root: lxml.etree.Element*)

Bases: object

Repräsentation eines XML-Dokuments

getDuplicate() → Tuple[Set[str], bool]

Extrahiert alle Properties, die für Duplizieren konfiguriert sind. Zudem wird ein Flag geliefert, ob diese Properties ein oder ausgeschlossen werden sollen. :return: Tuple aus allen Properties und ob dies aus-(True) oder ein-(False) zuschließen sind. :rtype: Tuple[Set[str], bool]

root: lxml.etree.Element

das Root-Element, repräsentiert „object“ aus Datei.

5.6 PyAPplus64.applus_server module

class PyAPplus64.applus_server.**APplusServerConnection**(*settings:* PyAPplus64.applus_server.APplusServerSettings)

Bases: object

Verbindung zu einem APplus APP- und Web-Server

Parameter **settings** (*APplusAppServerSettings*) – die Einstellungen für die Verbindung mit dem APplus Server

getAppClient(*package: str*, *name: str*) → zeep.client.Client

Erzeugt einen zeep - Client für den APP-Server. Mittels dieses Clients kann die WSDL Schnittstelle angesprochen werden. Wird als *package* „p2core“ und als *name* „Table“ verwendet und der resultierende client „client“ genannt, dann kann z.B. mittels „client.service.getCompleteSQL(sql)“ vom AppServer ein Vervollständigen des SQLs angefordert werden.

Parameter

- **package** (*string*) – das Packet, z.B. „p2core“
- **name** – der Name im Packet, z.B. „Table“

Rückgabe den Client

Rückgabetyp Client

getWebClient(*url: str*) → zeep.client.Client

Erzeugt einen zeep - Client für den Web-Server. Mittels dieses Clients kann die von einer ASMX-Seite zur Verfügung gestellte Schnittstelle angesprochen werden. Als parameter wird die relative URL der ASMX-Seite erwartet. Die Base-URL automatisch ergänzt. Ein Beispiel für eine solche relative URL ist „masterdata/artikel.asmx“.

ACHTUNG: Als Umgebung wird die Umgebung des sich anmeldenden Nutzers verwendet. Sowohl Nutzer als auch Umgebung können sich von den für App-Clients verwendeten Werten unterscheiden. Wenn möglich, sollte ein App-Client verwendet werden.

Parameter **url** – die relative URL der ASMX Seite, z.B. „masterdata/artikel.asmx“

Rückgabe den Client

Rückgabetyp Client

class PyAPplus64.applus_server.**APplusServerSettings**(*webserver: str*, *appserver: str*, *appserverPort: int*, *user: str*, *env: Optional[str] = None*, *webserverUser: Optional[str] = None*, *webserverUserDomain: Optional[str] = None*, *webserverPassword: Optional[str] = None*)

Bases: object

Einstellungen, mit welchem APplus App- and Web-Server sich verbunden werden soll.

5.7 PyAPplus64.applus_sysconf module

class PyAPplus64.applus_sysconf.**APplusSysConf**(*server: APplusServer*)
Bases: object

SysConf Zugriff mit Cache über AppServer

Parameter **server** (APplusServer) – die Verbindung zum Server

clearCache() → None

property client: zeep.client.Client

getBoolean(*module: str*, *name: str*, *useCache: bool = True*) → bool

getDouble(*module: str*, *name: str*, *useCache: bool = True*) → float

getInt(*module: str*, *name: str*, *useCache: bool = True*) → int

getList(*module: str*, *name: str*, *useCache: bool = True*, *sep: str = ','*) → Optional[Sequence[str]]

getString(*module: str*, *name: str*, *useCache: bool = True*) → str

5.8 PyAPplus64.applus_usexml module

class PyAPplus64.applus_usexml.**UseXmlRow**(*applus: APplusServer*, *table: str*, *cmd: str*)
Bases: object

Klasse, die eine XML-Datei erzeugen kann, die mittels p2core.useXML genutzt werden kann. Damit ist es möglich APplus BusinessObjekte zu erzeugen, ändern und zu löschen. Im Gegensatz zu direkten DB-Zugriffen, werden diese Anfragen über den APP-Server ausgeführt. Dabei werden die von der Weboberfläche bekannten Checks und Änderungen ausgeführt. Als sehr einfaches Beispiel wird z.B. INSDATE oder UPDDATE automatisch gesetzt. Interessanter sind automatische Änderungen und Checks.

Bei der Benutzung wird zunächst ein Objekt erzeugt, dann evtl. mittels *addField()* Felder hinzugefügt und schließlich mittels *exec()* an den AppServer übergeben. Normalerweise sollte die Klasse nicht direkt, sondern über Unterklassen für das Einfügen, Ändern oder Löschen benutzt werden.

Parameter

- **applus** (APplusServer) – Verbindung zu APplus
- **table** (*str*) – die Tabelle
- **cmd** (*str*) – cmd-attribut der row, also ob es sich um ein Update, ein Insert oder ein Delete handelt

addField(*name: Optional[str]*, *value: Any*) → None
Fügt ein Feld zum Row-Node hinzu.

Parameter

- **name** (*string*) – das Feld
- **value** – Wert des Feldes

addTimestampField(*id: int*, *ts: Optional[bytes] = None*) → None
Fügt ein Timestamp-Feld hinzu. Wird kein Timestamp übergeben, wird mittels der ID der aktuelle Timestamp aus der DB geladen. Dabei kann ein Fehler auftreten. Ein Timestamp-Feld ist für Updates und das Löschen nötig um sicherzustellen, dass die richtige Version des Objekts geändert oder gelöscht wird. Wird z.B. ein Objekt aus der DB geladen, inspiziert und sollen dann Änderungen gespeichert werden, so sollte der Timestamp des Ladens übergeben werden. So wird sichergestellt, dass nicht ein anderer User zwischenzeitlich Änderungen vornahm. Ist dies der Fall, wird dann bei „exec“ eine Exception geworfen.

Parameter

- **id** (*string*) – DB-id des Objektes dessen Timestamp hinzugefügt werden soll
- **ts** (*bytes*) – Fester Timestamp der verwendet werden soll, wenn None wird der Timestamp aus der DB geladen.

addTimestampIDFields(*id: int*, *ts: Optional[bytes] = None*) → None
Fügt ein Timestamp-Feld sowie ein Feld id hinzu. Wird kein Timestamp übergeben, wird mittels der ID der aktuelle Timestamp aus der DB geladen. Dabei kann ein Fehler auftreten. Intern wird *addTimestampField()* benutzt.

Parameter

- **id** (*string*) – DB-id des Objektes dessen Timestamp hinzugefügt werden soll
- **ts** (*bytes*) – Fester Timestamp der verwendet werden soll, wenn None wird der Timestamp aus der DB geladen.

checkFieldSet(*name: Optional[str]*) → bool
Prüft, ob ein Feld gesetzt wurde

checkFieldsSet(**names: str*) → bool
Prüft, ob alle übergebenen Felder gesetzt sind

exec() → Any
Führt die UseXmlRow mittels useXML aus. Je nach Art der Zeile wird etwas zurückgeliefert oder nicht. In jedem Fall kann eine Exception geworfen werden.

getField(*name: str*) → Any
Liefert den Wert eines gesetzten Feldes

toprettyxml() → str
Gibt das formatierte XML aus. Dieses kann per useXML an den AppServer übergeben werden. Dies wird mittels *exec()* automatisiert.

class PyAPplus64.applus_usexml.**UseXmlRowDelete**(*applus: APplusServer*, *table: str*, *id: int*, *ts: Optional[bytes] = None*)
Bases: *PyAPplus64.applus_usexml.UseXmlRow*

Klasse, die eine XML-Datei für das Löschen eines neuen Datensatzes erzeugen kann. Die Felder *id* und *timestamp* werden automatisch gesetzt. Dies sind die einzigen Felder, die gesetzt werden sollten.

Parameter

- **applus** (APplusServer) – Verbindung zu APplus
- **table** (*string*) – die Tabelle
- **id** (*int*) – die zu löschende ID
- **ts** (*bytes optional*) – wenn gesetzt, wird dieser Timestamp verwendet, sonst der aktuelle aus der DB

delete() → None
Führt das delete aus. Evtl. wird dabei eine Exception geworfen. Dies ist eine Umbenennung von `exec()`.

class PyAPplus64.applus_usexml.**UseXmlRowInsert**(*applus: APplusServer*, *table: str*)
Bases: *PyAPplus64.applus_usexml.UseXmlRow*

Klasse, die eine XML-Datei für das Einfügen eines neuen Datensatzes erzeugen kann.

Parameter

- **applus** (APplusServer) – Verbindung zu APplus
- **table** (*string*) – die Tabelle

insert() → int
Führt das insert aus. Entweder wird dabei eine Exception geworfen oder die ID des neuen Eintrags zurückgegeben. Dies ist eine Umbenennung von `exec()`.

class PyAPplus64.applus_usexml.**UseXmlRowInsertOrUpdate**(*applus: APplusServer*, *table: str*)

Bases: *PyAPplus64.applus_usexml.UseXmlRow*

Klasse, die eine XML-Datei für das Einfügen oder Ändern eines neuen Datensatzes, erzeugen kann. Die Methode *checkExists* erlaubt es zu prüfen, ob ein Objekt bereits existiert. Dafür werden die gesetzten Felder mit den Feldern aus eindeutigen Indices verglichen. Existiert ein Objekt bereits, wird ein Update ausgeführt, ansonsten ein Insert. Bei Updates werden die Felder *id* und *timestamp* automatisch gesetzt.

Parameter

- **applus** (APplusServer) – Verbindung zu APplus
- **table** (*string*) – die Tabelle

checkExists() → Optional[int]

Prüft, ob der Datensatz bereits in der DB existiert. Ist dies der Fall, wird die ID geliefert, sonst None

exec() → int

Führt entweder ein Update oder ein Insert durch. Dies hängt davon ab, ob das Objekt bereits in der DB existiert. In jedem Fall wird die ID des erzeugten oder geänderten Objekts geliefert.

insert() → int

Führt ein Insert aus. Existiert das Objekt bereits, wird eine Exception geworfen.

update(*id: Optional[int] = None*, *ts: Optional[bytes] = None*) → int

Führt ein Update aus. Falls ID oder Timestamp nicht übergeben werden, wird nach einem passenden Objekt gesucht. Existiert das Objekt nicht, wird eine Exception geworfen.

updateOrInsert() → int

Führt das update oder das insert aus. Evtl. wird dabei eine Exception geworfen. Dies ist eine Umbenennung von *exec()*. Es wird die ID des Eintrages geliefert

class PyAPplus64.applus_usexml.**UseXmlRowUpdate**(*applus: APplusServer*, *table: str*, *id: int*, *ts: Optional[bytes] = None*)

Bases: *PyAPplus64.applus_usexml.UseXmlRow*

Klasse, die eine XML-Datei für das Ändern eines neuen Datensatzes, erzeugen kann. Die Felder *id* und *timestamp* werden automatisch gesetzt.

Parameter

- **applus** (APplusServer) – Verbindung zu APplus
- **table** (*string*) – die Tabelle
- **id** (*int*) – die ID des zu ändernden Datensatzes
- **ts** (*bytes optional*) – wenn gesetzt, wird dieser Timestamp verwendet, sonst der aktuelle aus der DB

update() → None

Führt das update aus. Evtl. wird dabei eine Exception geworfen. Dies ist eine Umbenennung von `exec()`.

5.9 PyAPplus64.duplicate module

Dupliziert ein oder mehrere APplus Business-Objekte

class PyAPplus64.duplicate.**DuplicateBusinessObject**(*table: str*, *fields: Dict[str, Any]*, *fieldsNotCopied: Dict[str, Any] = {}*, *allowUpdate: bool = False*)

Bases: `object`

Klasse, die alle Daten zu einem BusinessObject speichert und zum Duplizieren dieses Objektes dient. Dies beinhaltet Daten zu abhängigen Objekten sowie die Beziehung zu diesen Objekten. Zu einem Artikel wird

z.B. der Arbeitsplan gespeichert, der wiederum Arbeitsplanpositionen enthält. Als Beziehung ist u.a. hinterlegt, dass das Feld „APLAN“ der Arbeitsplans dem Feld „ARTIKEL“ des Artikels entsprechen muss und dass „APLAN“ aus den Positionen, „APLAN“ aus dem APlan entsprichen muss. So kann beim Duplizieren ein anderer Name des Artikels gesetzt werden und automatisch die Felder der abhängigen Objekte angepasst werden. Einige Felder der Beziehung sind dabei statisch, d.h. können direkt aus den zu speichernden Daten abgelesen werden. Andere Felder sind dynamisch, d.h. das Parent-Objekt muss in der DB angelegt werden, damit ein solcher dynamischer Wert erstellt und geladen werden kann. Ein typisches Beispiel für ein dynamisches Feld ist „GUID“.

addDependentBusinessObject(*dObj: Optional[PyAPplus64.duplicate.DuplicateBusinessObject]*, **args: Tuple[str, str]*) → None

Fügt ein neues Unterobjekt zum DuplicateBusinessObject hinzu. Dabei handelt es sich selbst um ein DuplicateBusinessObject, das zusammen mit dem Parent-Objekt dupliziert werden sollen. Zum Beispiel sollen zu einem Auftrag auch die Positionen dupliziert werden. Zusätzlich zum Objekt selbst können mehrere (keine, eine oder viele) Paare von Feldern übergeben werden. Ein Paar („pf“, „sf“) verbindet das Feld „pf“ des Parent-Objekts mit dem Feld „sf“ des Sub-Objekts. So ist es möglich, Werte des Parent-Objekts zu ändern und diese Änderungen für Sub-Objekte zu übernehmen. Üblicherweise muss zum Beispiel die Nummer des Hauptobjekts geändert werden. Die gleiche Änderung ist für alle abhängigen Objekte nötig, damit die neuen Objekte sich auf das Parent-Objekt beziehen.

Parameter

- **dObj** (DuplicateBusinessObject) – das Unter-Objekt
- **args** – Liste von Tupeln, die Parent- und Sub-Objekt-Felder miteinander verbinden

allowUpdate
Erlaube Updates statt Fehlern, wenn Objekt schon in DB existiert

dependentObjs: List[Dict[str, Any]]
Abhängige Objekte

fields
die Daten

fieldsNotCopied
Datenfelder, die im Original vorhanden sind, aber nicht kopiert werden sollen

getField(*field: str*, *onlyCopied: bool = False*) → Any
Schlägt den Wert eines Feldes nach. Wenn onlyCopied gesetzt ist, werden nur Felder zurückgeliefert, die auch kopiert werden sollen.

insert(*server:* PyAPplus64.applus.APplusServer) → *PyAPplus64.applus_db.DBTableIDs*
Fügt alle Objekte zur DB hinzu. Es wird die Menge der IDs der erzeugten Objekte gruppiert nach Tabellen erzeugt. Falls ein Datensatz schon existiert, wird dieser entweder aktualisiert oder eine Fehlermeldung geworfen. Geliefert wird die Menge aller Eingefügten Objekte mit ihrer ID.

setFields(*upds: Dict[str, Any]*) → None
Setzt Felder des DuplicateBusinessObjektes und falls nötig seiner Unterobjekte. So kann zum Beispiel die Nummer vor dem Speichern geändert werden.

Parameter **upds** – Dictionary mit zu setzenden Werten

table
für welche Tabelle ist das BusinessObject

class PyAPplus64.duplicate.**FieldsToCopyForTableCache**(*server:* PyAPplus64.applus.APplusServer)
Bases: object

Cache für welche Felder für welche Tabelle kopiert werden sollen

getFieldsToCopyForTable(*table: str*) → Set[str]
Bestimmt die für eine Tabelle zu kopierenden Spalten. Dazu wird in den XML-Definitionen geschaut. Ist dort ‚include‘ hinterlegt, werden diese Spalten verwendet. Ansonsten alle nicht generierten Spalten, ohne die ‚exclude‘ Spalten. In jedem Fall werden Spalten wie „ID“, die nie kopiert werden sollten, entfernt.

PyAPplus64.duplicate.**addSachgruppeDependentObjects**(*do:* PyAPplus64.duplicate.DuplicateBusinessObject, *server:* PyAPplus64.applus.APplusServer, *cache: Optional[*PyAPplus64.duplicate.FieldsToCopyForTableCache*]* = *None*) → None

Fügt Unterobjekte hinzu, die die Sachgruppenwerte kopieren.

Parameter

- **do** – zu erweiterndes DuplicateBusinessObject
- **server** (APplusServer) – Verbindung zum APP-Server, benutzt zum Nachschlagen der zu kopierenden Felder
- **cache** (*Optional[*FieldsToCopyForTableCache*]*) – Cache, so dass benötigte Felder nicht immer wieder neu berechnet werden müssen

PyAPplus64.duplicate.**getFieldsToCopyForTable**(*server:* PyAPplus64.applus.APplusServer, *table: str*, *force: bool* = *True*) → Set[str]

Bestimmt die für eine Tabelle zu kopierenden Spalten. Dazu wird in den XML-Definitionen geschaut. Ist dort ‚include' hinterlegt, werden diese Spalten verwendet. Ansonsten alle nicht generierten Spalten, ohne die ‚exclude' Spalten. In jedem Fall werden Spalten wie „ID", die nie kopiert werden sollten, entfernt.

PyAPplus64.duplicate.**initFieldsToCopyForTableCacheIfNeeded**(*server:* PyAPplus64.applus.APplusServer, *cache: Optional[*PyAPplus64.duplicate.FieldsToCopyForTableCac → *PyAPplus64.duplicate.FieldsToCopyForTableCache*

Hilfsfunktion, die einen Cache erzeugt, falls dies noch nicht geschehen ist.

PyAPplus64.duplicate.**loadDBDuplicateAPlan**(*server:* PyAPplus64.applus.APplusServer, *aplan: str*, *cache: Optional[*PyAPplus64.duplicate.FieldsToCopyForTableCache*]* = *None*) → Optional[*PyAPplus64.duplicate.DuplicateBusinessObject*]

Erstelle DuplicateBusinessObject für einzelnen Arbeitsplan.

Parameter

- **server** (APplusServer) – Verbindung zum APP-Server, benutzt zum Nachschlagen der zu kopierenden Felder
- **aplan** (*str*) – Aplan, der kopiert werden soll.
- **cache** (*Optional[*FieldsToCopyForTableCache*]*) – Cache, so dass benötigte Felder nicht immer wieder neu berechnet werden müssen

Rückgabe das neue DuplicateBusinessObject

Rückgabetyp *DuplicateBusinessObject*

PyAPplus64.duplicate.**loadDBDuplicateArtikel**(*server:* PyAPplus64.applus.APplusServer, *artikel: str*, *cache: Optional[*PyAPplus64.duplicate.FieldsToCopyForTableCache*]* = *None*, *dupAplan: bool* = *True*, *dupStueli: bool* = *True*) → Optional[*PyAPplus64.duplicate.DuplicateBusinessObject*]

Erstelle DuplicateBusinessObject für einzelnen Artikel.

Parameter

- **server** (APplusServer) – Verbindung zum APP-Server, benutzt zum Nachschlagen der zu kopierenden Felder
- **artikel** (*str*) – Artikel, der kopiert werden soll

- **cache** (*Optional[FieldsToCopyForTableCache]*) – Cache, so dass benötigte Felder nicht immer wieder neu berechnet werden müssen
- **dupAplan** (*bool optional*) – Arbeitsplan duplizieren?
- **dupStueli** (*bool optional*) – Stückliste duplizieren?

Rückgabe das neue DuplicateBusinessObject

Rückgabetyp *DuplicateBusinessObject*

PyAPplus64.duplicate.**loadDBDuplicateBusinessObject**(*server:* PyAPplus64.applus.APplusServer, *table: str*, *cond:* PyAPplus64.sql_utils.SqlCondition, *cache: Optional[*PyAPplus64.duplicate.FieldsToCopyForTableCache*]* = *None*, *allowUpdate: bool* = *False*) → Optional[*PyAPplus64.duplicate.DuplicateBusinessObject*]

Läd ein einzelnes DuplicateBusinessObjekt aus der DB. Die Bedingung sollte dabei einen eindeutigen Datensatz auswählen. Werden mehrere zurückgeliefert, wird ein zufälliger ausgewählt. Wird kein Datensatz gefunden, wird None geliefert.

Parameter

- **server** (APplusServer) – Verbindung zum APP-Server, benutzt zum Nachschlagen der zu kopierenden Felder
- **table** (*str*) – Tabelle für das neue DuplicateBusinessObjekt
- **cond** (sql_utils.SqlCondition) – SQL-Bedingung zur Auswahl eines Objektes
- **cache** (*Optional[FieldsToCopyForTableCache]*) – Cache, so dass benötigte Felder nicht immer wieder neu berechnet werden müssen
- **allowUpdate** (*bool*) – ist Update statt Insert erlaubt?

Rückgabe das neue DuplicateBusinessObject

Rückgabetyp Optional[*DuplicateBusinessObject*]

PyAPplus64.duplicate.**loadDBDuplicateBusinessObjectSimpleCond**(*server:* PyAPplus64.applus.APplusServer, *table: str*, *field: str*, *value: Optional[Union[str, int, float,* PyAPplus64.sql_utils.SqlParam, PyAPplus64.sql_utils.SqlField, PyAPplus64.sql_utils.SqlFixed, PyAPplus64.sql_utils.SqlDate, PyAPplus64.sql_utils.SqlDateTime, *datetime.datetime, datetime.date, datetime.time, bool]]*, *cache: Optional[*PyAPplus64.duplicate.FieldsToCopyForTableC = *None*, *allowUpdate: bool* = *False*) → Optional[*PyAPplus64.duplicate.DuplicateBusinessObjec*

Wrapper für loadDBDuplicateBusinessObject, das eine einfache Bedingung benutzt, bei der ein Feld einen bestimmten Wert haben muss.

Parameter

- **server** (APplusServer) – Verbindung zum APP-Server, benutzt zum Nachschlagen der zu kopierenden Felder
- **table** (*str*) – Tabelle für das neue DuplicateBusinessObjekt

- **field** (*str*) – Feld für Bedingung
- **value** – Wert des Feldes für Bedingung
- **cache** (*Optional[FieldsToCopyForTableCache]*) – Cache, so dass benötigte Felder nicht immer wieder neu berechnet werden müssen

Rückgabe das neue DuplicateBusinessObject

Rückgabetyp Optional[*DuplicateBusinessObject*]

PyAPplus64.duplicate.**loadDBDuplicateBusinessObjects**(*server:* PyAPplus64.applus.APplusServer, *table: str*, *cond:* PyAPplus64.sql_utils.SqlCondition, *cache: Optional[*PyAPplus64.duplicate.FieldsToCopyForTableCache*] = None*, *allowUpdate: bool = False*) → Sequence[*PyAPplus64.duplicate.DuplicateBusinessObject*]

Läd eine Liste von DuplicateBusinessObjekten aus der DB. Die Bedingung kann mehrere Datensätze auswählen.

Parameter

- **server** (APplusServer) – Verbindung zum APP-Server, benutzt zum Nachschlagen der zu kopierenden Felder
- **table** (*str*) – Tabelle für das neue DuplicateBusinessObjekt
- **cond** (sql_utils.SqlCondition) – SQL-Bedingung zur Auswahl eines Objektes
- **cache** (*Optional[FieldsToCopyForTableCache]*) – Cache, so dass benötigte Felder nicht immer wieder neu berechnet werden müssen

Rückgabe Liste der neuen DuplicateBusinessObjects

Rückgabetyp Sequence[*DuplicateBusinessObject*]

PyAPplus64.duplicate.**loadDBDuplicateBusinessObjectsSimpleCond**(*server:* PyAPplus64.applus.APplusServer, *table: str*, *field: str*, *value: Optional[Union[str, int, float,* PyAPplus64.sql_utils.SqlParam*,* PyAPplus64.sql_utils.SqlField*,* PyAPplus64.sql_utils.SqlFixed*,* PyAPplus64.sql_utils.SqlDate*,* PyAPplus64.sql_utils.SqlDateTime*, datetime.datetime, datetime.date, datetime.time, bool]]*, *cache: Optional[*PyAPplus64.duplicate.FieldsToCopyForTable*] = None*, *allowUpdate: bool = False*) → Sequence[*PyAPplus64.duplicate.DuplicateBusiness*

Wrapper für loadDBDuplicateBusinessObjects, das eine einfache Bedingung benutzt, bei der ein Feld einen bestimmten Wert haben muss.

Parameter

- **server** (APplusServer) – Verbindung zum APP-Server, benutzt zum Nachschlagen der zu kopierenden Felder
- **table** (*str*) – Tabelle für das neue DuplicateBusinessObjekt
- **field** – Feld für Bedingung

- **value** – Wert des Feldes für Bedingung
- **cache** (*Optional[FieldsToCopyForTableCache]*) – Cache, so dass benötigte Felder nicht immer wieder neu berechnet werden müssen

Rückgabe Liste der neuen DuplicateBusinessObjects

Rückgabetyp Sequence[*DuplicateBusinessObject*]

PyAPplus64.duplicate.**loadDBDuplicateStueli**(*server: PyAPplus64.applus.APplusServer*, *stueli: str*, *cache: Optional[PyAPplus64.duplicate.FieldsToCopyForTableCache] = None*) → Optional[*PyAPplus64.duplicate.DuplicateBusinessObject*]

Erstelle DuplicateBusinessObject für einzelne Stückliste.

Parameter

- **server** (APplusServer) – Verbindung zum APP-Server, benutzt zum Nachschlagen der zu kopierenden Felder
- **stueli** (*str*) – Stückliste, die kopiert werden soll.
- **cache** (*Optional[FieldsToCopyForTableCache]*) – Cache, so dass benötigte Felder nicht immer wieder neu berechnet werden müssen

Rückgabe das neue DuplicateBusinessObject

Rückgabetyp Optional[*DuplicateBusinessObject*]

PyAPplus64.duplicate.**noCopyFields = {'GUID', 'ID', 'ID_A', 'INSDATE', 'INSUSER', 'MANDANT', 'TIMESTAMP', 'TIMESTAMP_A', 'UPDDATE', 'UPDUSER'}**

Menge von Feld-Namen, die nie kopiert werden sollen.

5.10 PyAPplus64.pandas module

Pandas Interface für PyAPplus64.

PyAPplus64.pandas.**createSqlAlchemyEngine**(*server: PyAPplus64.applus.APplusServer*) → sqlalchemy.engine.base.Engine

Erzeugt eine SqlAlchemy-Engine für die Verbindung zur DB.

PyAPplus64.pandas.**exportToExcel**(*filename: Union[str, PathLike[str], pandas._typing.WriteExcelBuffer, pandas.io.excel._base.ExcelWriter]*, *dfs: Sequence[Tuple[pandas.core.frame.DataFrame, str]]*, *addTable: bool = True*) → None

Schreibt eine Menge von Dataframes in eine Excel-Tabelle

Parameter

- **filename** – Name der Excel-Datei
- **dfs** – Liste von Tupeln aus DataFrames und Namen von Sheets.

PyAPplus64.pandas.**mkDataframeColumn**(*df: pandas.core.frame.DataFrame*, *makeValue: Union[Callable, str, List[Union[Callable, str]], Dict[Hashable, Union[Callable, str, List[Union[Callable, str]]]]]*) → pandas.core.series.Series

Erzeugt für alle Zeilen eines Dataframes eine neuen Wert. Dies wird benutzt, um eine Spalte zu berechnen. Diese kann eine Originalspalte ersetzen, oder neu hinzugefügt werden.

Parameter

- **df** – der Dataframe
- **makeValue** – Funktion, die eine Zeile als Parameter bekommt und den neuen Wert berechnet

PyAPplus64.pandas.**mkHyperlinkDataframeColumn**(*df: pandas.core.frame.DataFrame*, *makeOrig: Union[Callable, str, List[Union[Callable, str]], Dict[Hashable, Union[Callable, str, List[Union[Callable, str]]]]]*, *makeLink: Callable[[Any], str]*) → pandas.core.series.Series

Erzeugt für alle Zeilen eines Dataframes einen Hyperlink. Dies wird benutzt, um eine Spalte mit einem Hyperlink zu berechnen. Diese kann eine Originalspalte ersetzen, oder neu hinzugefügt werden.

Parameter

- **df** – der Dataframe
- **makeOrig** – Funktion, die eine Zeile als Parameter bekommt und den Wert berechnet, der angezeigt werden soll
- **makeLink** – Funktion, die eine Zeile als Parameter bekommt und den Link berechnet

PyAPplus64.pandas.**pandasReadSql**(*server:* PyAPplus64.applus.APplusServer, *sql: Union[*PyAPplus64.sql_utils.SqlStatementSelect*, str]*, *raw: bool = False*, *engine: Optional[sqlalchemy.engine.base.Engine] = None*) → pandas.core.frame.DataFrame

Wrapper für pd.read_sql für sqlalchemy-engine.

Parameter

- **server** (*APplusServer*) – APplusServer für Datenbankverbindung und complete-SQL
- **sql** – das SQL-statement

5.11 PyAPplus64.sql_utils module

Diese Datei enthält Funktionen für den Bau von SQL Statements, besonders SELECT-Statements. Es gibt viel ausgefeiltere Methoden für die Erstellung von SQL Statements. APplus benötigt jedoch die Statements als Strings, die dann an APplus für Änderungen und erst danach an die DB geschickt werden. Dies erschwert die Nutzung von Tools wie SqlAlchemy.

Hier werden einfache Hilfsfunktionen, die auf Strings basieren, zur Verfügung gestellt. PyODBC erlaubt Parameter (dargestellt als ‚¿) in SQL Statements, die dann beim Aufruf gefüllt werden. Dies funktioniert auch im Zusammenspiel mit APplus. Oft ist es sinnvoll, solche Parameter zu verwenden.

class PyAPplus64.sql_utils.**SqlCondition**

Bases: object

Eine abstrakte Sql-Bedingung. Unterklassen erledigen die eigentliche Arbeit.

getCondition() → str

Liefert die Bedingung als String

Rückgabe die Bedingung

Rückgabetyp str

class PyAPplus64.sql_utils.**SqlConditionAnd**(**conds: Union[*PyAPplus64.sql_utils.SqlCondition*, str]*)

Bases: *PyAPplus64.sql_utils.SqlConditionList*

class PyAPplus64.sql_utils.**SqlConditionBinComp**(*op: str*, *value1:* SqlValue, *value2:* SqlValue)

Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

Bedingung der Form ‚value1 op value2‘

Parameter

- **op** (*str*) – der Vergleichsoperator
- **value1** (*SqlValue*) – der Wert, kann unterschiedliche Typen besitzen

- **value2** (*SqlValue*) – der Wert, kann unterschiedliche Typen besitzen

class PyAPplus64.sql_utils.**SqlConditionBool**(*b: bool*)
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

Fixe True-oder-False Bedingung

class PyAPplus64.sql_utils.**SqlConditionDateTimeFieldInDay**(*field: str*, *year: int*, *month: int*, *day: int*)
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

Liegt Datetime in einem bestimmten Monat?

Parameter

- **field** (*str*) – das Feld
- **year** – das Jahr
- **month** – der Monat
- **day** – der Tag

class PyAPplus64.sql_utils.**SqlConditionDateTimeFieldInMonth**(*field: str*, *year: int*, *month: int*)
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

Liegt Datetime in einem bestimmten Monat?

Parameter

- **field** (*string*) – das Feld
- **year** – das Jahr
- **month** – der Monat

class PyAPplus64.sql_utils.**SqlConditionDateTimeFieldInRange**(*field: str*, *datetimeVon: Optional[datetime.datetime]*, *datetimeBis: Optional[datetime.datetime]*)
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

Liegt Datetime in einem bestimmten Zeitraum?

Parameter

- **field** (*str*) – das Feld
- **datetimeVon** – der untere Wert (einschließlich), None erlaubt beliebige Zeiten
- **datetimeBis** – der obere Wert (ausschließlich), None erlaubt beliebige Zeiten

class PyAPplus64.sql_utils.**SqlConditionDateTimeFieldInYear**(*field: str*, *year: int*)
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

Liegt Datetime in einem bestimmten Jahr?

Parameter

- **field** (*str*) – das Feld
- **year** – das Jahr

class PyAPplus64.sql_utils.**SqlConditionEq**(*value1: Union[SqlValue, bool, None]*, *value2: Union[SqlValue, bool, None]*)
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

Bedingung der Form ‚v1 is null‘, ‚v2 is null‘, ‚v1 = v2‘, ‚(1=1)‘ oder ‚(0=1)‘

Parameter

- **value1** – der Wert, kann unterschiedliche Typen besitzen
- **value2** – der Wert, kann unterschiedliche Typen besitzen

class PyAPplus64.sql_utils.**SqlConditionFalse**
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

False-Bedingung

class PyAPplus64.sql_utils.**SqlConditionFieldEq**(*field: str*, *value: Union[*SqlValue*, bool, None]*)
Bases: *PyAPplus64.sql_utils.SqlConditionEq*

class PyAPplus64.sql_utils.**SqlConditionFieldGe**(*field: str*, *value:* SqlValue)
Bases: *PyAPplus64.sql_utils.SqlConditionGe*

class PyAPplus64.sql_utils.**SqlConditionFieldGt**(*field: str*, *value:* SqlValue)
Bases: *PyAPplus64.sql_utils.SqlConditionGt*

class PyAPplus64.sql_utils.**SqlConditionFieldIn**
Bases: *PyAPplus64.sql_utils.SqlConditionIn*

class PyAPplus64.sql_utils.**SqlConditionFieldIsNotNull**(*field: str*)
Bases: *PyAPplus64.sql_utils.SqlConditionIsNotNull*

class PyAPplus64.sql_utils.**SqlConditionFieldIsNull**(*field: str*)
Bases: *PyAPplus64.sql_utils.SqlConditionIsNull*

class PyAPplus64.sql_utils.**SqlConditionFieldLe**(*field: str*, *value:* SqlValue)
Bases: *PyAPplus64.sql_utils.SqlConditionLe*

class PyAPplus64.sql_utils.**SqlConditionFieldLt**(*field: str*, *value:* SqlValue)
Bases: *PyAPplus64.sql_utils.SqlConditionLt*

class PyAPplus64.sql_utils.**SqlConditionFieldStringNotEmpty**(*field: str*)
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

Feld soll nicht den leeren String oder null enthalten. Der Ausdruck wird wörtlich übernommen.

Parameter **field** (*str*) – das Feld

class PyAPplus64.sql_utils.**SqlConditionGe**(*value1:* SqlValue, *value2:* SqlValue)
Bases: *PyAPplus64.sql_utils.SqlConditionBinComp*

Bedingung der Form ‚value1 >= value2‘

Parameter

- **value1** – der Wert, kann unterschiedliche Typen besitzen
- **value2** – der Wert, kann unterschiedliche Typen besitzen

class PyAPplus64.sql_utils.**SqlConditionGt**(*value1:* SqlValue, *value2:* SqlValue)
Bases: *PyAPplus64.sql_utils.SqlConditionBinComp*

Bedingung der Form ‚value1 > value2‘

Parameter

- **value1** – der Wert, kann unterschiedliche Typen besitzen
- **value2** – der Wert, kann unterschiedliche Typen besitzen

class PyAPplus64.sql_utils.**SqlConditionIn**
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

Bedingung der Form ‚v in …‘

Parameter

- **value** (*SqlValue*) – der Wert, kann unterschiedliche Typen besitzen
- **values** (*Sequence[SqlValue]*) – die erlaubten Werte

class PyAPplus64.sql_utils.**SqlConditionIsNotNull**(*v:* SqlValue)
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

Wert soll nicht null sein

Parameter **v** (*SqlValue*) – der Wert

class PyAPplus64.sql_utils.**SqlConditionIsNull**(*v:* SqlValue)
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

Wert soll null sein

Parameter **v** (*SqlValue*) – das Feld

class PyAPplus64.sql_utils.**SqlConditionLe**(*value1:* SqlValue, *value2:* SqlValue)
Bases: *PyAPplus64.sql_utils.SqlConditionBinComp*

Bedingung der Form ‚value1 <= value2‘

Parameter

- **value1** – der Wert, kann unterschiedliche Typen besitzen
- **value2** – der Wert, kann unterschiedliche Typen besitzen

class PyAPplus64.sql_utils.**SqlConditionList**(*connector: str*, *emptyCond: str*)
Bases: *PyAPplus64.sql_utils.SqlCondition*

Eine SQL Bedingung, die sich aus einer Liste anderer Bedingungen zusammensetzen. Dies kann eine „AND“ oder eine „OR“ Liste sein.

Parameter

- **connector** (*str*) – wie werden Listenelemente verbunden (AND oder OR)
- **emptyCond** (*str*) – Rückgabewert für leere Liste

addCondition(*cond: Optional[Union[*PyAPplus64.sql_utils.SqlCondition*, str]]*) → None

addConditionEq(*value1: Union[*SqlValue*, bool, None]*, *value2: Union[*SqlValue*, bool, None]*) → None

addConditionFieldEq(*field: str*, *value: Union[*SqlValue*, bool, None]*) → None

addConditionFieldGe(*field: str*, *value:* SqlValue) → None

addConditionFieldGt(*field: str*, *value:* SqlValue) → None

addConditionFieldIn()

addConditionFieldIsNotNull(*field: str*) → None

addConditionFieldIsNull(*field: str*) → None

addConditionFieldLe(*field: str*, *value:* SqlValue) → None

addConditionFieldLt(*field: str*, *value:* SqlValue) → None

addConditionFieldStringNotEmpty(*field: str*) → None

addConditionFieldsEq(*field1: str*, *field2: str*) → None

addConditionFieldsGe(*field1: str*, *field2: str*) → None

addConditionFieldsGt(*field1: str*, *field2: str*) → None

addConditionFieldsLe(*field1: str*, *field2: str*) → None

addConditionFieldsLt(*field1: str*, *field2: str*) → None

addConditionGe(*value1:* SqlValue, *value2:* SqlValue) → None

addConditionGt(*value1:* SqlValue, *value2:* SqlValue) → None

addConditionLe(*value1:* SqlValue, *value2:* SqlValue) → None

addConditionLt(*value1:* SqlValue, *value2:* SqlValue) → None

addConditions(**conds: Optional[Union[*PyAPplus64.sql_utils.SqlCondition*, str]]*) → None

getCondition() → str
Liefert die Bedingung als String

Rückgabe die Bedingung

Rückgabetyp str

isEmpty() → bool

class PyAPplus64.sql_utils.**SqlConditionLt**(*value1:* SqlValue, *value2:* SqlValue)
Bases: *PyAPplus64.sql_utils.SqlConditionBinComp*

Bedingung der Form ‚value1 < value2‘

Parameter

- **value1** – der Wert, kann unterschiedliche Typen besitzen
- **value2** – der Wert, kann unterschiedliche Typen besitzen

class PyAPplus64.sql_utils.**SqlConditionNot**(*cond:* PyAPplus64.sql_utils.SqlCondition)
Bases: *PyAPplus64.sql_utils.SqlCondition*

Negation einer anderen Bedingung

Parameter **cond** (SqlCondition) – die zu negierende Bedingung

getCondition() → str
Liefert die Bedingung als String

Rückgabe die Bedingung

Rückgabetyp str

class PyAPplus64.sql_utils.**SqlConditionOr**(**conds: Union[*PyAPplus64.sql_utils.SqlCondition*, str]*)
Bases: *PyAPplus64.sql_utils.SqlConditionList*

class PyAPplus64.sql_utils.**SqlConditionPrepared**(*cond: Union[*PyAPplus64.sql_utils.SqlCondition*, str]*)
Bases: *PyAPplus64.sql_utils.SqlCondition*

Eine einfache Sql-Bedingung, die immer einen festen String zurückgibt.

getCondition() → str
Liefert die Bedingung als String

Rückgabe die Bedingung

Rückgabetyp str

class PyAPplus64.sql_utils.**SqlConditionStringStartsWith**(*field: str*, *value: str*)
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

Feld soll mit einem bestimmten String beginnen

Parameter

- **field** (*str*) – das Feld
- **value** (*str*) – der Wert

class PyAPplus64.sql_utils.**SqlConditionTrue**
Bases: *PyAPplus64.sql_utils.SqlConditionPrepared*

True-Bedingung

class PyAPplus64.sql_utils.**SqlDate**(*d: Union[datetime.datetime, datetime.date] = datetime.datetime(2023, 11, 13, 12, 52, 48, 792832)*)
Bases: object

Wrapper um DateTime, die die Formatierung erleichtern

Parameter **d** (*Union[datetime.datetime, datetime.date]*) – das Datum

class PyAPplus64.sql_utils.**SqlDateTime**(*dt: Union[datetime.datetime, datetime.date] = datetime.datetime(2023, 11, 13, 12, 52, 48, 792824)*)

Bases: object

Wrapper um DateTime, die die Formatierung erleichtern

Parameter **dt** (*Union[datetime.datetime, datetime.date]*) – der Zeitpunkt

class PyAPplus64.sql_utils.**SqlField**(*fn: str*)

Bases: object

Wrapper um SQL Feldnamen, die die Formatierung erleichtern

Parameter **fn** (*str*) – der Feldname

class PyAPplus64.sql_utils.**SqlFixed**(*s: str*)

Bases: object

Wrapper um Strings, die ohne Änderung in SQL übernommen werden

Parameter **s** (*str*) – der string

class PyAPplus64.sql_utils.**SqlInnerJoin**(*table: str*, **conds: Union[*PyAPplus64.sql_utils.SqlCondition*, str]*)

Bases: *PyAPplus64.sql_utils.SqlJoin*

Ein Inner-Join.

Parameter

- **table** (*str*) – die Tabelle, die gejoint werden soll
- **conds** – Bedingungen, die bereits hinzugefügt werden soll. Weitere können über Attribut *on* hinzugefügt werden.

class PyAPplus64.sql_utils.**SqlJoin**(*joinType: str*, *table: str*, **conds: Union[*PyAPplus64.sql_utils.SqlCondition*, str]*)

Bases: object

Ein abstrakter Sql-Join

Parameter

- **joinType** (*str*) – Art des Joins, wird in SQL übernommen, z.B. „LEFT JOIN“.
- **table** (*str*) – die Tabelle, die gejoint werden soll
- **conds** – Bedingungen, die bereits hinzugefügt werden soll. Weitere können über Attribut *on* hinzugefügt werden.

getJoin() → str

Liefert den Join als String

on: *PyAPplus64.sql_utils.SqlConditionAnd*

Bedingung des Joins, kann noch nachträglich erweitert werden

class PyAPplus64.sql_utils.**SqlLeftJoin**(*table: str*, **conds: Union[*PyAPplus64.sql_utils.SqlCondition*, str]*)

Bases: *PyAPplus64.sql_utils.SqlJoin*

Ein Left-Join.

Parameter

- **table** (*str*) – die Tabelle, die gejoint werden soll
- **conds** – Bedingungen, die bereits hinzugefügt werden soll. Weitere können über Attribut *on* hinzugefügt werden.

class PyAPplus64.sql_utils.**SqlParam**
Bases: object

Hilfsklasse, für einen Parameter (?)

class PyAPplus64.sql_utils.**SqlStatementSelect**(*table: str*, **fields: str*)
Bases: object

Klasse, um einfache Select-Statements zu bauen.

Parameter

- **table** (*str*) – die Haupt-Tabelle
- **fields** – kein oder mehrere Felder, die selektiert werden sollen

addFields(**fields: str*) → None
Fügt ein oder mehrere Felder, also auszuwählende Werte zu einem SQL-Statement hinzu.

addFieldsTable(*table: str*, **fields: str*) → None
Fügt ein oder mehrere Felder, die zu einer Tabelle gehören zu einem SQL-Statement hinzu. Felder sind Strings. Vor jeden dieser Strings wird die Tabelle mit einem Punkt getrennt gesetzt. Dies kann im Vergleich zu ‚addFields' Schreibarbeit erleitern.

addGroupBy(**fields: str*) → None
Fügt ein oder mehrere GroupBy Felder zu einem SQL-Statement hinzu.

addInnerJoin(*table: str*, **conds: Union[*PyAPplus64.sql_utils.SqlCondition*, str]*) → *PyAPplus64.sql_utils.SqlInnerJoin*

addJoin(*j: Union[*PyAPplus64.sql_utils.SqlJoin*, str]*) → None
Fügt ein Join zum SQL-Statement hinzu. Beispiel: ‚LEFT JOIN personal p ON t.UPDUSER = p.PERSONAL'

addLeftJoin(*table: str*, **conds: Union[*PyAPplus64.sql_utils.SqlCondition*, str]*) → *PyAPplus64.sql_utils.SqlLeftJoin*

fields: List[str]
Liste von auszuwählenden Feldern

getSql() → str
Liefert das SQL-SELECT-Statement als String

groupBy: List[str]
die Bedingung, Default ist True

having: *PyAPplus64.sql_utils.SqlConditionList*
die Bedingung having, Default ist True

joins: List[Union[*PyAPplus64.sql_utils.SqlJoin*, str]]
Joins mit extra Tabellen

order: Optional[str]
Sortierung

setTop(*t: int*) → None
Wie viele Datensätze sollen maximal zurückgeliefert werden? 0 für alle

table: str
die Tabelle

top: int
wie viele Datensätze auswählen? 0 für alle

where: *PyAPplus64.sql_utils.SqlConditionList*
die Bedingung, Default ist True

class PyAPplus64.sql_utils.**SqlTime**(*t: Union[datetime.datetime, datetime.time] = datetime.datetime(2023, 11, 13, 12, 52, 48, 792836)*)
Bases: object

Wrapper um DateTime, die die Formatierung erleichtern

Parameter **t** (*Union[datetime.datetime, datetime.time]*) – die Zeit

PyAPplus64.sql_utils.**SqlValue**
Union-Type aller unterstützter SQL-Werte

alias of Union[str, int, float, *PyAPplus64.sql_utils.SqlParam*, *PyAPplus64.sql_utils.SqlField*, *PyAPplus64.sql_utils.SqlFixed*, *PyAPplus64.sql_utils.SqlDate*, *PyAPplus64.sql_utils.SqlDateTime*, datetime.datetime, datetime.date, datetime.time]

PyAPplus64.sql_utils.**formatSqlValue**(*v:* SqlValue) → str
Formatiert einen Wert für SQL. Je nachdem um welchen Typ es sich handelt, werden andere Formatierungen verwendet.

Parameter **v** (*SqlValue*) – der Wert

Rückgabe der formatierte Wert

Rückgabetyp str

PyAPplus64.sql_utils.**formatSqlValueString**(*s: str*) → str
Formatiert einen String für ein Sql-Statement. Der String wird in „“ eingeschlossen und Hochkomma im Text maskiert.

Parameter **s** (*str*) – der String

Rückgabe der formatierte String

Rückgabetyp str

PyAPplus64.sql_utils.**normaliseDBfield**(*f: str*) → str
Normalisiert die Darstellung eines DB-Feldes

PyAPplus64.sql_utils.**normaliseDBfieldList**(*fields: Sequence[str]*) → Sequence[str]
Normalisiert eine Menge von DB-Feldern

PyAPplus64.sql_utils.**normaliseDBfieldSet**(*s: Set[str]*) → Set[str]
Normalisiert eine Menge von DB-Feldern

PyAPplus64.sql_utils.**sqlParam = <PyAPplus64.sql_utils.SqlParam object>**
Da SqlParam keinen Zustand hat, reicht ein einzelner statischer Wert

5.12 PyAPplus64.utils module

PyAPplus64.utils.**checkDirExists**(*dir: Union[str, pathlib.Path]*) → pathlib.Path
Prüft, ob ein Verzeichnis existiert. Ist dies nicht möglich, wird eine Exception geworfen.

Parameter **dir** (*Union[str, pathlib.Path]*) – das Verzeichnis

Rückgabe den normalisierten Pfad

Rückgabetyp pathlib.Path

PyAPplus64.utils.**containsOnlyAllowedChars**(*charset: Set[str]*, *s: str*) → bool
Enthält ein String nur erlaubte Zeichen?

PyAPplus64.utils.**formatDateTimeForAPplus**(*v: Union[datetime.datetime, datetime.date, datetime.time]*) → str
Formatiert ein Datum oder eine Uhrzeit für APplus

5.13 Module contents

Python-Modulindex

p