

Univention Management Console (UMC)

Thema:	Dokumentation für Entwickler	
Datum:	15. Dezember 2009	
Seitenzahl:	27	
Versionsnummer:	4532	
Autoren:	Univention GmbH	feedback@univention.de

Inhaltsverzeichnis

1	Einleitung	3
2	Architektur	3
3	Asynchrones Framework	4
4	Protokoll - UMCP	5
4.1	Ablauf	5
4.2	Authentisierung	5
4.3	Nachrichtenformat	6
4.4	Python API	7
5	Berechtigungsmodell	10
6	Module	10
6.1	Aufbau	11
6.2	Tools	15
7	Dialoge	16
7.1	Elemente	17
7.2	Interaktion	20
8	Häufig verwendete Elemente	23
8.1	Information	23
8.2	Fragen	23
8.3	Suchdialog	24
9	FAQ	25
A	Status-Codes	27
B	Layout-Attribute	27

1 Einleitung

Die Univention Management Console ist ein Dienst, der auf jedem UCS-System gestartet werden kann, über den Informationen abgefragt und auch Anpassungen durchgeführt werden können. Welche Möglichkeiten im Detail dieser Dienst auf den einzelnen Systemen bieten kann hängt von den installierten UMC-Modulen ab.

Mit UMC ist es möglich gleichzeitig auf mehreren Systemen Informationen abzufragen oder sogar Befehle auszuführen. Dafür muss sich der Benutzer nur mit einem System der UCS-Domäne verbinden, beispielsweise über das Web-Interface und kann von dort aus auch andere Systeme administrieren.

Im folgenden werden die technischen Details von UMC genauer beschrieben. Dabei wird der Schwerpunkt auf die Teile gelegt, die für die Entwicklung eigener UMC-Module wichtig sind.

Mit dem Wechsel auf UCS 2.0 haben sich im allgemeinen Wording einige Bezeichnungen geändert. So heißt zum Beispiel Univention Baseconfig jetzt Univention Configuration Registry und analog dazu wurde unter anderem univention-baseconfig in univention-config-registry umbenannt. Um die Kompatibilität mit alten Skripten zu gewährleisten sind die Namen aus UCS 1.3 noch gültig, es wird jedoch empfohlen in neuen Skripten die aktuellen Namen zu verwenden.

2 Architektur

Die Univention Management Console besteht aus mehreren Komponenten, die sich entweder über eine verschlüsselte Verbindung über das Netz oder lokale Interprozesskommunikationsmechanismen verständigen, wie in Abbildung 1. Im folgenden eine Liste der wichtigsten Komponenten:

Frontend Dämon wird für jede Sitzung über das Web-Frontend von UMC gestartet. Aufgabe ist die Kommunikation mit dem PHP-Framework und dem UMC-Server bzw. Proxy

Server ist der Dienst, der den Zugriff auf die UMC Befehle für den lokalen Rechner kontrolliert und diese gegebenenfalls ausführt.

Proxy übernimmt das Sitzungsmanagement für UMC-Befehle, die an mehrere Rechner gleichzeitig sendet werden sollen. Dafür spricht der UMC-Proxy mit den UMC-Servern auf den jeweiligen Systemen, schickt die Anfragen dahin weiter, sammelt die Antworten ein und generiert eine Zusammenfassung, die dann als Antwort an den Client zurückgeschickt wird. Diese Komponente ist momentan noch nicht entwickelt.

Modulprozess ist für die Ausführung von Befehlen eines UMC-Moduls auf einem lokalen System zuständig. Für jede Sitzung eines Benutzers wird pro verwendetem Modul ein Prozess gestartet. Diese beenden sich automatisch, wenn sie für eine definierte Zeitspanne nicht verwendet worden sind.

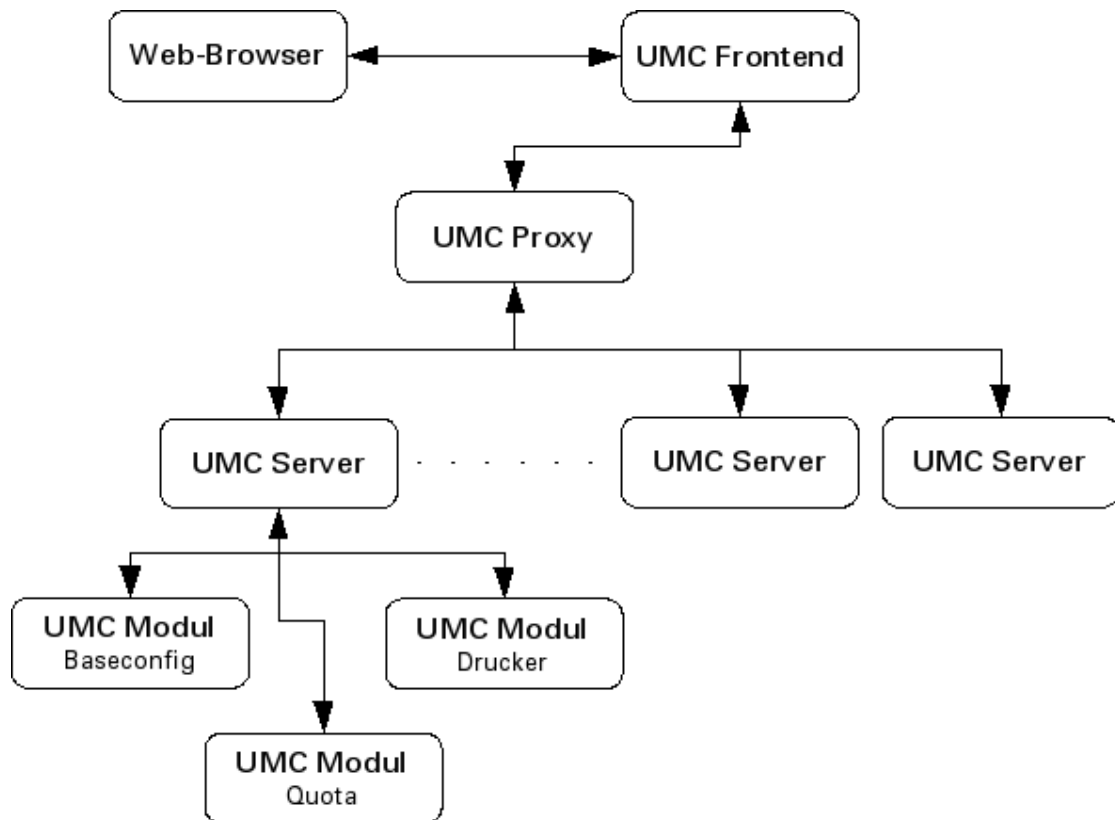


Abbildung 1: Komponenten von UMC

3 Asynchrones Framework

Sämtliche Komponenten von UMC basieren auf einem asynchronen Framework, der Techniken zur quasi-parallelen Abwicklung von mehreren Abläufen innerhalb eines Prozesses bereitstellt. Dabei gibt es drei wichtige Konzepte, die von dem Framework 'pynotifier' bereitgestellt bzw. genutzt werden:

Non-blocking Sockets Um mehrere Kommunikationskanäle über das Netz parallel betreiben zu können dürfen die Lese- und Schreiboperationen nicht blockieren. Daher muss bei dem Einsatz von Sockets immer das Flag für die nicht blockierenden Sockets gesetzt werden. Das Framework bietet die Möglichkeiten um auf Ereignisse an einer Socket zu reagieren.

Timer Um die Möglichkeit zu haben Aktionen nach Ablauf eines bestimmten Zeitintervalls durchzuführen bietet das Framework Methoden um Funktionen zu registrieren, die nach einer festgesetzten Zeit ausgeführt werden. Anhand des Rückgabewertes wird entschieden, ob dies erneut geschehen soll.

Events Mit diesem Mechanismus bietet das Framework die Möglichkeit Ereignisse bekannt zu geben, wodurch dann registrierte Funktionen angesprungen werden kön-

nen. Beispielsweise kann die Hauptklasse eines UMC-Servers das Ereignis 'receive' definieren und auslösen sobald eine gültige UMCP-Nachricht empfangen wurde. Beliebige andere Objekte können eine Registrierung für dieses Ereignis setzen, so dass eine angegebene Funktion angesprochen wird, wenn dieses Ereignis eintritt. Dadurch können mehrere unabhängige Komponenten in einem Prozess über dieses Ereignis informiert werden und ihr Verhalten dementsprechend anpassen.

Ein wichtiges Set von Tools im Zusammenhang mit UMC, das das Framework zur Verfügung stellt, hilft bei dem Starten von Kindprozessen oder Threads ohne den aktuellen Prozess zu blockieren. Dabei werden Mechanismen zur Verfügung gestellt mit denen der laufende Prozess über den Status der Kindprozesse und Threads informiert werden kann.

Beispiele und weitere Informationen dazu können auf der Webseite des Projektes nachgelesen werden (<http://www.bitkipper.net/Package/pynotifier>).

4 Protokoll - UMCP

Die Kommunikation zwischen den UMC-Komponenten wird über ein eigenes Protokoll abgewickelt, das in dem folgenden Abschnitt genauer beschrieben wird.

4.1 Ablauf

Das Protokoll basiert auf einem Server/Client-Modell. Dabei werden Anfragen vom Client (Request) an den Server geschickt. Dieser sollte daraufhin mit einer Antwort-Nachricht (Response) reagieren.

Anhand eines Status-Code in der Response-Nachricht kann der Client eine von drei Situation unterscheiden:

- Es ist ein Fehler bei der Bearbeitung aufgetreten. Dabei kann anhand des Status-Code auch eine Kategorie des Fehlers festgestellt werden.
- Der Befehl wurde erfolgreich ausgeführt
- Die Ausführung des Befehls ist noch nicht abgeschlossen. Diese Meldungen können vom Server geschickt werden um den Client zu informieren, dass der Befehl weiterhin bearbeitet wird und es noch zu keinen Fehlern gekommen ist. Optional kann solch eine Nachricht auch schon einen Teil der Antwort enthalten.

4.2 Authentisierung

Bevor ein Client UMC-Befehle ausführen kann muss er sich gegenüber dem Server authentisieren. Ist dies erfolgreich wird anhand von Berechtigungen, die im LDAP-Verzeichnis hinterlegt sind, geprüft, ob der angemeldete Benutzer diesen Befehl ausführen darf. Ist auch diese Prüfung positiv ausgefallen wird der Befehl vom Server verarbeitet.

Die Authentisierung gegen den UMC-Server wird mittels PAM durchgeführt. Dabei wird der PAM-Service **univention-management-console** verwendet, der einen Passwort-Cache verwendet, damit sich Benutzer auch anmelden können, wenn der LDAP-Server gerade nicht erreichbar ist. Für die Berechtigungsinformationen eines Benutzers gibt es ebenfalls einen lokalen Cache auf den im Notfall zurückgegriffen werden kann.

4.3 Nachrichtenformat

UMCP-Nachrichten bestehen aus einem Kopf und dem Rumpf. Im Kopf werden verschiedene Steuerungsinformationen untergebracht. Anhand dieser Informationen kann ohne Analyse des Rumpfs die syntaktische Korrektheit der Nachricht, der Nachrichtentyp, die korrekte Länge, das UMCP-Kommando sowie die Zuordnung zu einem UMC-Modul getroffen werden.

4.3.1 Nachrichtenkopf

Der Nachrichtenkopf besteht aus einer Zeile und ist wie folgt aufgebaut:

```
(REQUEST|RESPONSE)/<ID>/<LENGTH>: <UMCP command> <arguments>
```

Durch das erste Schlüsselwort wird der Nachrichtentyp definiert. Mit einem / vom nächsten Feld getrennt folgt dann der eindeutige Bezeichner der Nachricht. Dieser muss nur innerhalb einer Sitzung eindeutig sein. Um den Zusammenhang zwischen einer Request-Nachricht und den zugehörigen Response-Nachrichten zu erkennen haben alle Response-Nachrichten die selbe Kennung wie die Request-Nachricht.

Nach einem weiteren / folgt die Länge des Nachrichtenrumpfes in Bytes. Diese wird zuerst überprüft bevor die Nachricht weiter analysiert wird. Durch einen : und ein Leerzeichen getrennt folgt dann das UMCP-Kommando. Dies ist nicht mit einem UMC-Befehl zu verwechseln. Diese werden erst als Argument zu dem UMCP-Kommando COMMAND angegeben. Weitere definierte UMCP-Kommandos werden kurz in der folgenden Liste beschrieben:

AUTH Dieses Kommando muss als erstes von einem UMCP Client geschickt werden, um eine Authentisierung gegenüber einem Server durchzuführen. Erst wenn dies positiv bestätigt wurde akzeptiert der Server weitere Kommandos.

GET Mit diesem Kommando können Einstellungen, die für die aktuelle Sitzung gelten, abgefragt werden. Beispielsweise kann mit diesem Kommando und dem Argument `modules/list` eine Liste aller verfügbaren Kommandos abgefragt werden.

SET Um Einstellungen für die aktuelle Sitzung zu verändern kann dieses Kommando vom Client genutzt werden. Mit dem Argument `locale` kann beispielsweise die Spracheinstellung angepasst oder mit `interface` kann die Art des Client-Interfaces definiert werden (z.B. **web**).

VERSION Mit diesem Kommando kann die UMCP Version abgefragt werden.

COMMAND Dies ist das wichtigste UMCP-Kommando mit dem es möglich ist UMC-Befehle auszuführen. Der Befehl selbst wird als Argument übergeben während die Optionen zum dem UMC-Befehl im Nachrichtenrumpf enthalten sind.

4.3.2 Nachrichtenrumpf

In dem Rumpf der Nachricht ist ein Python Dictionary-Objekt, dass mit **Pickle** in der Protokoll Version 0 kodiert ist. Dieses Objekt enthält einige vordefinierte Schlüssel, kann aber um beliebige erweitert werden.

_options Der Wert dieses Schlüssels enthält ein weiteres Dictionary-Objekt in dem die Optionen für den UMC-Befehl enthalten sind.

_hosts In diesem Feld wird die Liste der Rechner spezifiziert auf denen das Kommando ausgeführt werden soll. Ist die Liste leer wird das Kommando nur auf dem lokalen Rechner ausgeführt.

_dialog Enthält eine für das Client-Interface aufbereitete Antwort des UMC-Befehls; wird also in Response-Nachrichten verwendet. Eine detaillierte Beschreibung des Wertes ist in Kapitel 7 zu finden.

_status In diesem Schlüssel wird der Status-Code einer Response-Nachricht gespeichert. Eine Liste aller definierte Codes ist in Anhang A zu finden.

_report Dieses Feld ist dazu gedacht kurze Nachrichten zu einer Antwort mit zuschicken. Dies kann beispielsweise bei Teilantworten genutzt werden um den aktuellen Stand der Bearbeitung zu umschreiben.

_module Dieses Feld ist ausschließlich für Teilantworten gedacht und wird genutzt um den Grund für die lange Bearbeitungszeit zu beschreiben. Dabei wird grundsätzlich zwischen zwei Varianten unterschieden: Einmal ist das Modul aktiv mit der Bearbeitung des Befehls beschäftigt oder es wartet gerade auf das Ergebnis eines weiteren Prozesses, das für die Antwort notwendig ist.

4.4 Python API

Das Python Module `univention.management.console.protocol` bietet eine komplette Implementierung von UMCP. Dazu gehören der Nachrichten-Parser, Objekte für Request- und Response-Nachrichten sowie Client, Server und Proxy Implementierungen. Im folgenden wird mit Schwerpunkt auf die Nachrichten Objekt die Verwendung der Klassen mittels Beispielen erklärt.

4.4.1 Request-Objekte

Um neue UMCP-Requests zu erstellen gibt es die Klasse Request.

```
class Request( Message ):
    def __init__( self, command, args = [], opts = {},
                 hosts = None, incomplete = False ):
```

Mittels des Konstruktors kann ein gültiges UMCP-Request Objekt erstellen, das direkt an einen Server versendet werden kann.

```
req = Request( 'COMMAND', args = [ 'cups/printer/show' ],
              opts = { 'printer' : 'Printer1' },
              hosts = [ 'slave1.domain.tld', 'slave2.domain.tld' ] )
```

Der letzte Parameter des Konstruktors *incomplete* dient dazu dem Server mitzuteilen, dass dieser Client diesen Befehl absichtlich unvollständig geschickt hat. Damit gibt es die Möglichkeit den Server dazu aufzufordern eine Art Formular zuschicken, dass die notwendigen Parameter für den jeweiligen UMC-Befehl abfragt.

Um ein Request-Objekt zu verschicken kann es an die request-Funktion eines UMCP-Client-Objektes übergeben werden. Das folgende kleine Beispielprogramm liest Benutzernamen und Passwort ein, verbindet sich mit einem lokalen UMCP-Server, authentisiert gegenüber dem Server und schickt einen UMCP-Request mit dem UMC-Befehl cups/list zu dem Server

```
import notifier
from getpass import getpass

notifier.init( notifier.GENERIC )

client = Client()

def auth( success, status, text ):
    global client
    print 'authentication', success, status, text
    req = umcp.Request( 'COMMAND', [ 'cups/list' ] )
    client.request( req )

client.signal_connect( 'authenticated', auth )
if client.connect():
    print 'connected successfully'
else:
    print 'ERROR connecting to daemon'
username = raw_input( 'Username: ' )
password = getpass()
client.authenticate( username, password )

notifier.loop()
```

4.4.2 Flags

Für die Steuerung des Verhaltens eines Kommandos im Web-Interface des UMCP-Clients können einem Request einige Flags mitgegeben werden.

web:startup (bool) Wird dieses Flag auf *True* gesetzt, wird mit diesem Befehl ein neuer Reiter geöffnet. Alle weiteren Flags mit der Präfix *web:startup* zeigen nur Wirkung, wenn dieses Flag auf *True* gesetzt ist.

web:startup_format (string) Mit diesem Flag wird der Titel für den Reiter definiert. Dabei können in dem Title Platzhalter wie in Python-Formatzeichenketten verwendet werden.

```
'Variable: %(key)s'
```

Die angegebenen Schlüssel in der Formatzeichenkette (in dem Beispiel *(key)*) müssen in den Optionen des UMCP-Kommandos definiert sein.

web:startup_dialog (bool)

web:startup_referrer (bool) Werden dieses beiden Flags gesetzt, dann wird nach dem Schließen des neuen Reiters wieder zu dem Reiter zurückgekehrt, von dem aus dieser Reiter geöffnet wurde.

web:startup_reload (bool) Wird auf den Titel des Reiter geklickt, wird der Befehl erneut ausgeführt, wenn dieses Flag auf **True** gesetzt ist.

Zum Setzen dieser Flags bietet die Request-Klasse die Methode `set_flag`, der Name und Wert des Flags übergeben werden:

```
req.set_flag( 'web:startup', True )
```

Zusätzlich gibt es noch die Methode `get_flag`, um den Wert eines Flags auszulesen und die Methode `has_flag` um zu prüfen, ob das Flag überhaupt gesetzt ist.

4.4.3 Vereinfachungen

Als Vereinfachungen zur Klasse `Request` stellt die API zwei weitere Klassen zur Verfügung. Während die Klasse `Command` den ersten Parameter (**command**) einspart bietet die Klasse `SimpleCommand` noch weitere Vereinfachungen:

```
class SimpleCommand( Request ):
    def __init__( self, command, options = {}, **flags ):
```

Bei der Erstellung von UMCP-Kommandos werden oft nur einige der verfügbaren Parameter benötigt. Die Klasse `SimpleCommand` ist darauf zugeschnitten. In den ersten beiden Parametern wird das Kommando (als Zeichenkette) und im zweiten die Optionen übergeben. Der dritte Parameter bietet die Möglichkeit eine beliebige Anzahl von Flags zu übergeben:

```
scmd = SimpleCommand( 'cups/printer/show', { 'printer' : 'lpt1' }, startup = True,
                    startup_dialog = True, startup_cache = False )
```

Die Flags werden ohne die Präfix 'web:' angegeben.

4.4.4 Response-Objekte

Um ein `Response`-Objekt zu erstellen, das als Antwort auf einen UMCP-Request verwendet werden soll, kann dem Konstruktor der Klasse `Response` ein `Request`-Objekt übergeben werden. Dabei werden die notwendigen Daten übernommen, so dass ein UMCP-Server die Zugehörigkeit erkennen kann.

```
import univention.management.console.protocol as umcp

req = umcp.Request( 'COMMAND', [ 'cups/list' ] )
resp = umcp.Response( req )
```

5 Berechtigungsmodell

Ein Ziel von UMC ist es die Möglichkeit zu schaffen Aufgabenbereiche an verschiedene Personen bzw Gruppen zu delegieren. Somit muss es die Fähigkeit besitzen für Benutzern genau zu konfigurieren auf welchen Rechnern sie welche Befehle ausführen dürfen. Wobei es eventuell notwendig sein kann bestimmte Befehle nur zuzulassen, wenn vorgegebene Parameter genutzt werden.

Genau dafür bietet UMC ein umfassendes Berechtigungsmodell, das es ermöglicht die Rechte eines Benutzers bzw. einer Gruppe auf mehreren Ebenen einzuschränken. Umgesetzt wird dieses Modell mittels drei neuer Univention Directory Manager Objekte, die im folgenden beschrieben werden:

Einstellungen: Console Operationen Dieses Objekt definiert eine Menge von UMC Befehlen, die von UMC-Modulen bereitgestellt werden. Diese Befehlssätze können dann bei der Definition von **Einstellungen: Console ACLs** Objekten verwendet werden um dafür Zugriffsrechte festzulegen.

Einstellungen: Console ACLs Ein Objekt von diesem Typ verbindet eine Liste von Befehlssätzen mit einer Liste von Rechnern und ordnet diese einer Kategorie zu. Wenn ein solches Objekt in einer Richtlinie **Console Zugriff** verknüpft wird hat dies folgende Bedeutung: Benutzer oder Gruppen, auf die diese Richtlinie zutrifft, dürfen die Befehle aus den Befehlssätzen auf den angegebenen Rechnern ausführen oder genau das Gegenteil, diese Befehlssätze sind auf den Rechnern verboten.

Richtlinie: Console Zugriff Dieses Richtlinien-Objekt ermöglicht es die zuvor beschriebenen **Console ACLs**-Objekte an Benutzer oder Gruppen zu binden um den Zugriff auf UMC-Befehle zu erlauben oder zu verbieten.

Falls der Zugriff auf den LDAP-Server zu einem Zeitpunkt nicht möglich ist, werden die zuletzt ermittelten Zugriffsrechte für Benutzer zwischengespeichert. So ist es möglich, dass sich Benutzer weiterhin anmelden und Befehle ausführen können.

6 Module

Die eigentliche Funktion eines UMCP-Servers ist in den Modulen implementiert. Jedes dieser Module definiert einen Satz von Befehlen, den es versteht und ausführen kann. Wie solche Module entwickelt werden und was dabei berücksichtigt werden muss wird in den folgenden Abschnitten beschrieben.

6.1 Aufbau

Der Aufbau eines UMC-Moduls ist mit dem eines Univention Directory Manager-Moduls vergleichbar. Es gibt ein definiertes Unterverzeichnis in dem ein UMC-Server nach Modulen sucht. Ein Unterschied ist dabei, dass UMC-Module im Gegensatz zu Univention Directory Manager-Modulen ein Verzeichnis nutzen und nicht nur aus einer einzelnen Datei bestehen. Dadurch gibt es bessere Möglichkeiten den Quellcode zu strukturieren.

Der UMC-Server importiert diese Module wie ein normales Python-Module. Damit ein Verzeichnis als ein gültiges UMC-Modul akzeptiert wird muss es folgende Kriterien erfüllen:

- In dem Verzeichnis muss sich eine Datei mit dem Namen `__init__.py` oder `__init__.pyo` befinden.
- Das Modul muss mit `import` geladen werden können
- In dem Modul müssen mindestens folgende Attribute definiert sein: `icon`, `short_description`, `long_description`, `categories`

Der Name eines Moduls leitet sich durch den Namen des Verzeichnisses ab, so dass dieser nicht extra definiert werden muss.

6.1.1 Beschreibung

Werden die gerade definierten Kriterien erfüllt, wertet der UMC-Server weitere Attribute des Moduls aus, die das Modul genauer beschreiben. Die folgende Liste enthält alle wichtigen globalen Attribute eines UMC-Moduls und beschreibt kurz ihre Bedeutung.

icon Jedem Modul wird ein Icon zugeordnet, das das Modul in graphischen Interfaces repräsentieren soll. Dafür wird im Modul ein Name angegeben, der vom Frontend mit einer Bilddatei assoziiert wird (Wie im Abschnitt 7.1 für das Element Image beschrieben).

```
icon = 'univention-config-registry/module'
```

short_description Eine kurze Beschreibung des Moduls, die als Name des Moduls in den Client-Interfaces verwendet werden kann.

```
short_description = 'Univention Configuration Registry'
```

long_description Eine ausführlichere Beschreibung des Moduls.

```
long_description = '''Set, get, search and remove
Univention Configuration Registry keys'''
```

categories Jedes UMC-Module kann zu Kategorien zugeordnet werden. Diese Kategorien können von Client-Interfaces genutzt werden die Module übersichtlicher darzustellen in dem immer nur Module aus einer Kategorie angezeigt werden. In diesem Attribut muss eine Liste der internen Namen der Kategorien eingetragen werden. Es gibt zwei vordefinierte Kategorien *all* und *misc*.

```
categories = [ 'all', 'misc' ]
```

command_description Dieses Attribut ist eines der wichtigsten da es die UMC-Befehle definiert, die von diesem Modul zur Verfügung gestellt werden. Details zu dieser Definition und der Implementierung von Befehlsfunktionen sind im folgenden Abschnitt zu finden.

```
import univention.management.console.handlers as umch

command_description = {
    'univention-config-registry/get': umch.command( ... ),
}
```

handler Das ist wie bei Univention Directory Manager-Modulen die Hauptklasse in der alle Befehlsfunktionen definiert werden müssen. Diese Klasse muss von `simpleHandler` erben in der Basisfunktionalitäten definiert sind, die für das Abhandeln eines Befehls benötigt werden.

```
class handler( umch.simpleHandler ):
    def __init__( self ):
        umch.simpleHandler.__init__( self )
```

6.1.2 Befehle

Jeder Befehl den ein Modul zur Verfügung stellt muss in dem Attribut **command_description** aufgeführt sein. Dabei wird für jedes Kommando ein Objekt vom Typ `command` angelegt.

```
import univention.management.console.handlers as umch
import univention.management.console as umc

command_description = {
    'univention-config-registry/get': umch.command(
        short_description = _( 'Get' ),
        long_description = _( 'Get a univention-config-registry value' ),
        method = 'univention-config-registry_get',
        values = { 'key': umc.String( 'Variable' ) },
    ),
}
```

Ein solches Objekt beschreibt die Eigenschaften eines Kommandos. Der Schlüssel, dem dieses Objekt zugeordnet wird, ist hierbei der interne Name des Kommandos, wie er auch in UMCP-Requests angegeben werden muss. Alle Attribute, die in dem obigen Beispiel angegeben wurde sind Pflichtangaben. In dem Attribut **short_description** ist ein übersetzbarer Name für den Befehl einzutragen. Zusätzlich dazu enthält das Attribut **long_description** eine detailliertere Beschreibung. Für die Assoziation des Befehls mit einer Funktion, die ausgeführt werden soll, ist das Attribut **method** vorgesehen. Hier wird eine Zeichenkette eingetragen, die den Namen der Funktion in der von `simpleHandler` abgeleiteten Klasse, definiert. Das Attribut **values** definierte eine Liste möglicher Attribute und ihre Typen, die vom UMCP-Server zur Verifikation der ankommenden Nachrichten genutzt werden kann.

Zwei weiter, aber optionale Parameter sind **startup** und **priority**. Mit **startup** kann definiert werden, ob dieser Befehl ohne Parameter aufgerufen werden kann. Ist dies der Fall wird

er beispielsweise im Web-Interface als Einstiegspunkt in einem Modul dargestellt. Um die Reihenfolge der Startup-Kommandos zu definieren kann jedem Kommando mit dem Parameter **priority** eine Priorität zugeordnet werden. Anhand dieser werden die Kommandos sortiert.

Bei einigen Befehlen ist es erforderlich, dass der Benutzer die Ausführung zuvor bestätigen muss. Soll dies generell für einen Befehl gelten, kann dies mit dem optionalen Parameter **confirm** definiert werden. Diesem wird als Wert eine Instanz der Klasse **Confirm** zugewiesen, wie in dem folgenden Beispiel:

Nach der Definition aller vorhandenen UMC-Befehle eines Moduls muss die Klasse **handler** entsprechend der Definitionen implementiert werden. Nach dem vorherigen Beispiel muss die Klasse mindestens eine UMC-Befehlsfunktion mit dem Namen **univention-config-registry_get** enthalten. Eine solche Klasse würde ungefähr wie folgt aussehen.

```
import univention.management.console.handlers as umch

command_description = {
    'univention-config-registry/get': umch.command(
        short_description = _( 'Get' ),
        long_description = _( 'Get a univention-config-registry value' ),
        method = 'univention-config-registry_get',
        values = { 'key': umc.String( 'Variable' ), },
        confirm = umch.Confirm( 'Confirmation', 'Are you sure that you want to set'
    ),
}
```

Die Bestätigung wird nur verlangt, wenn der Befehl vom Client nicht als unvollständig (Option **incomplete**) geschickt wird.

```
import univention.management.console.handlers as umch

class handler( umch.simpleHandler ):
    def __init__( self ):
        umch.simpleHandler.__init__( self )

    def univention-config-registry_get( self, object ):
        # handle command
        self.finished( object.id(), answer_dialog )
```

Jede UMC-Befehlsfunktion bekommt einen Parameter übergeben in dem sich das UMCP-Request Objekt befindet. Dies kann die Funktion nutzen um die übergebenen Optionen und Einstellungen abzufragen. Ist die Abarbeitung des Befehls abgeschlossen, muss dies dem UMC-Server mitgeteilt werden. Dafür gibt es die Funktionen **finished**, die in dem Fall aufgerufen werden muss. Dabei sind folgende Parameter anzugeben:

id Gibt die eindeutige Kennzeichnung des UMC-Befehls an, dessen Beendigung gemeldet werden. Diese kann von dem Request-Objekt **object** erfragt werden:

```
object.id()
```

dialog Dieser Parameter enthält die Antwort auf den Befehl. Der verwendete Datentyp ist dabei beliebig und sollte alle notwendigen Informationen enthalten.

report Kann optional angegeben werden und eine textuelle Beschreibung des Ergebnisses enthalten.

success Ist ebenfalls optional und definiert den Erfolg der Ausführung. Voreingestellt ist eine positive Rückmeldung.

Eine UMC-Befehlsfunktion muss auf diese Art und nicht mit ihrem Rückgabewert den Status der Bearbeitung melden, weil die Bearbeitung von UMC-Befehlen asynchron innerhalb des UMC-Servers durchgeführt werden, d.h. nach Beendigung der Funktion muss es noch keinen Status zur Bearbeitung geben, weil beispielsweise noch auf die Beendigung eines externen Prozesses gewartet wird.

Durch den Parameter **success** kann der Erfolg eines Befehls festgelegt werden. War der Befehl nicht erfolgreich, wird an den Client ein Fehler gemeldet und zusätzlich wird die Beschreibung aus dem Parameter **report** als Grund angegeben. Bei einer erfolgreichen Ausführung muss die Abarbeitung des Befehls innerhalb des UMC-Server noch nicht abgeschlossen sein. Es kann optional eine sogenannte "Aufbereitungsfunktion" definiert werden, in der die Antwort für den entsprechenden Client angepasst werden kann (Details dazu folgendes in dem nächsten Abschnitt). Beispielsweise kann die Antwort eines UMC-Befehls für einen interaktiven Client aufbereitet werden. Sollte für einen Befehl nur eine einfache positive Rückmeldung notwendig sein, kann dies direkt über die **finished**-Funktion durchgeführt werden. Dafür muss dem Parameter **dialog** der Wert **None** und dem Parameter **report** ein Text übergeben werden.

6.1.3 Aufbereitung für den Client

Die Antworten einer UMC-Befehlsfunktion sind meistens einfache Datenstrukturen, die sich noch nicht für interaktive Interfaces wie ein Web-Frontend verwenden lassen. Um eine Antwort auf einen UMCP-Request für ein bestimmtes Client-Interface aufzubereiten, können für jeden Befehl so genannte "Aufbereitungsfunktionen" definiert werden. Diese müssen genauso heißen wie die Befehlsfunktion selbst plus eine Präfix, die sich aus einem Unterstrich und dem Typ des Client-Interfaces zusammensetzt. Beispielsweise muss die "Aufbereitungsfunktionen" für die Befehlsfunktion **univention_config_registry_search** für ein Web-Interface **_web_univention_config_registry_search** heißen.

```
class handler( simpleHandler ):  
    ...  
    def _web_univention_config_registry_search( self, object, res ):
```

Eine "Aufbereitungsfunktionen" bekommt zwei Parameter. Im ersten Parameter ist das UMCP-Request Objekt enthalten. Im Zweiten ist die Antwort der UMC-Befehlsfunktion in ein UMCP-Reponse Objekt **dialog** eingebettet. Das Attribut **res (dialog.res)** enthält den Wert, der an den Parameter **dialog** bei Aufruf von **finished** übergeben wurde. Im Fall des Befehls **univention-config-registry/search** wäre in der Antwort beispielsweise ein Dictionary-Objekt mit allen gefundenen Variablen enthalten.

In solchen Funktionen kann mit Hilfe der UMC Dialog-Klassen (Details siehe Abschnitt [7](#)) eine Struktur zusammengebaut werden, die eine Seite beschreibt. Diese kann Textelemente, Eingabefelder, Bilder und Schaltflächen enthalten. Damit ist einerseits möglich fertige Antworten darzustellen oder fehlende Argumente abzufragen.

Wenn der gewünschte Antwortdialog zusammengebaut wurde, muss dieser in das UMCP-Response Objekt eingetragen werden und dies dann an die Funktion **revamped** übergeben werden. Beispiel:

```
def _web_univention_config_registry_search( self, object, res ):
    dlg = umcd.Dialog()
    ...
    # Ergebnis aus der Befehlsfunktion verarbeiten
    for item in res.dialog:
        ...

    res.dialog = dlg

    self.revamped( object.id(), res )
```

Die Trennung zwischen der UMC-Befehlsfunktion und der "Aufbereitungsfunktion" ist optional. Die UMC-Befehlsfunktionen können auch direkt den UMC-Dialog erzeugen und zurückgeben. Ist zusätzlich noch eine "Aufbereitungsfunktion" definiert wird diese abschließend noch aufgerufen und kann den UMC-Dialog noch verändern. Ein Beispiel für eine UMC-Befehlsfunktion, die direkt einen Dialog erzeugt würde wie folgt aussehen:

```
def univention_config_registry_search( self, object ):
    res = umcp.Response( object )

    result = umcd.List()
    ...
    # Tabelle mit Ergebnissen zusammenbauen
    for var in variables:
        result.add_row( ... )

    res.dialog = [ result ]

    self.finished( object.id(), res )
```

6.2 Tools

Für die Entwicklung von UMC-Modulen werden einige Hilfsfunktionen und Objekte zur Verfügung gestellt, die im folgenden kurz beschrieben werden.

6.2.1 Prozessverwaltung

Der asynchrone Framework bietet Möglichkeiten Kindprozesse zu starten und nach ihrer Beendigung informiert zu werden. In manchen Modulen ist diese asynchrone Technik schwieriger zu implementieren, besonders wenn eine Abfolge von Kommandos gestartet wird. Um dies zu vereinfachen werden in dem Modul `univention.management.console.tools` zwei Funktionen definiert. `run_process` bietet die Möglichkeit einen Kindprozess zu starten.

```
def run_process( command, timeout = 0, shell = True, output = True )
```

Die Funktion kehrt erst zurück, wenn der Prozess sich beendet hat oder ein definierter Timeout **timeout** in Millisekunden abgelaufen ist. In dem Parameter **command** wird in einer Zeichenkette oder als Liste der auszuführende Befehl angegeben. Ob der Befehl über eine Shell ausgeführt werden soll, kann über den Parameter **shell** festgelegt werden. Der letzte Parameter **output** definiert, ob die Ausgabe des Prozess (stdout und stderr) gespeichert werden soll.

Der Rückgabewert der Funktion ist ein Dictionary, das folgende Schlüssel enthält:

pid Wenn der Prozess noch läuft, d.h. er wurde nach dem angegebenen Timeout nicht beendet, dann ist hier die Prozess-ID enthalten, anderenfalls ist der Wert **None**

exit Wenn der Prozess beendet ist, dann ist hier der Exit-Code des Prozess gespeichert.

stdout, stderr Diese beiden Schlüssel enthalten einen File-Descriptor, der jeweils auf eine temporäre Datei verweist, die stdout und stderr enthalten. Sollte der Prozess noch laufen, ist die aktuelle Position das Ende der Datei.

Sollte ein Prozess nach dem angegebenen Timeout noch laufen, kann dieser mit der Funktion `kill_process` beendet werden.

```
def kill_process( pid, signal = 15, timeout = 0 )
```

Im ersten Parameter muss die Prozess-ID angegeben werden. In der Voreinstellung wird der Prozess "freundlich" mit einem Signal 15 (Parameter **signal**) beendet und es ist kein Timeout vorgegeben. Wenn die Beendigung des Prozesses nur für ein bestimmtes Intervall abgewartet werden soll, kann dies über den Parameter **timeout** (in Millisekunden) gesetzt werden.

7 Dialoge

In den UMC-Modulen wird die Ausführung von einzelnen UMC-Befehlen von der Logik des jeweiligen Client-Interfaces getrennt. Diese Technik ermöglicht es in einem Modul mehrere Client-Interfaces mit verschiedenen Arbeitsweisen zu implementieren. Zusätzlich ist diese Technik für den UMC-Proxy wichtig, da diese nur die direkten Antworten der Befehlsmethoden einsammelt und erst lokal die Umwandlung für das jeweilige Client-Interface vornimmt.

Die für ein Client-Interface aufbereitete Antwort beschreibt im Fall eines Web-Interfaces den logischen Aufbau einer Webseite. Dabei werden die einzelnen Elemente wie Texte, Eingabefelder, Auswahllisten und Schaltflächen durch einfache Strukturelemente ausgerichtet und positioniert. Für das individuelle Layout der einzelnen Elemente kann fast jeder Klasse als letzter Parameter des Konstruktors eine Schlüssel-Wert-Liste mitgegeben werden. Details dazu befinden sich im Anhang [B](#).

7.1 Elemente

In der Implementierung werden alle diese Elemente durch Python-Klassen repräsentiert. Für Strukturelemente stehen dabei folgende Klassen zur Verfügung:

Dialog Diese Klasse ist der Container für alle weiteren Elemente, die auf einer Seite angezeigt werden sollen. Die Elemente des Containers werden in einer Liste gespeichert und in der angegebenen Reihenfolge untereinander angezeigt werden.

```
class Dialog( base.Frame ):
    def __init__( self, elements = [] ):
```

Bei der Initialisierung eines solchen Objekts kann der Inhalt über den Parameter **elements** vorgegeben werden.

Frame Der Frame kann genutzt werden um den Inhalt einer Seite in mehrere Bereich aufzuteilen. Dafür können die gewünschten Inhalte eines Abschnitts in einen Frame eingetragen werden, in diese an den Parameter **elements** übergeben werden. Zusätzlich ist es möglich diesem Abschnitt mit dem Parameter mit **title** eine Überschrift zu geben.

```
class Frame( Element, list ):
    def __init__( self, elements = [], title = '' ):
```

List Diese Container erzeugt eine Tabelle mit einer optionalen Kopfzeile. Die einzelnen Spalten einer Zeile werden als Liste an die Klasse übergeben. Hierbei ist selbst darauf zu achten, dass jede Zeile die gleiche Anzahl von Spalten enthält.

```
class List( Element ):
    def __init__( self, header = None, content = None,
                 sec_header = None, attributes = {} ):
```

Der Parameter **header** kann eine Liste von Spaltentiteln enthalten und sprechend kann in dem Parameter **content** eine Liste von Listen mit Zelleninhalten definiert sein. Des weiteren ist es möglich eine zweite Kopfzeile zu definieren. Dafür kann der Parameter **sec_header** genutzt werden.

Objekte der **List**-Klasse können auch ineinander verschachtelt werden um komplexer Strukturen zu bauen. Als weitere Elemente stehen Klassen aus der folgenden Liste zur Verfügung. Bei den meisten dieser Klassen wird dem Konstruktor als erstes Argument der Name einer Option für einen UMCP-Request übergeben. Diese Verknüpfung wird benötigt, damit das Web-Frontend von Univention Management Console aus den Elementen die Werte für die zu versendenden UMCP-Requests ermitteln kann.

Image Mit Objekten dieser Klasse können Bilder integriert werden. Dabei werden diese Bilder mit einem textuellen Bezeichner beschrieben. Mittels einer Funktion aus der UMC Bibliothek kann das Frontend anhand dieser Angabe die Bilddatei ermitteln.

```
class Image( base.Element ):
    def __init__( self, tag = None, size = umct.SIZE_MEDIUM ):
```

Bei der Initialisierung eines **Image**-Objektes kann im Parameter **tag** direkt der Bezeichner übergeben werden. Mit dem optionalen Parameter **size** kann eine Größe des Bildes definiert werden. Dabei stehen folgende Größen zur Verfügung:

<i>Bezeichner</i>	<i>Pixel</i>
SIZE_MICROSCOPIC	8
SIZE_TINY	12
SIZE_SMALL	16
SIZE_MEDIUM	24
SIZE_NORMAL	32
SIZE_LARGE	64
SIZE_HUGE	92

Die Bezeichner für die verschiedenen Größen sind im Module ***univention.management.console.tools*** definiert.

TextInput, **ReadOnlyInput** Diese beiden Klassen sind für einzeilige Texteingabefelder zu verwenden. Die Variante **ReadOnlyInput** ist dabei ein Textfeld, das nicht bearbeitet werden kann.

```
class ReadOnlyInput( Input ):  
    def __init__( self, option, text, default = '',  
                 static_options = {} ):  
  
class TextInput( Input ):  
    def __init__( self, option, text, default = '',  
                 static_options = {} ):
```

Bei der Initialisierung einer dieser beiden Klassen sind mindestens zwei Parameter anzugeben. Der erste Parameter ***option*** definiert den Namen einer UMC-Befehlsoption, in die der Wert dieses Textfeld übernommen werden soll. Mit dem Parameter ***text*** kann dem Eingabefeld ein kurzer beschreibender Text zugeordnet werden. Des weiteren kann optional mit ***default*** noch ein Vorgabewert definiert werden, der initial in dem Eingabefeld stehen soll. Mit dem letzten Parameter kann eine Menge von UMC-Befehlsoptionen definiert werden, die zusätzlich an einen UMC-Befehl übergeben werden können.

Checkbox Um bool'sche Werte darzustellen können Objekte dieser Klasse genutzt werden.

```
class Checkbox( Input ):  
    def __init__( self, option, text, default = False,  
                 static_options = {} ):
```

Die Parameter des Konstruktors stimmen mit denen der Klassen **TextInput** überein. Einziger Unterschied ist der Typ des Parameters ***default***, der in diesem Fall Bool ist.

Selection Um Auswahllisten von einer fest definierten Menge von Werten darzustellen kann ein Objekt dieser Klasse erstellt werden. Dabei kann dem Objekt eine Liste der möglichen Werte bei der Initialisierung übergeben und über einen Vorgabewerte aus dieser Liste das vorausgewählte Element gesetzt werden.

```
class Selection( Input ):  
    def __init__( self, option, text = '', choices = [],  
                 default = '', static_options = {} ):
```

Die Parameter ***option***, ***text***, ***default*** und ***static_options*** entsprechen von der Bedeutung der Verwendung in den anderen Input-Klassen. Mit dem zusätzlichen Parameter ***choices*** kann eine Liste von Auswahlmöglichkeiten definiert werden.

Text, Date, Number Diese drei Klassen sind zur Darstellung von statischen Textelementen zu verwenden. Die Klassen **Date** und **Number** sind für Texte gedacht, die ein Datum bzw. eine Zahl enthalten. Durch die verschiedenen Klassen hat das Client-Interface die Möglichkeit die Layoutierung entsprechend anzupassen.

```
class Text( Element ):
    def __init__( self, text = '', attributes = {} ):
class Date( Text ):
    def __init__( self, date = '', attributes = {} ):
class Number( Text ):
    def __init__( self, number = '' , attributes = {} ):
```

Die Konstrukturen der Klassen bekommen alle nur einen Parameter, der den Wert für das Feld definiert.

Fill Diese Variante eines Textfeldes kann über mehrere Spalten beziehungsweise Zeilen gezogen werden. Voreingestellt ist eine Streckung über zwei Spalten, die mit dem Parameter **columns** angepasst werden kann. Wenn sich das Feld vertikal ausdehnen soll, kann dies mit dem optionalen Parameter **vertical** angegeben werden indem dieser auf **True** gesetzt wird. Mit dem Parameter **text** kann der Inhalt definiert werden.

```
class Fill( Text ):
    def __init__( self, columns = 2, text = '', vertical = False ):
```

HTML Dieses Element erlaubt es direkt HTML-Fragmente zuschreiben, die wie angegeben in die Webseite übernommen werden.

```
class HTML( Text ):
    def __init__( self, text = '', attributes = {} ):
```

Link Um einen Verweis auf externe Ressourcen zu erstellen kann diese Klasse benutzt werden.

```
class Link( text.Text ):
    def __init__( self, description = '', link = '', icon = None ):
```

Bei der Initialisierung eines Objektes dieses Types können zwei Parameter übergeben werden. Mit **description** kann ein Text angegeben werden, der statt der URI angezeigt wird, die mit Parameter **link** definiert wird.

Das folgende Beispiel erstellt eine einfache Tabelle mit zwei Eingabefeldern und einem Bild:

```
import univention.management.console.dialog as umcd
import univention.management.console.tools as umct

lst = umcd.List()

key = umcd.TextInput( 'key', \
    _( 'Univention Configuration Registry-Variable' ) )
descr = umcd.TextInput( 'description', _( 'Beschreibung' ) )
image = umcd.Image( 'actions/info', umct.SIZE_SMALL )
lst.add_row( [ image ] )
lst.add_row( [ key ] )
lst.add_row( [ descr ] )
```

Eine Vereinfachung bei der Erstellung von Dialog-Eingabefeldern bietet die Funktion **make** aus dem dialog-Modul von Univention Management Console. Damit lassen sich passende Dialogelemente für die Attribute eines Univention Management Console-Befehls automatisch erstellen. Beispiel:

```
import univention.management.console.dialog as umcd

text = umcd.make( self.[ 'univention-config-registry/search', 'key' ],
                  default = '*' )
```

Anhand der Syntaxdefinition zu diesem Parameter (aus dem Modul-Attribut **command_description**) kann die Funktion **make** das richtige Element ermitteln (in diesem Fall ein Texteingabefeld) und dies zurückliefern. Wenn für Befehlsparameter eigene Syntaxdefinitionen verwendet werden, dann muss für die Verwendung von **make** erst eine Regel definiert werden, wie für diese Syntax ein Element erstellt wird.

7.2 Interaktion

Um interaktive Client-Interfaces mit UMC zu gestalten ist es notwendig Antworten generieren zu können die Elemente enthalten, die weitere Aktionen auslösen. Dafür wird die Klasse `Button` bzw. davon abgeleitete Klassen verwendet, die dafür viele verschiedene Methoden bietet, um möglichst viele Varianten abdecken zu können.

Die primäre Aufgabe eines `Button`-Objektes ist bei Betätigung neue UMCP-Anfragen zu generieren, die dann an den Server geschickt werden. Dabei ist es notwendig, dass die UMCP-Anfragen nicht nur statisch vordefiniert werden können, sondern das einzelne Parameter oder sogar der UMC-Befehl selbst aus Einstellungen resultieren, die in dem Client-Interface vorgenommen wurden.

Dafür bietet die Klasse `Button` einige Methoden, die im Folgenden kurz an Beispielen illustriert werden sollen. Zusätzlich werden einige weitere Klassen beschrieben, die in diesem Zusammenhang Verwendung finden.

Wichtig für diese Technik ist eine Eigenschaft aller Elemente der Dialog-Klassen. Diese besitzen eine für den Prozess eindeutige ID, die dazu benutzt werden kann Verknüpfungen zwischen den einzelnen Elementen herzustellen bzw. auf sie zu referenzieren. Genau dies wird auch von der Klasse `Button` verwendet um auf Elemente zu verweisen aus denen Informationen bezogen werden, die für die Zusammensetzung der abzusetzenden UMCP-Anfragen notwendig sind.

Die folgende Liste bietet eine Übersicht der wichtigsten Klassen zur Erstellung von Antwortseiten mit Schaltflächen.

Action Die Klasse dient zur Beschreibung einer Aktion, die ausgeführt werden soll, wenn die Schaltfläche ausgewählt wird. Definiert wird eine Aktion durch einen UMCP-Request (**command**) sowie einer optionalen Liste von IDs von Elementen (**options**). Ist diese Liste nicht leer, dann werden die Werte aus den Elementen als Optionen an den UMCP-Request angehängt bevor dieser zum Server geschickt wird. Wird zusätzlich der optionale Parameter **selection** auf die ID einer Auswahlliste gesetzt, dann wird das UMCP-Kommando aus dem ausgewählten Eintrag genommen.

```
class Action( object ):
    def __init__( self, command = None, options = [], selection = False ):
```

Button Dies ist das wichtigste Element, wenn es darum geht ein interaktives Interface zu gestalten. Objekte dieser Klasse repräsentieren Schaltflächen hinter denen sich Aktionen verbergen, die wieder UMCP-Anfragen auslösen können. Details zu den Möglichkeiten werden im folgenden Abschnitt ausführlich beschrieben.

```
class Button( Text, Image ):
    def __init__( self, text, image = None, actions = [] ):
```

Bei der Erzeugung eines Objektes muss mindestens der Parameter **text** angegeben werden, der auf der Seite dargestellt wird. Optional können ein Bild mit dem Parameter **image** und eine Liste von auszuführenden Aktionen mit dem Parameter **actions** definiert werden.

SelectionButton Diese Klasse bietet die Möglichkeit eine Auswahlliste darzustellen. Dafür werden dem Konstruktor der Klasse die verfügbaren Werte als Liste von Paaren in dem Parameter **choices** übergeben.

```
class SelectionButton( Button ):
    def __init__( self, label = '', choices = [], actions = [], attributes = {},
                  close_dialog = True ):
```

Jedes Paar enthält als erstes einen internen Identifier und als zweites eine übersetzbare Zeichenkette, die im Frontend angezeigt wird. Wird ein Eintrag ausgewählt, so wird die Liste der Aktionen in dem Parameter **actions** ausgeführt. Wird bei einer der Action-Objekte der Parameter **selection** auf **True** gesetzt, dann wird das erste Argument des UMCP-Request von der Aktion (der Modul-Befehl) durch den internen Schlüssel des ausgewählten Elements ersetzt.

```
actions = ( umcd.Action( req, boxes, True ), umcd.Action( req_list ) )
choices = [ ( 'cups/printer/enable', _( 'Activate printers' ) ),
            ( 'cups/printer/disable', _( 'Deactivate printers' ) ) ]
select = umcd.SelectionButton( _( 'Select the operation' ), choices, actions )
```

Zusätzlich zu den angegebenen Auswahlmöglichkeiten werden automatisch noch die folgenden Punkte ergänzt:

- Auswahl umkehren
- Alles auswählen

Diese Operationen beziehen sich auf Checkboxes, die mit der Schaltfläche über die Aktionen verbunden sind.

ChoiceButton Diese Klasse ist ähnlich zu **SelectionButton**. Der Unterschied ist, dass jedem Eintrag ein eigener Befehl zugeordnet werden kann.

```
class ChoiceButton( Button ):
    def __init__( self, label = '', choices = [], attributes = {}, default = None,
                  close_dialog = True ):
```

Die Auswahlmöglichkeiten **Auswahl umkehren** und **Alles auswählen** können hier auch angegeben werden, müssen allerdings selbst in die Liste eingefügt werden. Dafür müssen die internen Schlüssel **::invert** und **::select_all** verwendet werden. Das gleiche gilt für die Auswahlmöglichkeit, die keine Aktion auslöst und somit gut als Vorauswahl verwendet werden kann. Diese ist über den Schlüssel **::none** hinzuzufügen.

SetButton, SearchButton, AddButton, SetButton Diese Klassen sind abgeleitet von **Button** und sind können für vordefinierte häufig verwendete Schaltflächen verwendet werden

Zusätzlich zu den Schaltflächen, die UMCP-Kommandos an den Server schicken, gibt es auch noch eine Variante, die Kommandos innerhalb des Web-Frontends verschicken. Beispiel für ein solches Kommando ist der Abbruch einer Aktion oder das Schließen eines Dialoges. Die folgende Liste zeigt die vordefinierten Schaltflächen-Klassen, die interne Kommandos versenden:

CancelButton Wurde ein Dialog geöffnet kann mit diesem solch einer Schaltfläche der Dialog geschlossen werden und zum aufrufenden Dialog zurückgekehrt werden.

CloseButton Führt die gleiche Operation aus wie der CancelButton, nur dass diese mit dem 'Schließen' bezeichnet wird.

ErrorButton Löst einen internen Fehler aus, der dann einen Fehlermeldung produziert.

ResetButton Kann verwendet werden um die Felder auf einer Seite auf den Vorgabewert zurückzusetzen. Dafür kann einem solchen Objekt im Parameter **fields** eine Liste von Element-IDs mitgegeben werden.

ReturnButton Wird eine solche Schaltfläche betätigt, dann wird des initale Kommando des Reiters erneut ausgeführt.

Im folgenden Beispiel wird eine einfache Suchmaske definiert, die aus einem Texteingabefeld und einer Schaltfläche besteht. Der Inhalt des Textfeldes wird dabei an die Option **key** für den UMC-Befehl `univention_config_registry/search` übergeben.

Bei der Erstellung des Textfeldes wird als erster Parameter der Name einer UMC-Befehlsoptions (in diesem Fall **key**) angegeben. Um an dem Button-Objekt eine Verknüpfung zu diesem Objekt zu hinterlegen wird eine Action definiert, die als Parameter ein UMCP-Request Objekt und eine Liste von relevanten IDs von Elementen übergeben bekommt. Wird diese Schaltfläche ausgewählt, dann werden alle angegebenen Action-Objekte der Reihenfolge nach analysiert und ausgeführt.

```
import univention.management.console.dialog as umcd
import univention.management.console.protocol as umcp

form = umcd.List()

form.set_header( [ umcd.Text( 'Search Form' ) ] )
text = umcd.TextInput( 'key', \
    _( 'Univention Configuration Registry-Variable' ) )
form.add_row( [ text ] )
req = umcp.Command( args = [ 'univention-config-registry/search' ] )
form.add_row( [ umcd.Button( _( 'Search' ), 'actions/search',
    umcd.Action( req, [ text.id() ] ) ) ] )
```

8 Häufig verwendete Elemente

Einige Kombinationen von Elementen werden häufiger in UMC-Modulen wiederverwendet. Damit diese nicht immer wieder neu zusammengesetzt werden müssen bietet das UMC-Dialog Modul einige Hilfsklassen, die solche Elementgruppen zur Verfügung stellen.

8.1 Information

Häufig ist es notwendig eine wichtige Information auf einer Seite hervorzuheben, die als Rückgabe auf einen UMC-Befehle geschickt wurde. Um diese hervorzuheben kann ein Symbol vorangestellt werden, dass dem Benutzer signalisiert, dass die Informationen wichtig sind. Um solch ein Element auf einer Seite darzustellen kann die Klasse `InfoBox` verwendet werden. Ein Beispiel dazu ist in [Abbildung 2](#) zu sehen.

```
class InfoBox( base.Text, image.Image ):
    def __init__( self, text = '', columns = 1, icon = 'actions/info',
                 size = umct.SIZE_SMALL ):

```

Die `InfoBox` kann in Tabellen verwendet werden. Damit der Inhalt über mehrere Spalten gestreckt werden kann gibt es den Parameter `columns`. Für das Symbol wird in der Vergabe 'Information' verwendet, welches mit dem Parameter `icon` verändert werden kann. Die Größe des Symbols ist über `size` zu setzen.

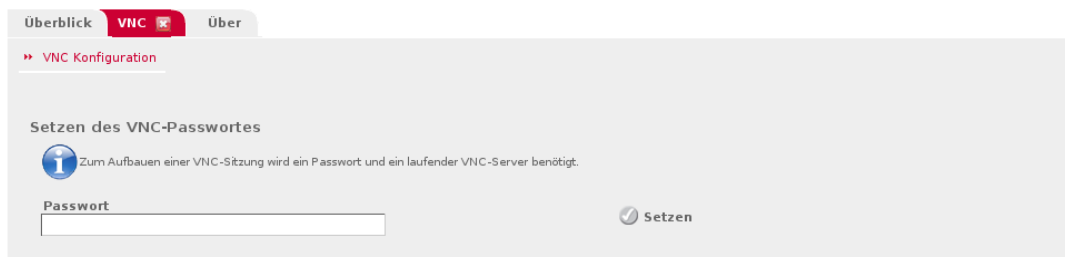


Abbildung 2: Element *InfoBox*

8.2 Fragen

Ein weiteres Element, das häufig bei interaktiven Interfaces verwendet wird ist eine Sicherheitsfrage, die abfragt, ob eine Aktion wirklich durchgeführt werden soll. Diese kann dann mit einer Bestätigung wirklich das Kommando ausführen oder bei einem Abbruch die Ausführung verhindern. Mit der Klasse `Question` steht eine Klasse zur Verfügung die solch ein Element zur Verfügung stellt. Ein Beispiel dazu ist in [Abbildung 3](#) zu sehen.

```
class Question( base.List ):
    def __init__( self, text = '', actions = [], okay = _( 'Ok' ) ):

```

In der Vorgabe wird ein Informationssymbol, ein Fragetext sowie eine [**Abbrechen**]- und [**OK**]-Schaltfläche angezeigt. Mit dem Parameter **text** kann die Frage angegeben werden. Um das Kommando bzw. die Kommandofolge zu definieren, die bei Betätigung der [**OK**]-Schaltfläche ausgeführt werden soll, ist Parameter **actions** zu verwenden. Optional kann über den Parameter **okay** der Text für die [**OK**]-Schaltfläche verändert werden.

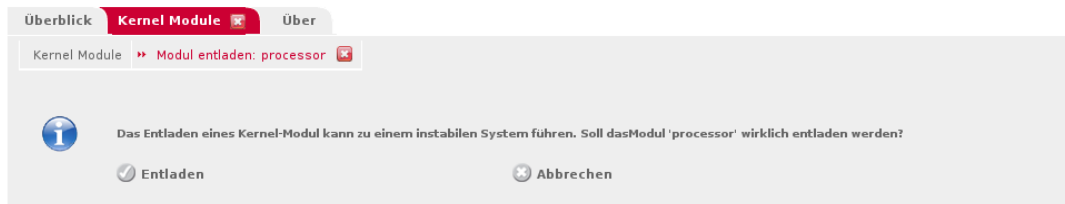


Abbildung 3: Element **Question**

Eine weitere Variante ist die Klasse `YesNoQuestion`, die ebenfalls für Fragen genutzt werden kann, die bestätigt oder abgelehnt werden. Im Unterschied zu der Klasse `Question` bietet diese Klasse weitere Optionen. Beispielsweise können die Texte beider Schaltflächen sowie ein Title für den Dialog gesetzt werden. Des Weiteren kann das zu verwendende Symbol verändert werden.

```
class YesNoQuestion( base.Frame ):
    def __init__( self, title = _( 'Confirmation' ), text = _( 'Are you sure?' ), acti
```

8.3 Suchdialog

In vielen Modulen wird die Möglichkeit geboten nach relevanten Informationen zu suchen bevor beispielsweise eine Bearbeitung möglich ist. Hierfür können in der Regel einfache Filter definiert, ob die gewünschten Daten schneller zu finden. Für solche Suchdialoge bietet das Dialog-Modul von Univention Management Console eine Hilfsklasse, die es ermöglicht mit wenigen Angaben immer gleiche aufgebaute Suchdialoge zu erstellen, wie in Abbildung 4 zu sehen ist.

```
class SearchForm( base.List ):
    def __init__( self, command = None, fields = [], opts = {} ):
```

Bei der Verwendung der Klasse müssen zwei Parameter angegeben werden. Der erste Parameter **command** definiert das auszuführende UMC-Kommando, das ausgeführt werden soll, um die Suche zu starten. Der Zweite Parameter definiert den Aufbau der Felder im Suchdialog. Dabei wird mit Listenstrukturen eine Tabelle zusammengestellt. Jede Zeile in der Tabelle wird als eine Liste dargestellt. In diesen Listen sind die einzelnen Zellen enthalten, die als Tupel definiert werden, welche aus einem Element und einem Vorgabewert bestehen. Der Vorgabewert wird dann gesetzt, wenn die Schaltfläche [**Zurücksetzen**] ausgewählt wird, die automatisch von der `SearchForm` erstellt wird. Im folgenden Beispiel wird ein Suchdialog erstellt, bei dem zwei Kriterien definiert sind. Einmal kann eine Kategorie ausgewählt werden und es kann der Wert eines Attributes mit einem Textfilter

eingegrenzt werden. Verwendet wird dieser Programmcode im Univention Configuration Registry-Modul. In der Abbildung 4 ist das Ergebnis zu sehen.

```
select = umcd.make( self[ 'baseconfig/search' ][ 'category' ],
                    default = category,
                    attributes = { 'width' : '200' } )
key = umcd.make( self[ 'baseconfig/search' ][ 'key' ],
                default = key,
                attributes = { 'width' : '200' } )
text = umcd.make( self[ 'baseconfig/search' ][ 'filter' ],
                 default = filter,
                 attributes = { 'width' : '250' } )

form = umcd.SearchForm( 'baseconfig/search', [ [ ( select, 'all' ), '' ],
                                              [ ( key, 'variable' ), ( text, '*' ) ] ] )
```




Abbildung 4: Element *SearchForm*

9 FAQ

Bei der Entwicklung von UMC-Modulen tauchen einige Fragen und Probleme häufiger auf. Im Folgenden befindet sich eine Liste einiger dieser Punkte.

Nach einer Abfolge von UMCP-Befehlen, soll die selbe Seite wieder angezeigt werden.

Beispielsweise wird die Liste von verfügbaren Druckern angezeigt und es soll über eine Schaltfläche die Möglichkeit bestehen einige der Drucker zu reaktivieren und anschließend die Druckerliste wieder anzuzeigen. Dafür reicht es aus über die Schaltfläche den Befehl zum Reaktivieren der Drucker zu versenden. Wenn dieser Befehl keinen Dialog und auch keinen Report zurückliefert, wird automatisch der Befehl, der zum aktuellen Reiter gehört, erneut ausgeführt.

Müssen die Befehls- und die Aufbereitungsfunktion in separate Dateien?

Der UMC-Server lädt die UMC-Modul wie ein Python-Modul. Dadurch ist es notwendig, dass in dem Modul Verzeichnis mindestens eine Datei mit dem Namen `__init__.py` zu finden ist. In diesem Modul sucht der UMC-Server nach einer Klasse `handler`. Diese Klasse muss die Befehls- und Aufbereitungsfunktionen zur Verfügung stellen, damit der UMC-Server sie findet. In vielen Modulen werden die Befehls- und Aufbereitungsfunktionen in unterschiedlichen Dateien definiert. In dem Fall wird in der Datei mit den Aufbereitungsfunktionen eine Klasse definiert, von der die `handler`-Klasse erbt und somit diese Funktionen auch dort verfügbar sind.

A Status-Codes

<i>Status-Code</i>	<i>Beschreibung</i>
200	OK, Operation war erfolgreich
210	OK, Teilantwort
300	Der UMC-Befehl ist nicht verfügbar bzw. nicht erlaubt
400	Ungültige UMCP-Nachricht
401	Unbekanntes UMCP-Kommando
402	Nicht erlaubte UMCP-Kommando Argumente
403	Unvollständiges UMCP-Kommando
404	Ungültiger Nachrichtenrumpf
405	Ungültiger Nachrichtenkopf
410	Nicht autorisiert
411	Authentisierung ist fehlgeschlagen
412	Benutzerkonto ist abgelaufen
413	Benutzerkonto ist deaktiviert
414	Der Zugriff auf UMC wurde verweigert
415	Die Ausführung des UMC-Befehls ist untersagt
500	Interner Fehler
501	Der Request zu einer Response-Nachricht konnte nicht gefunden werden
502	Ein UMC-Modulprozess hat sich unerwartet beendet
503	Die Verbindung zum UMC-Modulprozess wurde unterbrochen
600	Ein Fehler ist während der Verarbeitung eines UMC-Befehls aufgetreten
601	Der spezifizierte Zeichensatz bzw. die Spracheinstellung sind ungültig

B Layout-Attribute

Layout-Attribute werden bei der Erstellung von Elementen in einem Dictionary angegeben. Dabei sind der Name sowie der Wert als eine Zeichenkette anzugeben.

<i>Attribut</i>	<i>Wert</i>	<i>Beschreibung</i>
align	left, center, right	Gibt die Ausrichtung des Elementes innerhalb der Tabellenzelle an
colspan	Anzahl der Spalten	Das Element wird sich einer Tabelle über die angegebene Anzahl von Spalten erstrecken
type	Bezeichner	Definiert einen Bezeichner für das Element, der in Cascading Style Sheeting (CSS) verwendet werden kann
width	Pixel	Definiert die Breite des Elementes in Pixel