Backlog

Wirtschaftsprojekt «px: PEAX Command Line Client»

Patrick Bucher

16.12.2019

#	User Story	Status	Story Points
1	Konfiguration sämtlicher Umgebungen	umgesetzt in Sprint 1	1
2	Erweiterung der CI-Pipeline	umgesetzt in Sprint 1	5
3	Login mit Zwei-Faktor-Authentifizierung	umgesetzt in Sprint 1	3
4	Sichere Verwahrung der Tokens	umgesetzt in Sprint 1	5
5	Handhabung mehrerer Umgebungen	umgesetzt in Sprint 2	3
6	Generische `GET`-Schnittstelle	umgesetzt in Sprint 2	3
7	Automatische Aktualisierung von Tokens	umgesetzt in Sprint 2	5
8	Login für Agent API	umgesetzt in Sprint 2	3
9	Verbesserung der Hilfe-Funktion	umgesetzt in Sprint 2	3
10	Vollzugsmeldungen mit `-v`/`-verbose`-Flag	umgesetzt in Sprint 2	1
11	Verbesserung der Quellcodedokumentation	umgesetzt in Sprint 3	1
12	Aktuelle Version ausgeben	umgesetzt in Sprint 3	1
13	Einliefern von Dokumenten per Agent API	umgesetzt in Sprint 3	3
14	Generische `POST`-Schnittstelle	umgesetzt in Sprint 3	3
15	Generische `PUT`-Schnittstelle	umgesetzt in Sprint 3	3
16	Generische `PATCH`-Schnittstelle	umgesetzt in Sprint 3	3
17	Generische `DELETE`-Schnittstelle	umgesetzt in Sprint 3	1
18	Rekursives Hochladen von Dokument-Ordnern	umgesetzt in Sprint 3	5
19	Fehlerkorrekturen: Bugs 3, 4 und 5	umgesetzt in Sprint 4	3
20	Statusangabe bei Upload von Dokument-Ordnern	umgesetzt in Sprint 4	1
21	Nebenläufiger Upload von Dokument-Ordnern	umgesetzt in Sprint 4	1
22	Automatisches Tagging hochgeladener Ordner	umgesetzt in Sprint 4	5
	Lokale Umgebung unterstützen	offen	
	PEAX ID als Variable in der Ressourcenangabe	offen	
	Verbesserung der Testabdeckung	offen	
	Automatische Formatierung von JSON-Ausgaben	offen	
	Überarbeitung der README	offen	

Sprints

Sprint	Stories geplant	Stories umgesetzt	Aufwand	Stories offen	h/SP
1	6 (1-6), 20 SP	4 (1-4), 14 SP	14.5h	2 (5-6), 6 SP	1.05
2	6 (5-10), 18 SP	6 (5-10), 18 SP	20.5h	0	1.15
3	8 (11-18), 20 SP	8 (11-18), 20 SP	19.5h	0	0.98
4	4 (19-22), 10 SP	4 (19-22), 10 SP	11.5h	0	1.15

User Stories

Die User Stories haben drei Grössen: S (small: 1 Story Point), M (medium: 3 Story Points) und L (large: 5 Story Points). Diese Grössen dienen zur relativen Einschätzung der Story-Grössen zueinander, und sollen nicht in eine Stundenplanung heruntergerechnet werden. Vielmehr sollen sie dazu dienen, übergrosse Stories als solche zu erkennen, um diese herunterbrechen zu können. Am Ende eines jeden Sprints soll eingeschätzt werden, wie viele Story Points ungefähr machbar sind.

Story 1: Konfiguration sämtlicher Umgebungen

Als Entwickler möchte ich sämtliche relevanten PEAX-Umgebungen vorkonfiguriert haben, damit diese dem Nutzer zur Verfügung gestellt werden können.

Akzeptanzkriterien:

1. Es sollen die Umgebungen `dev`, `test`, `devpatch`, `testpatch`, `stage`, `prod`, `perf` und `prototype` zur Verfügung stehen.
2. Für jede Umgebung muss eine erreichbare URL zum jeweiligen Identity Provider und zu den relevanten APIs (User API, Admin API, Agent API) automatisch generiert werden können.

Notizen

- Die verschiedenen Umgebungen sind in einer statischen Liste aufgezählt.
- Die `init()`-Funktion des `env`-Moduls iteriert über diese Liste und generiert die URLs und Realm-Bezeichnungen anhand des Umgebungsnamen.
- Da bei `prod` benutzerfreundliche, kurze URLs verwendet werden, wird diese Umgebung separat gehandhabt.

Testprotokoll

- Der Unit Test `env_test.go` ist gemäss *Table Driven Test Design* (Donovan & Kernighan, 2015) entwickelt worden.
- Es wurde manuell mit Logins auf verschiedenen Umgebungen getestet.
- Die Umgebungen `perf` und 'prototype waren heruntergefahren und konnten so nicht getestet werden.
- Auf der Umgebung `prod` stehen derzeit nur Benutzer mit aktivierter Zwei-Faktor-Authentifizierung zur Verfügung. Da dies noch nicht umgesetzt
- Die Antworten wiesen wie erwartet die folgenden HTTP-Stati auf:

    1. Login vor der Registrierung: `401` (nicht möglich)
    2. Login nach derRegistrierung bzw. vor der Verifizierung: `403` (nicht möglich)
    3. Login nach der Verifizierung: `200` (erfolgreich)

Story 2: Erweiterung der CI-Pipeline

Als Entwickler möchte ich `px` in Skripts einbauen können, welche anschliessend in der CI-Pipeline berücksichtigt, d.h. ausgeführt werden.

Akzeptanzkriterien:

1. Der Ausführungsschritt `script` soll nur dann ausgeführt werden, wenn die vorherigen Schritte `build` und `test` erfolgreich waren.
2. Um der Pipeline ein weiteres Skript hinzufügen zu können, soll die neu erstellte Skriptdatei nur an einer einzigen Stelle (Konfigurationsdatei) hinzugefügt werden müssen.

Notizen

- Es wurde zunächst viel Zeit aufgrund mangelhafter Bash-Kenntnisse verloren, v.a. beim Iterieren über Liste von Skriptdateinamen.
- Das erste Akzeptanzkriterium wurde nur teilweise eingehalten. Zwar wird vor den Testskripts immer ein Build ausgeführt, um ein entsprechendes ausführbares Artefakt zur Verfügung zu haben, dies geschieht jedoch nicht über den `build`-Schritt, der ein Artefakt für Windows, Linux und macOS erstellt.
- Nach Überwindung der Anfangsschwierigkeiten konnte die Pipeline sehr schnell umgesetzt werden; der Aufwand wurde letztlich überschätzt.
- Zusätzlich wurde die Pipeline lokal ausführbar gemacht, indem die Umgebungsvariablen nicht zwingend von GitLab, sondern aus einem optionalen, lokalen Skript namens `envvars.sh` geladen werden.
- Der Shell-Code wurde zum Schluss überarbeitet und dabei vereinfacht.

Testprotokoll

- Zunächst wurde ein Skript erstellt, das die anderen Skripts ausführt.
- Es wurde je ein scheiterndes und ein durchlaufendes Skript erstellt und in die Testliste des vorher genannten Skript eingetragen.
- Die Pipeline wurde für GitLab (`.gitlab-ci`) konfiguriert. Der erste Test ist erfolgreich gescheitert, d.h. das scheiternde Skript hat die Pipeline wie gewünscht zum Abbruch gebracht.
- Das scheiternde Skript wurde nun aus der Liste der auszuführenden Tests entfernt, sodass die Pipelien nun durchlief.
- Es wurde ein minimales, jedoch produktives Testskript für `px help` erstellt und integriert.
- Es wurde ein Login-Testfall erstellt, der die Credentials als Umgebungsvariablen von der GitLab-Konfiguration bezieht.
- Die gescheiterten Loginversuche mit `px login` gaben zunächst den Status 0 zurück, was korrigiert werden musste.
- Das Login-Skript hat schliesslich wie gewünscht funktioniert.

Story 3: Login mit Zwei-Faktor-Authentifizierung

Als Benutzer möchte ich mich per Zwei-Faktor-Authentifizierung einloggen können, damit ich `px` auch mit entsprechend konfigurierten Zugängen verwenden kann.

Akzeptanzkriterien:

1. Die Abfrage des zweiten Faktors soll interaktiv passieren.
2. Es sollen die Authentifizierungsarten SMS und OTP (One-Time Password) unterstützt werden.
3. Das Login soll bei entsprechend konfigurierten Benutzerkonti auch weiterhin ohne Zwei-Faktor-Authentifizierung funktionieren.

Notizen

- Bevor die bestehende `login`-Funktion erweitert werden konnte, musste sie zunächst etwas aufgeräumt werden.
- Hierzu wurden verschiedene Teilaspekte (interaktive Abfrage fehlender Credentials, Erstellen des Requests und Parsen der Response) in eigene Funktionen ausgelagert.
- Die `script`-Pipeline zahlte sich bereits aus, zumal die `login`-Funktion direkt in `cmd/px.go` implementiert war und darum nicht durch einen Unit Test abgedeckt werden konnte.
- Das Refactoring wurde schliesslich ausgedehnt; es enstanden die neuen Untermodule `px/tokenstore` und `px/utils`.

- Es wurden neue Datenstrukturen erstellt, etwa für die Credentials (mit und ohne 2FA), und für den Login-Payload mit 2FA
- Der Code wurde neu organisieren, was nötig und sinnvoll war.

Testprotokoll

- Beim Refactoring traten immer wieder Build-Fehler auf, die jedoch einfach zu beheben waren.
- Logik-Fehler traten jedoch keine auf.
- Login-Versuche führten zunächst zu einem HTTP-Fehlerstatus `400` (Bad Request). Dieser Fehler wurde lange untersucht, einschliesslich einer Debugging-Session auf dem IDP-Server.
- Es stellte sich heraus, dass beim Login kein JSON-Payload unterstützt wird, sondern nur `form-urlencoded`; die serverseitige Fehlersuche stellte sich als sinnlos heraus.
- Nach der Umstellung von `application/json` auf `form-urlencoded` funktionierte die 2FA schliesslich problemlos.
- Das Login wurde mit SMS und TOTP erfolgreich manuell getestet. Automatisierte Tests sind aufgrund der interaktiven Eingabe inmitten des Prozesses nicht möglich.
- Das Login ohne 2FA wird weiterhin via Skript-Pipeline getestet.

Story 4: Sichere Verwahrung der Tokens

Als Benutzer möchte ich, dass beim Login geholte Tokens sicher lokal verwahrt werden, damit ein Angreifer diese nicht auslesen kann.

Akzeptanzkriterien:

1. Die Credentials (Benutzername und Password) werden zu keinem Zeitpunkt lokal persistent abgespeichert.
2. Refresh Tokens, die von der Produktivumgebung (`prod`) geholt werden, dürfen standardmässig nicht im Klartext abgespeichert werden.
3. Access und Refresh Tokens von nicht-produktiven Umgebungen, sowie Access Tokens der Produktivumgebung, können lokal im Klartext abgespeichert werden, sofern dies der einfacheren Bedienbarkeit zuträglich ist (weniger Passwortabfragen durch den Keystore).
4. Der Benutzer soll das Standardverhalten für die jeweiligen Umgebungen (produktiv: nur sichere Verwahrung; nicht-produktiv: Verwahrung im Klartext) mit den Kommandozeilenparametern `-safe` bzw. `-unsafe` übersteuern können.
5. Die sichere Verwahrung der Tokens muss Windows, macOS und Linux funktionieren.
6. Auf Systemen ohne GUI soll zumindest die unsichere Variante der Token-Verwahrung funktionieren.

Notizen

- Die Umgebungskonfiguration wurde um ein Flag (`Confidential`) erweitert, das besagt, ob für die jeweilige Umgebung die Tokens per default sicher oder unsicher verwahrt werden sollen.
- Die Library `zalando/go-keyring` kann Schlüssel auf Linux, macOS und Windows sicher im nativen Keystore verwahren.
- Die Konfiguration des nativen Keystores ist für jede Plattform anders und ist im README des Projekts dokumentiert.
- Bei einem Anwendungsfall wie dem Upload musste die Logik erweitert werden, sodass der sicher abgelegte Token für die Autorisierung verwendet werden kann.
- Die Tokens sind jeweils mit einem Schlüssel der Form `px:Umgebung:TokenTyp` abgelegt, z.B. `px:prod:accessToken` oder `px:test:refreshToken`.

Testprotokoll

- Der Unit Test `env_test.go` wurde um das `Confidential`-Flag erweitert, wobei nur `prod` entsprechend konfiguriert ist.
- Tests auf `prod` mit Linux (Tokens schreiben) funktionierten nachdem der Keystore mithilfe von Seahorse korrekt konfiguriert worden war.
- Der Skript-Test `ci-px-login-test.sh` wurde erweitert und in `ci-px-login-logout-test.sh` umbenannt, sodass fortan auch das Logout getestet wird.
- Darin wird nach dem Login mit `jq` geprüft, ob das Feld `access_token` für die Umgebung `test` in `~/.px-tokens` vorhanden ist.
- Nach dem Logout wird das Fehlen desselben geprüft
- Auf Windows sind die Tokens in der Anwendung *Credential Manager* unter *Windows Credentials* zu finden.
- Auf macOS sind die Tokens in der Anwendung *Keychain Access* unter *login* zu finden.

Story 5: Handhabung mehrerer Umgebungen

Als Benutzer möchte ich, dass `px` meine Befehle standardmässig gegen die Umgebung ausführt, auf der ich mich zuletzt eingeloggt habe, damit ich nicht immer eine Umgebung per Kommandozeilenparameter anwählen muss.

Akzeptanzkriterien:

1. Es muss einen Unterbefehl geben, der mir die aktuelle Umgebung (d.h. die Umgebung, auf der sich der Benutzer zuletzt eingeloggt hat) anzeigt.
2. Es muss einen Unterbefehl geben, womit eine Umgebung mit bereits aktivem Logging als die Standardumgebung gesetzt werden kann.
3. Bei allen Befehlen, die gegen die API operieren, soll die Umgebung mit einem Kommandozeilenparameter `-e` bzw. `-env` spezifiziert werden können.

4. Fehlt der Parameter `-e` bzw. `-env`, ist die Standardumgebung zu verwenden (zuletzt eingeloggt bzw. manuell als Standard gesetzt via `px env`).

Notizen

- Inspiriert durch `oc project` soll `px` einen Unterbefehl namens `env` enthalten. Wird er ohne Parameter aufgerufen, zeigt er die aktuelle Arbeitsumgebung an. Wird er mit Parameter aufgerufen, wird die aktuelle Arbeitsumgebung entsprechend gesetzt, sofern ein Login (Token Pair) dazu existiert.
- Die Datenstruktur `TokenStore` wird dazu um ein Feld `DefaultEnvironment` erweitert, um die Umgebung über mehrere Aufrufe von `px` hinweg abzuspeichern.
- Das Wechseln auf eine unbekannte Umgebung oder auf eine Umgebung ohne Tokens ist nicht zulässig.

Testprotokoll

- Das Testskript `ci-px-env-test.sh` führt zunächst ein Login auf `test` aus und prüft dann direkt in `~/.px-tokens`, ob `test` die Standardumgebung ist.
- Es wird zudem geprüft, ob `px env` die gleiche Umgebung ausgibt, die in `~/.px-tokens` als Standard abgelegt ist.
- Um den Wechsel der Umgebung zu testen, musste ein weiteres Login eingerichtet werden.
- Das Testskript `ci-px-env-test.sh` verwendet dieses zusätzliche Login, um zwischen den Umgebungen `test` und `dev` hin- und herzuspringen.

Story 6: Generische `GET`-Schnittstelle

Als Benutzer möchte ich einen `get`-Befehl zur Verfügung haben, damit ich lesend auf meine Ressourcen zugreifen kann.

Akzeptanzkriterien:

1. Dem Befehl kann ein beliebiger Ressourcenpfad mitsamt Query-Parametern mitgegeben werden.
2. Die Base-URL der jeweiligen API und Umgebung wird dem Ressourcenpfad automatisch vorangestellt.
3. Im Falle eines erfolgreichen Zugriffs (`200 OK`) soll der resultierende Payload auf die Standardausgabe (`stdout`) ausgegeben werden.
4. Im Falle eines fehlerhaften Zugriffs soll der Status-Code auf die Standardfehlerausgage (`stdout`) ausgegeben werden.

Notizen

- Eine komfortable Zusatzfunktion wäre, dass man Pfade nicht komplett mit der PEAX ID ausschreiben müsste (`document/api/v3/account/455.5462.5012.69/collection`), sondern einen Platzhalter wie `{peaxId}` verwenden könnte (`document/api/v3/account/{peaxId}/collection`). Dies soll jedoch nicht Bestandteil dieser Story sein.

Testprotokoll

- Das Testskript `ci-px-get-test.sh` führt ein Login auf `test` aus und lädt sich die Ressource `document/api/v3/account/455.5462.5012.69/collection`. Das Ergebnis wird mittels Pipe an `jq` weitergeleitet, das scheitern würde, wäre der Payload kein korrektes JSON.

Story 7: Automatische Aktualisierung von Tokens

Als Benutzer möchte ich dass ein Request, der aufgrund eines abgelaufenen Access Tokens scheitert, mit einem neuen Access Token erneut versucht wird, damit ich mich nicht ständig neu einloggen muss.

Akzeptanzkriterien:

1. Der Retry-Mechanismus soll für den Benutzer transparent sein.
2. Der neue Access Token soll anhand des Refresh Tokens ausgestellt werden.
3. Kann aufgrund eines abgelaufenen Refresh Tokens kein neuer Access Token geholt werden, soll dies dem Benutzer gemeldet werden.

Notizen

- Der Token Store wird im Hauptprogramm (`cmd/px.go`) derzeit wie eine Map (Key: Umgebung, Value: Token Pair) angesprochen. Auf die sicher verwahrten Tokens muss separat zugegriffen werden. Dieser Zugriff soll vereinheitlicht werden, was ein Refactoring erfordert.
- Das Usage Log, das die Anzahl Aufrufe und das Datum des letzten Aufrufs von `px` trackte, wurde entfernt.
- Sicher verwahrte Schlüssel werden neu in `./px-tokens` als Dummy-Eintrag abgelegt, sodass der Token Store ohne Zugriffe auf den Keystore über die Information verfügt, ob zu einer Umgebung überhaupt ein sicher verwahrter Schlüssel vorhanden ist.

- Bei den Dummy-Einträgen für sicher verwahrte Tokens wurde zunächst ein eigenartiger Datumswert abgelegt. Hierbei handelte es sich um den Zero-Wert der Struktur `time.Time`. Das Problem konnte behoben werden, indem `time.Time` als Pointer statt als Wert verwendet wird.
- Um den automatischen Retry-Mechanismus umzusetzen, musste zuerst herausgefunden werden, wie man anhand des Refresh Tokens einen neuen Access Token erhalten kann. Dies passiert über den gleichen Endpoint wie das Login, nur dass die Credentials mit dem `grant_type refresh_token` (statt `grant_type password`) und dem Refresh Token als Payload (statt Benutzername/Passwort) mitgegeben werden. Dieser Mechanismus wurde per Reverse Engineering ermittelt. Hierzu kann man sich auf dem Portal einloggen, für über fünf Minuten warten, eine Aktion auslösen, die mit dem Server kommuniziert – und schon sieht man den entsprechenden Request ablaufen.
- Die Zwei-Faktor-Authentifizierung läuft ab als Folge von Request, Response, Request, wobei vor dem zweiten Request eine interaktive Eingabe des Benutzers (SMS- oder TOTP-Code) erforderlich ist. Um die Konsoleneingabe vom Request-Mechanismus zu entkoppeln, wurde die Funktion `requestTokenPair` um einen Funktionsparameter namens `secondFactorPrompt` erweitert. Eine entsprechende Funktion `promptSecondFactor` erwartet einen String als Prompt (z.B. `"SMS Code"` oder `"OTP Code"`, fragt die entsprechende Information vom Benutzer ab und gibt sie zurück – oder einen Fehler, falls die Eingabe abgebrochen wurde. Dank dieser Entkopplung konnte der Request-Code grösstenteils aus dem Hauptprogramm entfernt werden.
- Nachdem aller Code, der HTTP-Requests verwendet, von `cmd/px.go` in das `requests`-Modul verschoben werden konnte, hatte das Hauptprogramm keine Referenz mehr auf das HTTP-Package.
- Der urpsrüngliche Ansatz, einen Request (mit aktualisiertem `Authorization`-Header) erneut abzuschicken, funktioniert leider nicht bei Requests mit einem Body. Grund dafür ist, dass der Request Body bei diesem Vorgang konsumiert wird. Der Request muss also für den erneuten Versuch neu aufgebaut werden. Im neuen Lösungsansatz erwartet zentrale Funktion `doWithTokenRefresh` nicht mehr einen blossen Request zur Ausführung, sondern eine Funktion, die einen entsprechenden Request generiert. So können die Implementierungsdetails vom Retry-Mechanismus entkoppelt werden.
- Beim Anfordern eines neuen Token Pairs anhand des Refresh Tokens wird nicht ein neuer Access Token ausgestellt, es wird auch der Refresh Token aktualisiert. Somit kann ein Benutzer nach einem Login so lange mit `px` arbeiten, wie er will, solange zwischen zwei Requests nicht mehr als 30 Minuten vergehen. Wichtig ist, dass auch der aktualisierte Refresh Token im Token Store abgelegt wird.

Testprotokoll

- Verschiedenste Skripts schlugen nach dem Refactoring zunächst fehl. Der `env`-Befehl funktionierte zunächst nicht mehr. Dies konnte aber mit der neuen Token-Store-Schnittstelle schnell korrigiert werden. Die Skript-Pipeline hat sich dabei als sehr hilfreich erwiesen.
- Das Testskript `standalone-px-upload-test.sh` führt einen Login aus, lädt ein Dokument hoch, wartet etwas länger als fünf Minuten (Gültigkeitsdauer eines Access Tokens) und lädt das Dokument erneut hoch. Da dieser Testfall naturgemäss sehr lange dauert, wird er nicht in die automatische Pipeline integriert, sondern kann bei Bedarf manuell ausgeführt werden.
- Tatsächlich wurde mithilfe dieses Testskripts ein Fehler erkannt: Bei erneuten Versuch eines Requests wurde zwar ein neuer Access Token vom IDP geholt, der neue Request wurde jedoch noch mit dem alten Access Token erstellt, was naturgemäss fehlschlägt. Nach der entsprechenden Korrektur lief der Test dann erfolgreich durch.
- Das Skript `standalone.sh` bietet Hilfestellungen für solche Standalone-Testskripts, indem etwa der Kompilierungsschritt und das Aufräumen nach dem Test vorgegeben wird.
- Für den `get`-Befehl wurde ein testgetriebenes Vorgehen gewählt: `standalone-px-get-test.sh` wurde zuerst als Skript erstellt, das wie geplant scheiterte. Nachdem der `get`-Befehl auch mit Token Refresh arbeitete, funktionierte das Skript anschliessend.

Story 8: Login für Agent API

Als Benutzer möchte ich mich mit als Agent einloggen können, um anderen Benutzern Dokumente einliefern zu können.

Akzeptanzkriterien:

1. Das Login für Agents soll mit einem anderen Subcommand als `login` funktionieren (Vorschlag: `agent-login`).
2. Die Tokens sollen nach der gleichen Logik sicher bzw. unsicher verwahrt werden wie diejenige für die User API.
3. Die Agent Tokens sollen unabhängig von den User Tokens gespeichert werden, d.h. auf einer Umgebung kann gleichzeitig ein User Token und ein Agent Token abgespeichert werden.
4. Der `env`-Befehl soll keine Agent Tokens berücksichtigen: Ein `agent-login` ändert die Standardumgebung nicht.

Notizen

- Da `login` und `agent-login` die gleichen Flags verwenden, wurde das Parsen der Befehlszeile refactored. Beide Funktionen (`login` und `agentLogin`) können nun die gleiche Logik für das Ermitteln der Kommandozeilenoptionen verwenden.
- Beim Login für die Agent API werden eine Client ID und ein Client Secret verwendet, bei User Login eine Client ID, ein Benutzername und ein Passwort. Beim Login-Request unterscheiden sich nur die beiden Payloads, welche mit den Strukturen `Credentials` und `AgentCredentials` umgesetzt werden. Die gemeinsame Methode `ToFormDataURLEncoded` ist in einem Interface definiert, womit der Payload für die beiden Logins abstrahiert werden kann.
- Da neu pro Umgebung mehrere Tokens (Agent und User) abgelegt werden können, muss die Datenstruktur hinter dem Token Store angepasst werden. Bestand der Key vormals nur aus dem Namen der Umgebung, ist es neu eine Kombination aus dem Typ des Tokens (`"user"`, `"agent"`) und der Umgebung. Diese Anpassung erfordert ein Refactoring. Gerade die der `logout`-Befehl manipulierte die zugrundeliegende Map noch selber. Neu wird dies über eine Methode `RemoveToken` gemacht. Ein ergänzender `agent-logout`-Befehl wird zudem benötigt.
- Die Keys im Secret Token Store müssen ebenfalls um den `TokenType` erweitert werden, um zwischen `user-` und `agent`-Tokens unterscheiden zu können.

Testprotokoll

- Zunächst mussten die entsprechenden Credentials (Client ID und Client Secret) auf GitLab hinterlegt werden, um das Login damit testen zu können.
- Aufgrund des Refactorings sind zunächst Unit Tests und Skrittests fehlgeschlagen. Das Zeichen `:` als Key-Separator zwischen Token Type (`agent`, `user`) und Umgebung funktioniert nicht mit dem Utility `jq` zusammen, das zur Extraktion der JSON-Datenstrukturen in der Test-Pipeline dient. Darum wurde es durch einen Underscore _ ersetzt.
- Der Skrittests `ci-px-agent-login-logout-test.sh` funktioniert analog zum Testfall `ci-px-login-logout-test.sh`, nur dass er die Befehle `agent-login` und `agent-logout` statt `login` bzw. `logout` verwendet.

Story 9: Verbesserung der Hilfe-Funktion

Als Benutzer möchte ich eine ausführliche Hilfefunktion für `px` als Ganzes wie auch für die einzelnen Subcommands haben.

Akzeptanzkriterien:

1. Es muss eine generische Hilfefunktion `px help` geben.

2. Es muss für jeden Subcommand eine Hilfefunktion `px help [subcommand]` oder `px [subcommand] -h` geben.

Für zukünftige User Stories ist die Hilfefunktion entsprechend nachzuführen.

Notizen

- Das von der Go Standard Library zur Verfügung gestellte `flag`-Modul stellt mit dem Flag `-h` eine Hilfefunktion zur Verfügung, die mit der Syntax 'px [subcommand] -h aufgerufen werden kann. Diese Funktion ist sinnvoll zum Verständnis der Flags, jedoch ungenügend.
- Der bereits existierende Befehl `px help` (ohne Parameter) zeigt einen Überblick über alle Subcommands an. Diese Funktion ist sinnvoll und soll beibehalten werden. Zusätzlich soll es für jeden Subcommand einen eine ausführliche Hilfestellung geben, die mittels `px help [subcommand]` aufgerufen wird. `px [subcommand] -h` ist weiterhin für die Erläuterung der Flags zuständig.
- Die derzeitige `main()`-Funktion prüft den eingegebenen Subcommand mittels `switch/case`-Kontrollstruktur. Da die meisten Subcommand-Funktionen der gleichen Signatur folgen – den `TokenStore` erwarten und nichts als einen `error` zurückgeben, können die Beziehungen zwischen eingegebenem Befehl (`"agent-login"`) und der aufzurufenden Funktion (`"agentLogin"`) mit einer Map modelliert werden. Als Key wird der Befehlsname verwendet, als Value eine Struktur bestehend aus der auszuführenden Funktion – und einer Funktion, die einen Hilfestring zurückgibt.
- Die wenigen Befehle, deren Funktion eine andere Signatur haben, werden weiterhin per `switch/case` abgehandelt. Sie sind trotzdem in der Map abgelegt, wobei die Funktion den Wert `nil` hat. Einen Hilfestring-Funktion enthalten die Einträge in der Map dennoch für jeden Befehl.
- Die Hilfetexte werden als öffentliche Funktionen im Untermodul `help` abgelegt. Für statische Hilfetexte werden mehrzeilige Strings verwendet. Die Hilfe zur Login-Funktion ist etwa per `help.Login()` abrufbar. Andere Texte werden dynamisch generiert, z.B. um alle verfügbaren Umgebugen aufzulisten.
- Jeder Hilfetext enthält neben einer Erklärung des Befehls auch beispielhafte Aufrufe und Verweise auf die Hilfsfunktion zu den jeweiligen Flags, sowie Hinweise auf andere Befehle.

Testprotokoll

- Das Testskript `ci-px-help.sh` ruft die Hilfefunktion auf und prüft, ob dies fehlerfrei abläuft. Das Skript wurde erweitert, sodass es über alle verfügbaren Befehle iteriert, und für jeden dieser Befehle die Hilfefunktion aufruft. Es wird geprüft, ob dies

erstens fehlerfrei passiert, und ob der dabei zurückgelieferte Text zweitens kein leerer String ist.
- Nach zahlreichen manuellen Tests wurden die Hilfetexte verbessert.

Story 10: Vollzugsmeldungen mit `-v`/`-verbose`-Flag

Als Benutzer möchte ich Vollzugsmeldungen aktivieren können, damit ich sehen kann, ob ein Vorgang erfolgreich war.

Akzeptanzkriterien:

1. Es sollen für alle bestehenden Befehle entsprechende Meldungen erstellt werden, wo ein Erfolg nicht schon anderweitig dem Benutzer gemeldet wird.
2. Die Vollzugsmeldungen sollen mit dem Flag `-v` bzw. `-verbose` aktiviert werden.

Notizen

- Von den bestehenden Befehlen produzieren folgende bereits eine Ausgabe im Erfolgsfall:
  - `px help`: Es wird die gewünschte Hilfefunktion angezeigt.
  - `px upload`: Es wird das serverseitig generierte Metadaten-Objekt in seiner JSON-Repräsentation ausgegeben.
  - `px get`: Es wird die geladene Ressource in ihrer JSON-Representation angezeigt.
  - `px env` (ohne Angabe der Umgebung): Es wird die Standardumgebung angezeigt
- Bei den folgenden Befehlen wird im Erfolgsfall keine Ausgabe erzeugt:
  - `px login`
  - `px logout`
  - `px agent-login`
  - `px agent-logout`
  - `px env` (mit Angabe einer Umgebung)
- Die letztgenannten Befehle wurden jeweils mit den Flags `-v` und `-verbose` ausgestattet, die eine kurze Vollzugsmeldung mit relevanten Kontextinformatioen (Benutzername und Umgebung beim Login, Umgebung bei Setzen der Umgebung mit `px env`).

Testprotokoll

- Folgende Testskripts wurden erweitert, indem jeweils ein Aufruf mit dem `-v`/`-verbose`-Flag vollzogen und auf eine entsprechende Ausgabe überprüft wird:
  - `ci-px-env-test.sh`: für `env`
  - `ci-px-login-logout-test.sh`: für `login` und `logout`
- Für den Negativtest wurde jeweils das Flag `-v`/`-verbose` beim Aufruf im Testskript weggelassen. Nachdem das Flag hinzugefügt wurde, liefen die Tests durch.

Story 11: Verbesserung der Quellcodedokumentation

Als Entwickler möchte ich Dokumentationskommentare zu allen exportierten Elementen (Datentypen, Funktionen/Methoden usw.) haben, damit die Schnittstellen besser verständlich und anderen Entwicklern einfacher zu erläutern sind.

Akzeptanzkriterien:

1. Die Kommentare sollen den den Best Practices entsprechen.
2. Das Werkzeug `go lint` soll über die ganze Codebasis von `px` keine Beanstandungen im Bezug auf undokumentierte, exporte Elemente mehr machen.
3. Dieser Zustand ist auch in Zukunft am Ende eines jeden Sprints herzustellen. Die Aufwände für das Erstellen der entsprechenden Kommentare fliesst jeweils in die User Story ein, die neue exportierte (d.h. zu kommentierende) Elemente zur Folge hat.

Notizen

- Das `Makefile` wurde um das Target `lint` erweitert, das den Go-Linter für das ganze Projekt aufruft.
- Um das Package `px` zu dokumentieren, wurde eine neue Datei `px.go` im Root-Verzeichnis des Projekts erstellt, die nur eine kommentierte `package`-Deklaration enthält.
- Bei Go wird jedes gross geschriebene Symbol exportiert, d.h. anderen Packages zur Verfügung gestellt, was einen Dokumentationskommentar erfordert. Für jede Beanstandung vom Go-Linter soll nicht einfach blind ein solcher Kommentar erstellt, sondern geprüft werden, ob das Feld wirklich exportiert werden muss. Eine kleinere externe Schnittstelle erfordert weniger Dokumentation.
- Die `Command`-Datenstruktur und ihre Eigenschaften im `main`-Package werden mittels Kleinschreibung nicht länger exportiert.

Testprotokoll

- Der Go-Linter beanstandete zunächst 71 fehlende Kommentare.
- Nach der ersten Session konnten die Beanstandungen auf 36 reduziert werden.
- Nach einer zweiten Session von ungefähr gleicher Länge hab es keine Beanstandungen von `golint` mehr.

Story 12: Aktuelle Version ausgeben

Als Anwender möchte ich einen Befehl zur Verfügung haben, der die Version von `px` ausgibt, damit ich sehen kann, ob ich die aktuelle Version der Software verwende, und diese bei Rückmeldungen verwenden kann.

Akzeptanzkriterien:

1. Die Versionierung soll gemäss *Semantic Versioning* erfolgen.
2. Die Version soll beim Kompilieren von Release-Artefakten automatisch aus dem SCM (`git`) verwendet werden.
3. Die Versionsangabe soll über den Befehl `px version` in der Form `v0.3.1` ausgegeben werden.

Notizen

- Die jeweils aktulle Versionsangabe wird aus dem SCM mittels `git describe --tags` ermittelt.
- Die unititialisierte, exportierte String-Variable `Version` in `cmd/px.go` wird mit dem Parameter `-ldflags="-X main.Version=$(git describe --tags)` mit der jeweils aktuellen Versionsangabe initialisiert.
- Die bestehenden Targets `build/mac/px`, `build/linux/px` und `build/windows/px.exe` werden im `Makefile` mit dem entsprechenden Parameter (als Variable `LDFLAGS`) ausgestattet.
- Das Makefile wurde um ein zusätzliches Target `px` ergänzt, das eine ausführbare Version im aktuellen Arbeitszeichnis (für die jeweilige Plattform) mitsamt Versionsangaben erstellt. Andernfalls müsste zum Testen von `px version` jeweils `go run` mitsamt `-ldflags` aufgerufen werden.

Testprotokoll

- In einem Branch wird die Tag-Nummer jeweils um die Anzahl der Commits und die ersten sieben Stellen des letzten Commit-Hashes ergänzt. Der Tag `v0.2.4-9-gdc83825` sagt etwa aus, dass es sich um Version 0.2.4 mit zusätzlichen neun Commits handelt, und nach dem Präfix `g` die ersten sieben Stellen des letzten Commit-

Hashes (`dc83825`, hexadezimal) stehen. Das ist nützlich für ad-hoc erstellte Zwischenreleases.
- Im master-Branch wird nur die Tag-Nummer verwendet.
- Das Testskript `ci-px-version.sh` überprüft, ob `px version` einen String zurückgibt, welcher dem regulären Ausdruck `^v[0-9]+\.[0-9]+\.[0-9]+(-[0-9]+-g[0-9a-fA-F]{7})?$` entspricht.
- Der Build-Step der CI-Pipeline muss sicherstellen, dass die Versionsangaben ebenfalls mitkompiliert werden.

Story 13: Einliefern von Dokumenten per Agent API

Als Benutzer der Agent API möchte ich ein einzelnes Dokument mitsamt Metadaten einliefern können, um so Testdaten für verschiedene Benutzer erstellen zu können.

Akzeptanzkriterien:

1. Es sollen alle Dateiformate unterstützt werden, welche von der Delivery-Schnittstelle zugelassen sind.
2. Die Metadaten werden als JSON-Datenstruktur aus einer separaten Datei mitgegeben.
3. Der Befehl zur Einlieferung von Dokumenten soll `px deliver` heissen.

Notizen

- Zunächst wurden die Hilfetexte verfasst, womit auch die Flags für den Befehl festgelegt wurden (`-m`/`-meta` für die JSON-Metadaten).
- Der Token-Refresh-Mechanismus für die Agent API war noch nicht implementiert. Dies nachzuholen erforderte verschiedene Erweiterungen am Token Store, etwa das Extrahieren/Abspeichern der `clientID`, da diese bei der Agent API im Gegensatz zur User API, wo sie immer `peax.portal` lautet, sich bei jedem Agent unterscheidet.
- Verschiedene manuelle Tests mit einem gültigen Refresh Token ergaben, dass die Aktualisierung eines Token Pairs für die Agent API mit dem `grant_type` `refresh_token` nicht funktioniert. Da die automatische Token-Aktualisierung kein Akzeptanzkriterium ist, und das Problem serverseitig geprüft werden muss, soll diese Funktionalität vorerst offen bleiben.

Testprotokoll

- Der automatische Token Refresh funktionierte zunächst nicht für die Agent API.

- Der `delivery`-Endpoint gibt nach einer erfolgreichen Einlieferung nicht etwa den HTTP-Status-Code `201 Created` sondern `200 OK` zurück, wodurch der erste erfolgreiche Test als fehlerhaft interpretiert worden ist.
- Beim Einliefern eines Dokuments kommt ein JSON-Payload mit einem `id`-Feld zurück, das eine UUID ist. Eine UUID ist eine Sequenz von hexadezimalen Ziffern, die in Gruppen der Grössen acht, vier, vier, vier und zwölf auftreten; dazwischen steht ein Bindestrich. Das Testskript `ci-px-deliver-test.sh` prüft, ob nach dem Einliefern eines Dokuments mit Metadaten eine UUID dieser Form zurückkommt.
- Beim ersten Ausführen der Testpipeline ist es passiert, dass der Backend-Service `portal-document` nicht mehr verfügbar war, und die Tests somit wohl fälschlicherweise gescheitert sind.
- Der Token-Refresh-Mechanismus für die Agent API wurde mithilfe des Testskripts `standalone-px-deliver-retry-test.sh` getestet, wobei zwischen dem ersten und zweiten Versuch etwas mehr als fünf Minuten gewartet wird.
- Das Testskript `ci-px-help-test.sh` wurde um den Aufruf `px help deliver` erweitert.

Story 14: Generische POST-Schnittstelle

Als Benutzer der User API möchte ich einen beliebigen Endpoint mittels `POST`-Methode ansprechen können, damit ich Ressourcen auf dem PEAX-Portal erstellen kann.

Akzeptanzkriterien:

1. Der angegebene Ressourcenpfad wird automatisch anhand der Umgebungsinformationen zu einer URL ergänzt.
2. Es können beliebige Payloads im JSON-Format angegeben werden.
3. Der Payload soll als separate Datei angegeben werden können.
4. Falls die Anfrage einen Payload zurückliefert, soll dieser auf `stdout` ausgegeben werden.
5. Der Befehl soll `px post` heissen.

Notizen

- Wiederum wurde mit dem Erstellen der Hilfetexte begonnen.
- Der Befehl wurde analog zu `px get` mit automatischem Erneuern der Tokens umgesetzt.
- Bei der Implementierung war kein Refactoring nötig. Die Codebasis scheint somit in einem gut erweiterbaren Zustand zu sein.

Testprotokoll

- Es wurde das Testskript `ci-px-post-test.sh` erstellt, das sich auf der Umgebung `test` einloggt, dort eine neue Organisation erstellt, und sich wieder ausloggt. Es wird mit `jq` geprüft, ob der Response-Payload valides JSON ist.
- Andere Endpoints, z.B. das Erstellen von Tags, wurden manuell getestet. (Ressourcen wie Tags haben eindeutige Namen, d.h. mit dem gleichen Payload kann die Ressource nur einmal erstellt werden, was für wiederholte, automatisierte Test schlecht geeignet ist.)
- Das Testskript `ci-px-help-test.sh` wurde um den Aufruf `px help post` erweitert.

Story 15: Generische PUT-Schnittstelle

Als Benutzer der User API möchte ich einen beliebigen Endpoint mittels `PUT`-Methode ansprechen können, damit ich bestehende Ressourcen auf dem PEAX-Portal ersetzen kann.

Akzeptanzkriterien:

1. Der angegebene Ressourcenpfad wird automatisch anhand der Umgebungsinformationen zu einer URL ergänzt.
2. Es können Payloads verschiedener Formate mitgegeben werden (JSON, PNG usw.).
3. Der Payload soll als separate Datei angegeben werden können.
4. Der Befehl soll `px put` heissen.

Notizen

- Die API von PEAX bietet nur sehr wenige Endpoints an, die `PUT` unterstützen. Das einzige Beispiel, das auf Anhieb ermittelt werden konnte, ist die Aktualisierung des Profilbildes. Da es sich hierbei nicht um JSON-Daten handelt, sondern um ein Bild beliebigen Formats, muss der MIME-Type des Payloads automatisch ermittelt werden. Hierzu wurde im Package `utils` eine Funktion `utils.DetectContentType` entwickelt.
- Zunächst wurde wieder der Hilfetext verfasst.
- Der Befehl ist wie `get` und `post` mit automatischer Token-Erneuerung umgesetzt worden.
- Zusätzlich wurde eine Utility-Funktion `ReadFile` entwickelt, die einen `io.ReadCloser` auf die angegebene Datei sowie den Content-Type zurückgibt. Der `ReadCloser` hat den Vorteil, dass nicht die ganze Datei in den Arbeitsspeicher gelesen werden muss.

Testprotokoll

- Die Funktion `utils.DetectContentType` wird durch einen Unit Test abgedeckt. Hierzu wurde zunächst mit GraphViz ein kleiner Graph erstellt (`graph.dot`), woraus mithilfe von `dot` verschiedenste Formate generiert werden konnten: GIF, JPEG, PDF, PNG und PostScript.
- Der Testfall `TestDetectContentType` arbeitet eine Map ab, welche die Dateipfade der generierten Grafiken zu ihrem manuell ermittelten Mime-Type zuordnet. Für jede Iteration wird geprüft, ob für die jeweilige Datei der passende Mime-Type ermittelt werden kann. Für GIF, JPG, PDF, PNG und PostScript hat das auf Anhieb funktioniert.
- Das Testskript `ci-px-put-test.sh` aktualisiert das Profilbild dreimal: im PNG, im JPEG und schliesslich im GIF-Format.
- Das Testskript `ci-px-help-test.sh` wurde um den Aufruf `px help put` erweitert.

Story 16: Generische PATCH-Schnittstelle

Als Benutzer der User API möchte ich einen beliebigen Endpoint mittels `PATCH`-Methode ansprechen können, damit ich Ressourcen auf dem PEAX-Portal partiell/feingranular aktualisieren kann.

Akzeptanzkriterien:

1. Der angegebene Ressourcenpfad wird automatisch anhand der Umgebungsinformationen zu einer URL ergänzt.
2. Es können JSON-Payloads gemäss RFC6902 mitgegeben werden, wobei der Payload lokal nicht überprüft werden muss.
3. Der Payload soll als separate Datei angegeben werden können.
4. Falls die Anfrage einen Payload zurückliefert, soll dieser auf `stdout` ausgegeben werden.
5. Der Befehl soll `px patch` heissen.

Notizen

- Es wurden wiederum zuerst die Hilfetexte verfasst.
- Da die Command-Funktionen für die generischen HTTP-Funktionen `POST`, `PUT` und `PATCH` die gleichen Flags verwenden (Environment und Payload, inkl. Kurzformen, konnten deren Verarbeitung mit einer Funktion vereinheitlicht werden.
- Weitere Vereinheitlichungen gab es im `requests`-Package; so unterscheiden sich bei `POST`, `PUT` und `PATCH` nur die HTTP-Methoden; bei `PUT` wird jedoch kein Payload zurückgeliefert.

- Der Rest der Implementierung ist analog zu `POST`, ausser dass der Header `Content-Type` nicht einfach `application/json`, sondern `application/json-patch+json` lautet (siehe RFC6902).

Testprotokoll

- Das Testskript `ci-px-patch-test.sh` lädt zuerst ein PDF-Dokument per `px upload` hoch. Die UUID des hochgeladenen Dokuments wird ausgelesen. Anschliessend werden die Metadaten dieses Dokuments mittels `px patch` verändert. Für den Ressourcenzugriff wird die zurückgegebene UUID vom ersten Schritt (`px upload`) verwendet. Die `PATCH`-Operation liefert wiederum Metadaten zurück. Die UUID des modifizierten Dokuments wird wiederum extrahiert und gegen die ursprüngliche UUID auf Gleichheit geprüft. (Natürlich ändert sich die UUID im Fehlerfall nicht; es geht nur darum, den Erfolg der Operation mittels korrekt zurückgeliefertert Metadaten zu überprüfen).
- Der Test schlug zunächst fehl, weil fälschlicherweise der `application/json` statt `application/json-patch+json` als `Content-Type`-Header verwendet worden ist. Nach dieser Korrektur lief er durch.
- Das Testskript `ci-px-help-test.sh` wurde um den Aufruf `px help patch` erweitert.

Story 17: Generische `DELETE`-Schnittstelle

Als Benutzer der User API möchte ich einen beliebigen Endpoint mittels `DELETE`-Methode ansprechen können, damit ich Ressourcen auf dem PEAX-Portal entfernen kann.

Akzeptanzkriterien:

1. Der angegebene Ressourcenpfad wird automatisch anhand der Umgebungsinformationen zu einer URL ergänzt.
2. Es soll *kein* Payload mitgegeben werden können.
3. Der Befehl soll `px delete` heissen.

Notizen

- Da `delete` in Go ein reserviertes Keyword ist, musste auf einen anderen Funktionsnamen für den Befehl (`dlete`) ausgewichen werden.
- Es wurden wiederum zunächst die Hilfetexte verfasst.
- Die Request-Logik konnte mit wenig Aufwand analog zu `px get` umgesetzt werden, wobei hier kein Payload zurück- und ausgegeben wird.

Testprotokoll

- Da Testfälle wiederholbar ausführbar sein müssen, wurde für `DELETE` kein neues Testskript erstellt. Stattdessen werden bestehende Testskripts, die Ressourcen erzeugen, um eine Löschung derselben erweitert.
- Es stellte sich heraus, dass das Profilbild (siehe Testprotokoll zu `px put`) die einzige in den Testskripts erstellte Ressource ist, die auch gelöscht werden kann. Dementsprechend wurde das Testskript `ci-px-put-test.sh` um die Löschung des Profilbildes erweitert und zu `ci-px-put-delete-test.sh` umbenannt.
- Ein erster Aufruf schlug fehl, da beim Authorization-Header der Abstand zwischen `Bearer` und dem Access Token vergessen worden war. Nach dieser Korrektur lief das Skript durch. Das Profilbild verschwand damit aus dem Portal.
- Das Testskript `ci-px-help-test.sh` wurde um den Aufruf `px help delete` erweitert.

Story 18: Rekursives Hochladen von Dokument-Ordnern

Als Benutzer der User API möchte ich einen lokale Ordnstruktur, die Dokumente beinhaltet, mit einem Befehl hochladen können, sodass alle in dieser Ordnerstruktur enthaltenen Dokumente im Upload-Bereich des PEAX-Portals erscheinen.

Akzeptanzkriterien:

1. Es soll der bereits bestehende Befehl `px upload` um ein `-r`/`-recursive`-Flag ergänzt werden, der als Parameter ein Verzeichnis erwartet.
2. Es sollen sämtliche in diesem Ordner (und dessen Unterordner beliebiger Tiefe) enthaltenen Dateien hochgeladen werden.
3. Als Ergebnis soll eine Datenstruktur ausgegeben werden, die über Erfolg und Misserfolg jedes versuchten Uploads informiert. Zu jedem Eintrag soll der Pfad der Datei stehen, für die ein Upload versucht worden ist.
4. Scheitert das Hochladen einer Datei, soll der Eintrag eine entsprechende Fehlermeldung enthalten.
5. Ist das Hochladen einer Datei erfolgreich, soll der Eintrag die dabei generierte UUID enthalten.

Notizen

- Zunächst wurde eine Utility-Funktion `ListRecursively` entwickelt, die rekursiv über die Einträge eines Verzeichnisses iteriert. Diese ist mithilfe der Library-Funktion `filepath.Walk` umgesetzt, welche sich um die rekursive Iteration durch das Verzeichnis kümmert. Diese Funktion erwartet eine Funktion als Argument, welche auf jeden gefundenen Eintrag angewendet wird. Die Funktion `isReadable`, die hierfür entwickelt worden ist, prüft für jeden Eintrag, ob er lesbar ist, und fügt

ihn in diesem Fall in eine Liste ein (`isReadable` ist als Closure implementiert). Für unlesbare Fehler wird ein Fehler zurückgegeben. Tritt im ganzen zu prüfenden Verzeichnisbaum auch nur ein Fehler auf, gibt `ListRecursively` einen Fehler und keine Einträge zurück. Dies, weil der Benutzer gleich zu Beginn einer grösseren Operation über mögliche Fehler informiert werden soll, und nicht nach einem längeren Upload-Vorgang über einzelne fehlende Dateien enttäuscht ist.
- Der `upload`-Befehl erhält ein neues Flag `-r/-recursively`. Ist dieses Flag gesetzt, wird nicht wie für ein einzelnes Dokument `requests.UploadDocument`, sondern eine neue Funktion `requests.UploadRecursively` aufgerufen.
- Es wurden zwei neue Datenstrukturen erstllt: `DocumentUploadResult`, das den Erfolg/Misserfolg für einen einzelnen Upload signalisiert; im Erfolgsfall die erstelle UUID des Dokuments, im Fehlerfall eine Fehlermeldung enthält. `FolderUploadResult` enthält eine Liste von `DocumentUploadResult` (ein Eintrag pro Dokument) und statistische Angaben (Anzahl Versuche, Erfolge, Fehler).
- Die bestehende Upload-Funktion (`requests.UploadDocument`) wurde dahingehend refactored, dass der HTTP-Code in eine Funktion ausgelagert wurde, welche eine HTTP-Response zurückgibt. Für ein einzelnes Dokument kann der Payload dann weiter an den Aufrufer zurückgegeben werden, für eine Reihe von Dokumenten wird der Payload für jedes einzelne Dokument ausgewertet, und schliesslich in einem aggregierten Object (`FolderUploadResult`) an den Aufrufer zurückgegeben.
- Die Funktion `utils.ListRecursively` wurde dahingehend angepasst, dass sie Verzeichniseinträge ignoriert, d.h. nur noch Dateieinträge zurückgibt.
- Die Struktur zum Parsen der Upload-Response enthält das Feld `documentID` (Go-Namenskonvention: `ID`), die Response selber jedoch das Feld `documentId` (`Id` mit kleinem `d`). Aus diesem Grund scheiterte das Mapping der resultierenden UUID. Mithilfe eines entsprechenden Tags `json:"documentId"` konnte dieses Mappng korrigiert werden. Da in Go nur exportierte, d.h. grossgeschriebene Felder für die Serialisierung berücksichtigt werden, musste das Feld weiter in `DocumentID` umbenannt werden. Die UUIDs erschienen schliesslich in der Antwort.
- Zum Schluss wurde noch der Hilfetext für den `upload`-Befehl ergänzt, indem die Option mit dem rekursiven Upload erwähnt und die zurückgelieferte Datenstruktur beschrieben wurde.

Testprotokoll

- Mithilfe des Unit Tests `filesystem_test.TestListDirectory` wird eine rekursive Dateistruktur im Temp-Verzeichnis mit einer gegebenen Tiefe (`depth`), Anzahl Unterordner pro Stufe (`subFolders`) und Anzahl Dateien pro Unterordner (`filesPerFolder`) angelegt. Dies sollte insgesamt `subFolders^depth*filesPerFolder` Einträge ergeben. Beispiel: Mit `depth = 3`, `subFolders = 4` und `filesPerFolder = 5` sollen `4^3*5 = 320` Dateien angelegt werden.
- Die Upload-Funktion für ein einzelnes Dokument funktionierte nach dem Refactoring noch einwandfrei.

- Zum Testen des rekursiven Uploads wurde ein Testordner `docfolder` erstellt, der jeweils Unterordner der Struktur `Taxes/[2015..2019]/[Assets, Deductions, Donations, Insurance, Wage-Slips]/` enthält. Jeder dieser Unterordner enthält wiederum ein Testdokument.
- Beim ersten Test fiel auf, dass für erfolgreich hochgeladenen Dokumente in der erstellten Datenstruktur keine `documentId` vorhanden ist. Ausserdem wurde versucht, Verzeichnisse hochzuladen. Nach den entsprechenden Korrekturen (siehe Notizen) funktionierte alles wie gewünscth.
- Das Testskript `ci-px-login-upload-test.sh` wurde umbenannt zu `ci-px-upload-test.sh`, also ohne das `login`, da die meisten Testskripts ein Login ausführen, ohne dies im Namen zu tragen.
- Auf Basis des Testskripts `ci-px-upload-test.sh` wurde das neue Testskript `ci-px-upload-recursively-test.sh` entwickelt. Dieses lädt die Ordnestruktur `docfolder` (siehe Beschreibung oben) hoch. Aus der resultierenden JSON-Datenstruktur wird das Feld `uploaded` extrahiert, und mit der Anzahl Dateien in `docfolder` verglichen.

Story 19: Fehlerkorrekturen der Bugs 3, 4 und 5

- Bug 3: Als Anwender möchte ich, dass die Tokens beim Logout aus dem Keystore gelöscht werden, damit sie nicht in falsche Hände geraten können.
- Bug 4: Als Anwender möchte ich, dass beim Login der `token_type` automatisch gesetzt wird, damit dieser einfacher ersichtlich ist.
- Bug 5: Als Anwender möchte ich, dass beim Logout die Standardumgebung zurückgesetzt wird, damit ich keine Requestversuche mehr auf eine Umgebung unternehme, für die ich keine Token mehr habe, und bei einem solchen Versuch aussagekräftigere Fehlermeldungen erhalte.

Akzeptanzkriterien:

1. Nach dem Logout von einer Umgebung sind auf dieser keine Tokens mehr im Keystore vorhanden. Sollte dies aufgrund eines Fehlers in der Fremdkomponente `zalando/go-keyring` nicht möglich sein, soll der Token nicht gelöscht, aber wenigstens mit einem leeren oder ungültigen Wert überschrieben werden.
2. Tokens in `~/.px-tokens` sollen stets den korrekten Token-Typ gesetzt haben.
3. Nach dem Logout ist das Feld `default_environment` immer leer oder nicht vorhanden in `~/.px-tokens`.

Notizen

- Bei einem Logout-Vorgang wurde `deleteSecret` gar nie aufgerufen, weil das Token Pair die Option `UseKeystore` als `false` gesetzt hatte. In `~/.px-tokens` war die Angabe jedoch korrekt. Der Grund dafür ist, dass das Token Pair vor dem Löschen aus

dem Keystore geladen wird, wobei die Felder `UseKeystore`, `Environment` und `Type` nicht initialisiert worden sind. So konnte der korrekte Key zum Löschen nicht erstellt werden. Die Löschung schlug fehl, ohne einen Fehler zurückzugeben.
- Die Funktion `loadSafely` wurde dahingehend ergänzt, dass immer alle Informationen in die `TokenPair`-Struktur geschrieben werden. Nach dieser Korrektur funktionierte das Löschen von sicher verwahrten Tokens korrekt, zumindest für die Tokens einer bestimmten Umgebung.
- Beim Löschen aller Tokens mit `px logout -a` werden alle User-Token gelöscht; bei `px agent-logout -a` alle Agent-Token. Da der Token-Typ jedoch in `~/.px-tokens` fehlt, können die zu löschenden Tokens nicht anhand dieser Information gefunden werden. Bug 3 und 4 hängen also zusammen.
- Der Token-Typ fehlte in `~/.px-tokens`, da zu sicher verwahrten Token Pairs nur ein Dummy-Eintrag geschrieben wird, bei welchem das Typ-Feld nicht gesetzt worden ist. Dies konnte bei der Erstellung des Dummy-Eintrages entsprechend korrigiert werden.
- Beim Löschen *aller* Tokens besteht jedoch das Problem, dass das Token Pair nicht anhand einer bestimmten Umgebung gelöscht werden kann. Grund dafür ist, dass die Tokens zwar aufgrund der Umgebung gefunden werden können (der Key im Token Store enthält diese Information), das dazugehörige Feld auf dem Token Pair jedoch nicht serialisiert wird, und daher über mehrere Aufrufe hinweg verloren geht. Die Serialisierung ist hier, aus welchem Grund auch immer, für dieses Feld explizit deaktiviert worden. Neu wird das Feld wieder serialisiert, wodurch die Löschung aller Tokens mit `px logout -a` wieder funktioniert.
- Die Standardumgebung wird bei einem Logout zurückgesetzt, falls das `-e`/`-env`-Flag auf die Standardumgebung lautet, oder wenn sich das Logout auf alle Umgebungen bezieht.

Testprotokoll

- Nach der ersten Korrektur (Token Pair beim Laden aus dem Keystore vollständig aufbauen) funktionierte `px logout` auf eine bestimmte Umgebung, jedoch nicht mit dem `-a`/`-all`-Flag.
- Nach der zweiten Korrektur (erweiterter Dummy-Eintrag, serialisieren/abspeichern er Umgebungsinformation) funktionierte auch das Logout auf allen Umgebungen wieder.
- Nach einem Login auf `dev` und `test` (in dieser Reihenfolge) war die Umgebung `test` als Standard gesetzt. Ein Logout von `dev` setzte die Standardumgebung nicht zurück. Beim Logout von `test` wurde die Standardumgebung dann korrekt zurückgesetzt.
- Dieser Testfall wurde zusätzlich im Testskript `ci-px-env-test.sh` abgebildet. Der Testfall schlug zunächst fehl, da `px` auf mehreren Rechnern entwickelt wird, und die letzten Aktualisierungen zunächst nicht mit `git pull` geholt worden sind, lief danach aber wie gewünscht durch.

- Das Testskript `ci-px-login-logout-test.sh` wurde ebenfalls erweitert, sodass nach einem Login überprüft wird, ob der Token Type und die Umgebung korrekt gesetzt sind, was bei sicher verwahrten Tokens nötig zu deren Löschung ist.

Story 20: Statusangabe bei Upload von Dokument-Ordnern

Als Benutzer möchte ich über den Verlauf des rekursiven Uploadvorgangs informiert werden, damit ich die verbleibende Zeitdauer besser abschätzen kann.

Akzeptanzkriterien:

1. Die Statusmeldung erfolgt nach dem Schema `m/n`, wobei `m` für die Anzahl abgeschlossener Uploadvorgänge und `n` für die Gesamtzahl der Uploadvorgänge, d.h. die Anzahl Dokumente im hochzuladenen Ordner, steht.
2. Ein gescheiterter, d.h. fehlerhafter Vorgang wird wie ein abgeschlossener Uploadvorgang behandelt, d.h. er erhöht die erste Zahl um 1.
3. Die Ausgabe soll nach `stderr` erfolgen.
4. Die Ausgabe soll nur erfolgen, wenn das Flag `-v`/`-verbose` gesetzt worden ist.

Notizen

- Zunächst wurde das Verhalten des `-v`/`-verbose`-Flags für Einzelnuploads definiert: Hier wird eine einfache Statusmeldung ausgegeben, dass das Dokument erfolgreich hochgeladen worden sei.
- Beim rekursiven Upload wird das `-v`/`-verbose`-Flag jedoch an die Funktion `Upload-Recursively` des `requests`-Packages übergeben, die dann die entsprechende Ausgabe nach jeder Iteration vornimmt.

Testprotokoll

- Das Testskript `ci-px-upload-recursively-test.sh` wurde dahingehend erweitert, dass `stderr` in eine Datei `upload.err` umgeleitet, und `px` mit dem Parameter `-v` aufgerufen wird. Die Datei `upload.err` wird anschliessend mit `grep` auf das Muster `^[0-9]+/[0-9]+$` geprüft: Dabei sollen so viele Matches auftauchen, wie Dokumente hochzuladen sind. Der Test funktionierte auf Anhieb.

Story 21: Nebenläufiger Upload von Dokument-Ordnern

Als Benutzer möchte ich, dass beim rekursiven Upload von Dokument-Ordnern mehrere Uploadvorgänge nebenläufig ausgeführt werden, um die Zeitdauer des Uploads so zu verkürzen.

Akzeptanzkriterien:

1. Die Anzahl der nebenläufigen Vorgänge soll auf maximal zehn beschränkt werden.
2. Die Statusangabe und das Reporting am Ende des Vorgangs dürfen durch die Nebenläufigkeit nicht beeinträchtigt werden.
3. Die Sortierreihenfolge im bestehenden Report darf durch die Nebenläufigkeit durcheinandergeraten bzw. ist nicht relevant.

Notizen

- Bei der Umsetzung kommen Goroutinen und ein Channel zum Einsatz. Pro Uploadvorgang wird eine Goroutine (Producer) gestartet, die ihr Ergebnis vom Typ `DocumentUploadResult` an einen Channel desselben Typs sendet. In einer weiteren Goroutine (Consumer) wird in einem Loop solange von diesem Channel gelesen, bis dieser geschlossen wird. Dabei werden die ankommenden Resultate ausgewertet und zur Gesamtstatistik (Anzahl erfolgreiche und gescheiterte Uploads) gezählt. Eine `WaitGroup`, die bei jedem gestarteten Uploadvorgang um eins erhöht und am Ende des Vorgangs um eins gesenkt wird, wartet nach dem Starten der beiden Goroutinen darauf, bis alle Vorgänge abgeschlossen sind, und schliesst anschliessend den Channel, sodass der Loop der Producer-Goroutine zu Ende geht. Die Procuder-Goroutine meldet die gesammelten Statistik wiederum per Channel an die Hauptgoroutine zurück, welches von dieser in eine JSON-Struktur umgewandelt und zurückgegeben wird. Das Prinzip ist in *The Go Programming Language* in Kapitel 8.5 (Seite 234 ff.) beschrieben.
- Der automatische Token-Refresh führte anfänglich zu einer Race-Condition. Da jeder parallelle Uploadvorgang theoretisch einen Token-Refresh zur Folge haben kann, kann der Seiteneffekt (Abspeicherung des Tokens) von mehreren Goroutinen gleichzeitig ausgeführt werden, was zu einem undefinierten Verhalten führt. Das Problem wurde mithilfe eines Semaphores gelöst, der zulässt, dass mehrere Token-Refreshs nebenläufig vonstatten gehen, aber immer nur eine Gouroutine gleichzeitig speichern lässt. Dies könnte zwar zu unnötigen Token-Refreshs führen; würde aber der kritische Bereich weiter gefasst, wäre der ganze Uploadvorgang wiederum serialisiert, womit man die Nebenläufigkeit gleich von Anfang an weglassen könnte. Semaphoren sind in *The Go Programming Language* in Kapitel 9.2 (Seite 262 ff.) beschrieben.

- Um die Anzahl gleichzeitiger Vorgänge – und somit die Anzahl parallel laufender HTTP-Anfragen – auf zehn zu beschränken, wurde wiederum eine Semaphore eingesetzt; dieses mal eine der Grösse zehn. Vor jedem Request wird auf einen Channel mit Bufferlänge zehn ein Dummy-Eintrag geschrieben. Nach dem Request wird wieder ein Dummy-Eintrag vom Channel gelesen. Beim Schreiben wird die Semaphore heruntergezählt, beim Lesen wieder hochgezählt. Ein gutes Beispiel dafür findet sich in *The Go Programming Language* in Kapitel 8.6 (Seite 239 ff.).
- Der Aufwand ist unterschätzt worden. Einerseits wurde die mögliche Race-Condition beim Abspeichern der aktualisierten Token nicht bedacht; andererseits führte das instabile Testsystem zu Verzögerungen. Die eigentliche Implementierung ging aber dank der hervorragenden Concurrency-Features von Go sehr schnell und einfach vonstatten.

Testprotokoll

- Die Testskripts wurden nicht erweitert, da die parallelen Uploadvorgänge nichts an der Funktionalität ändern sollen. Ein Performancegewinn in der Abarbeitung der Pipeline war jedoch ein erfreulicher Nebeneffekt.
- Alle manuellen Tests sind mit `go run -race` ausgeführt worden, d.h. mit dem Go Race Detector, der mögliche Race-Conditions aufzeigt. Das potenzielle Problem mit der gleichzeitigen Token-Abspeicherung konnte damit entdeckt und entschärft werden.
- Einige Tests auf der Umgebung `test` schlugen zunächst (mit unterschiedlichem Verhalten) fehl, was auf eine instabile Betriebslage an diesem Abend zurückzuführen war. Nach dem Ausweichen auf die Umgebung `prod` konnten die Testläufe erfolgreich durchgeführt werden.
- Das jüngst eingeführte `-v`/`-verbose`-Flag funktionierte nach der Umsetzung des nebenläufigen Uploads weiterhin tadellos.

Story 22: Automatisches Tagging hochgeladener Ordner

Als Benutzer möchte ich, dass bei einem rekursiven Upload die Dokumente automatisch anhand ihrer Überverzeichnisse getaggt werden, damit ich dies nicht manuell auf dem Portal machen muss.

Akzeptanzkriterien:

1. Das automatische Tagging wird mit dem Flag `-t`/`-tag` aktiviert.
2. Existiert ein Tag noch nicht für den betreffenden Benutzer, wird dieser zunächst erstellt.
3. Die für den rekursiven Upload angegebene Verzeichnisstruktur wird nach untenstehender Regel für das Tagging verwendet.

4. Für jedes Dokument soll im Report nachgewiesen werden, ob das Tagging funktioniert hat, oder eine Fehlermeldung angefügt werden.
5. Können fehlende Tags zu Beginn des Vorgangs nicht erstellt werden, finden keine Uploads statt.

Regel für das Tagging:

Der Benutzer `johndoe` befindet sich im Arbeitsverzeichnis `/home/johndoe`, wo er das Verzeichnis `documents` mittels `px upload -r` mit folgenden absoluten Pfaden rekursiv hochlädt.

```
/home/johndoe/documents/taxes/2018/deductions/insurances.pdf
/home/johndoe/documents/taxes/2018/deductions/donations.pdf
/home/johndoe/documents/taxes/2018/wage-slips/work.pdf
/home/johndoe/documents/taxes/2019/deductions/insurances.pdf
/home/johndoe/documents/taxes/2019/wage-slips/work.pdf
/home/johndoe/documents/taxes/2019/wage-slips/side-job.pdf
```

Für das Tagging soll nur der Teil ab `taxes` verwendet werden, nicht aber der Ordner `documents`, der eigentlich beim Upload angegeben worden ist. Es soll *der Ordner oberhalb des ersten differenzierenden Ordners* für das Tagging verwendet werden. Die Ordner unterhalb von `taxes` sind differenzierend, weil es mehr als einen davon gibt.

Sind zu Beginn des Vorgangs neue Tags zu erfassen, geschieht dies vor der nebenläufigen Ausführung der Hochlade- und Tagvorgänge. Tritt während dem initialen Erstellen der Tags ein Fehler auf, wird der ganze Prozess abgebrochen, ohne dass auch nur ein Dokument hochgeladen worden ist.

Notizen

- Zum Umsetzen der Tagging-Regel (*«der Ordner oberhalb des ersten differenzierenden Ordners»*) muss zunächst anhand einer Reihe von Pfaden der gemeinsame und der jeweils für jeden Pfad distinguirende Teil gefunden werden. Dies wird mithilfe der Funktion `SplitCommonDistinct` (`utils/filesystem.go`) bewerkstelligt.
- Der Befehl `px upload` erhält zusätzlich das Flag `-t`/`-tag`, womit das automatische Tagging aktiviert wird. Es kann nur im Zusammenhang mit `-r`/`-recurisve` verwendet werden, nicht für einzelne Dokumente.
- Anhand der gemeinsamen und distinguirenden Pfadelementen wird eine Liste der benötigten Tags erstellt, indem für die gemeinsamen Pfadsegmente nur das letzte, und für die distinguirenden Pfadsegmente alle ausser das letzte (welches der Dateiname ist) verwendet.

- Vom Portal wird eine Liste der existierenden Tags abgeholt. Hierzu konnte die Funktion `requests.Get` wiederverwendet werden. Die benötigten Tags werden anhand der geholten Tag-Liste auf die noch nicht existierenden Tags reduziert. Diese Liste wird wiederum für die Erstellung neuer Tags verwendet, wozu `requests.Post` wiederverwendet wird. Da diese Funktion eine Payload-Datei erwartet, werden die Payloads zuerst in temporäre Dateien geschrieben, die nach dem Request wieder gelöscht werden können.
- Das Zuordnen der – bestehenden und neu erstellten – Tags wird nach dem Upload eines Dokuments anhand der UUID des Dokuments und der zuvor erstellten Taglist für jedes Dokument vorgenommen. Hierzu konnte die Funktion `requests.Patch` wiederverwendet werden.
- Im Report wird zu jedem Dokument eine Liste erstellter Tags vermerkt. Löst das Tagging einen Fehler aus, wird die Fehlermeldung auch im Report vermerkt. Der Upload wird aber als erfolgreich nachgewiesen.
- Die Funktionen, die für das Tagging verwendet werden (`assignTagsToFile`, `extractUniqueTagNames`, `createTagsIfMissing`, `addTags`) befinden sich im `requests`-Package in einer separaten Datei (`requests/tagging.go`).

Testprotokoll

- Zur Funktion `SplitCommonDistinct` wurde ein Unit Test (`TestSplitCommonDistinct` in `utils/filesystem_test.go`), der für Pfade analog der oben stehenden prüft, ob der Teil bis `taxes` zum gemeinsamen und der Rest zum distinguirenden Teil der Pfadangabe gehört. Da die Testdaten der Bequemlichkeit halber als Unix-Pfade geschrieben worden sind, müssen die Pfadtrennzeichen (`/`) vor dem Test durch `os.PathSeparator` ersetzt werden, damit der Testfall auch auf Windows durchläuft. Die Implementierung verwendet konsequent `os.PathSeparator`.
- Das Erstellen der fehlenden Tags funktionierte auf Anhieb. Für den Ordner `scripts/testdata/docfolder` wurden beim ersten Versuch verschiedenste Tags erstellt, bei einer Wiederholung kein einziger mehr. Nachdem im Portal manuell einige der neu erstellten Tags gelöscht worden sind, konnten diese bei einem erneuten Versuch wieder erstellt werden.
- Mithilfe des Testskripts `ci-px-autotag-test.sh` wird das Verzeichnis `docfolder-small` in `scripts/testdata` rekursiv mit automatischem Tagging hochgeladen. Hat dies funktioniert, wird der Report ausgewertet, indem geprüft wird, ob für jedes Dokument mindestens ein Tag erstellt worden ist.

Manuelle Tests

Da nicht alle Funktionen automatisiert getestet werden können, sollen am Ende eines Sprints manuelle Tests auf allen Plattformen durchgeführt. Es handelt sich um folgende

Liste von Testfällen, die laufend erweitert wird:

#	Beschreibung	seit
1	Login mit Zwei-Faktor-Authentifizierung	Sprint 1
1a	Login mit TOTP-Code	Sprint 1
1b	Login mit SMS-Code	Sprint 1
2	Sichere Verwahrung der Tokens unter Windows	Sprint 1
2a	Login: Eintrag in Windows *Credential Manager*	Sprint 1
2b	Logout: Löschen aus Windows *Credential Manager*	Sprint 1
3	Sichere Verwahrung der Tokens unter macOS	Sprint 1
3a	Login: Eintrag in macOS *Keychain Access*	Sprint 1
3b	Logout: Löschen aus macOS *Keychain Access*	Sprint 1
4	Sichere Verwahrung der Tokens unter Linux	Sprint 1
4a	Login: Eintrag in Linux *Seahorse*	Sprint 1
4b	Logout: Löschen aus Linux *Seahorse*	Sprint 1

Scheitert einer dieser Tests am Ende eines Sprints, wird dies im folgenden Abschnitt «Bugs» entsprechend behandelt.

Zugänge

Für die Tests (2FA und Token Store) werden folgende (produktive) Zugänge verwendet:

- Produktivsystem app.peax.ch:
  - 2FA/TOTP: `patrick.bucher@peax.ch`
  - 2FA/SMS: `patrick.bucher@stud.hslu.ch`
- Testsytem peax-v3-frontend-test.osapps.peax.ch
  - 2FA/TOTP: `paedupeax+totp@gmail.com`
  - 2FA/SMS: `paedupeax+sms@gmail.com`

Testprotokolle

Die manuellen Tests wurden erst seit Ende von Sprint 3 systematisch ausgeführt. Hierbei werden nur gefundene Probleme aufgezeichnet, nicht erfolgreich verlaufene Tests. Letztere werden am Ende des Testdurchlaufs als «durchgeführt» erwähnt.

Sprint 3

1. 2b, 3b und 4b:
    1. Beim Logout werden die sicher verwahrten Tokens nicht aus dem nativen Keystore gelöscht.
    2. Beim Login wird der `token_type` nicht korrekt gesetzt in `~/.px-tokens`.
    3. Beim Logout wird die `default_environment` nicht zurückgesetzt.

Dieses Verhalten konnte auf allen drei Plattformen (Windows, macOS, Linux) nachvollzogen werden.

Alle andere Testfälle konnten erfolgreich durchgeführt werden.

Sprint 4

Sämtliche Tests konnten erfolgreich durchgeführt werden.

Bugs

#	Beschreibung	Status
1	Interaktive Eingabe auf Windows	behoben/workaround
2	Refresh-Mechanismus für Agent	offen (serverseitig)
3	Fehlende Löschung von Tokens aus Keystore	behoben
4	Login setzt `token_type` nicht	behoben
5	Logout setzt `default_environment` nicht zurück	behoben
6	`client_id` fehlt in sicher verwahrten Tokens	behoben

1: Interaktive Eingabe auf Windows funktioniert nicht

Tests auf Windows ergaben, dass es derzeit nicht möglich ist, ein Password sicher (ohne Echo) über die Kommandozeile einzugeben. Recherchen haben ergeben, dass es in diesem Bereich derzeit einen offenen Bug gibt.

Als Workaround wird bis zur Fehlerkorrektur auf die sichere Passworteingabe verzichtet. Mithilfe eines Build Tags konnte dieser Workaround auf Windows eingeschränkt werden, sodass auf macOS und Linux weiterhin die sichere Passworteingabe zum Einsatz kommt.

2: Refresh-Mechanismus funktioniert nicht für Agent API

Es ist derzeit nicht möglich, mit einem Refresh Token eines Agents einen neuen Access Token zu holen. Dieses Problem muss auf dem PEAX Identity Provider näher analysiert werden.

3: Fehlende Löschung von Tokens aus Keystore

Beim Logout werden die sicher verwahrten Tokens nicht aus dem nativen Keystore gelöscht. Zwar verschwindet die Referenz auf die Tokens in `~/.px-tokens`, könnte aber einfach wieder manuell erstellt werden. Dieser Fehler muss somit korrigiert werden. (Falls die Fehlerursache in der Fremdkomponente `go-keyring` ist, könnten die Tokens stattdessen mit einem ungültigen Wert überschrieben werden.)

4: Login setzt `token_type` nicht

Beim Login wird der `token_type` nicht korrekt gesetzt in `~/.px-tokens`. Hierbei handelt es sich eher um ein kosmetisches Problem, da der `token_type` bereits Teil des Keys ist, der auf den jeweiligen Eintrag verweist.

5: Logout setzt `default_environment` nicht zurück

Beim Logout wird die `default_environment` nicht zurückgesetzt. Dies ist ein potenzielles Usability-Problem, da beim nächsten Aufruf eine Standardumgebung angenommen wird, für die es keine Tokens mehr gibt.

6: Fehlende `client_id` bei sicherem Keystore

Die `client_id` wurde bei Laden aus dem nativen Keystore nicht korrekt gesetzt. Dies führte dazu, dass die Token-Aktualisierung für sicher verwahrte Schlüssel nicht funktionierte, sofern die `client_id` nicht aus dem Access Token selber gelesen werden konnte. Der Fehler wurde korrigiert, indem im Falle einer leeren `client_id` der Default-Wert `peax.portal` gesetzt wird. Die Token-Aktualisierung funktionierte anschliessend problemlos.