Backlog

Wirtschaftsprojekt «px: PEAX Command Line Client»

Patrick Bucher

10.11.2019

#	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 i Sprint 3	3
10	Vollzugsmeldungen mit `-v`/`-verbose`-Flag	eingeplant für Sprint 2	1
	Verbesserung der Testabdeckung	offen	
	Verbesserung der Quellcodedokumentation	offen	
	Einliefern von Dokumenten per Agent API	offen	
	Generische `POST`-Schnittstelle	offen	
	Generische `PUT`-Schnittstelle	offen	
	Generische `PATCH`-Schnittstelle	offen	
	Erweiterte Einliefern von Dokumenten per User API	offen	
	Variablen in der Ressourcenangabe	offen	
	Anzeigen der aktiven Logins	offen	
	Einlieferung von Verzeichnissen mit Dokumenten	offen	
	Auflisten von Dokumenten mit Suche/Filterung	offen	
	Ausführung von Befehlen für mehrere Umgebungen	offen	
	Fortschrittsanzeige bei längeren Vorgängen	offen	
	Ausgabe von Tokens	offen	
	Inspektion von Tokens	offen	

Sprints

Sprint 1

- Eingeplant
  - 20 Story Points
  - 6 Stories (1-6)
- Umgesetzt
  - 14 Story Points
  - 4 Stories (1-4)
  - 14.5 Stunden Arbeitsaufwand
- Offen
  - 6 Story Points
  - 2 Stories (5-6)

Sprint 2

- Eingeplant
  - 18 Story Points
  - 6 Stories (5-10)
- Umgesetzt
  - 17 Story Points
  - 5 Stories (5-9)
  - 19.5 Stunden Arbeitsaufwand
- Offen
  - 1 Story Point
  - 1 Story (10)

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.

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.

Testprotokoll

1. automatisiert: `env_test.go`
2. manuell: Login auf verschiedenen Umgebungen (nicht alle, perf/prototype sind heruntergefahren, prod benötigt 2FA) erfolgreich
    1. login vor Registrierung: 401
    2. login nach Registrierung/vor Verifizierung: 403
    3. login nach Verifizierung: 200

Notizen

- table-driven test design (gopl p. 306)
- Ausnahme: `prod`-Umgebung (via PEAX-Subdomains, issuer bei OAuth-Tokens relevant, siehe Mobile App)
- Aufwand: ca. 1 Stunde

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.

Testprotokoll

- zunächst Skript erstellt, das die anderen Skripts ausführt
- ein scheiterndes und ein durchlaufendes Skript erstellt und in ersterem referenziert
- Pipeline für GitLab konfiguriert, Test erfolgreich gescheitert, d.h. scheiterndes Skript hat die Pipeline wie gewünscht zum Abbruch gebracht
- scheiterndes Skript aus der Liste entfernt, Pipelien lief nun durch
- Artefakt kompilieren und dem Skript zur Verfügung stellen
- Minimaltest mit `px help` durchführen
- Login-Testfall (mit Variablen via GitLab) erstellt
- gescheiterte Versuche mit `px login` gaben Status 0 zurück, musste korrigiert werden
- Login-Skript hat schliesslich wie gewünscht funktioniert

Notizen

- viel Zeit aufgrund mangelhafter Bash-Kenntnisse verloren (iterieren über Liste von Skriptdateien)
- TODO: erneut prüfen (Begründung wegen lokaler Ausführbarkeit sinnvoll)! Akzeptanzkriterium 1 über den Haufen geworfen: das Artefakt aus dem `build`-Schritt kann nicht im `script`-Schritt verwendet werden, darum eigene Kompilierung im `script`-Schritt; dafür höhere Performance
- Aufwand schlussendlich überschätzt
- Erweiterung: soll auch lokal ausgeführt werden müssen
- dazu Verschiebung der Build-Logik von gitlab-ci zu Skript
- optionale envvars-Datei (unter .gitignore) für lokale Ausführung
- Bash-Code vereinfacht

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.

Testprotokoll

- beim Refactoring traten immer wieder Build-Fehler auf, die jedoch einfach zu beheben waren
- Logik-Fehler traten keine auf
- HTTP Status 400 nach erstem Versuch, lange am Fehler analysirt
- gemerkt, dass hier kein JSON unterstützt wird, sondern nur form-urlencoded (unnötiges Debugging auf Server)
- schliesslich Erfolgreich
- Login mit SMS und TOTP erfolgreich manuell getestet (automatischer Test nicht praktikabel)
- Login ohne 2FA wird weiterhin via Pipeline getestet

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 war
- das Refactoring wurde schliesslich ausgedehnt; es enstanden neue Submodule `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
- Code neu organisieren war nötig und sinnvoll

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.

Testprotokoll

- der Unittest `env_test` wurde um das `Confidential`-Flag erweitert, wobei nur `prod` so konfiguriert ist
- Tests auf `prod` mit Linux (Tokens schreiben) funktionierte nachdem Key Store korrekt konfiguriert wurde
- der Skript-Test `ci-px-login.sh` wurde erweitert und in `ci-px-login-logout.sh` umbenannt, sodass nun auch das Logout getestet wird
- nach dem Login wird mit `jq` geprüft, ob ein 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

Notizen

- die Umgebungskonfiguration wird um ein Flag (confidential) erweitert, dass 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 verwahren
- die Konfiguration des nativen Key Stores ist für jede Plattform anders und wird 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 wird

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.

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 komortable 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 {paexId} verwenden könnte (`document/api/v3/account/{peaxId}/coll`
  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.

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.

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.

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.

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.
2. Die Vollzugsmeldungen sollen mit dem Flag `-v` bzw. `-verbose` aktiviert werden.