px: PEAX Command Line Client

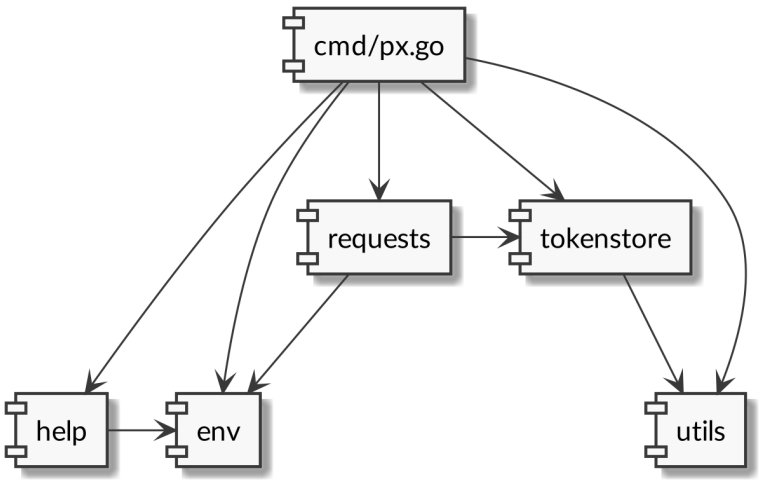
Wirtschaftsprojekt, Herbstsemester 2019

Patrick Bucher

17. Dezember 2019

Abstract

Die Software px ist ein Kommandozeilenprogramm, das den Zugriff auf die RESTful-API von PEAX für Benutzer mit einem technischen Hintergrund vereinfachen soll. PEAX ist ein digitaler Briefkasten, womit Endanwender ihre Post über verschiedene Kanäle empfangen und in einem komfortablen Web-Portal verwalten können. Mit px soll der direkte Zugriff auf die RESTful API, d.h. ohne Verwendung des Webportals, vereinfacht werden, indem die Handhabung von Tokens (Access Token, Refresh Token) abstrahiert wird, und dem Benutzer intuitiv bedienbare sowie generische (d.h. HTTP-nahe) Befehle für das Ansteuern verschiedener Endpoints zur Verfügung gestellt werden.



Inhaltsverzeichnis

1 Problemstellung

Die Problemstellung setzt sich einerseits aus dem gestellten Projektauftrag (siehe Anhang) und andererseits aus der damit implizierten Umgebung (Systeme, Technologien, Benutzer, etc.) zusammen (siehe dazu auch Abschnitt 8.1.1 *Systemkontext*, Seite 56).

1.1 Analyse des Projektauftrags

Der Projektauftrag beschreibt einen Command Line Client für eine RESTful API. In diesem Zusammenhang gibt es Aspekte aus folgenden Bereichen zu analsysieren:

Technologien das Protokoll HTTP und der Authentifizierungsmechanismus OAuth 2.0

Server-Umgebungen die Umgebungen, die eine PEAX API anbieten

Client-Umgebungen die Benutzer, ihr Betriebssystem und ihre Kommandozeile

1.1.1 Endpoints

Eine RESTful API besteht aus einer Reihe sogenannter *Endpoints*, d.h Pfade zu Ressourcen, die abgefragt und/oder manipuliert werden können. Aus Platzgründen soll hier nicht auf einzelne Endpoints eingegangen werden. Beispielhaft erwähnenswert sind aber etwa der Token-Endpoint, bei welchem der Benutzer im Austausch seiner Credentials (Benutzername, Passwort und optionaler zweiter Faktor) ein *Token Pair* holen kann; und der Document-Endpoint, auf welchem Dokumente hochgeladen werden können.

1.1.2 HTTP-Methoden

Ein Endpoint kann über verschiedene HTTP-Methoden angesprochen werden (R. Fielding & Reschke, 2014, Abschnitt 4.3). Im Kontext der PEAX API sind die folgenden Methoden relevant:

- `GET`: Erfragt eine Repräsentation einer bestimmten Ressource; greift nur lesend auf diese zu.
- `HEAD`: Analog zu `GET`, es wird jedoch nur der Header und nicht der Body der Ressource übertragen.
- `POST`: Übermittelt einen Payload zur Speicherung oder Manipulation einer bestehenden Ressource.

- `PUT`: Ersetzt eine bestehende Ressource durch den mitgeschickten Payload.
- `DELETE`: Löscht eine Ressource.
- `OPTIONS`: Beschreibt die Kommunikationsoptionen für eine bestimmte Ressource, wird etwa für CORS Pre-Flight Requests eingesetzt (Mozilla Developer Network, o. J.).
- `PATCH` Führt eine partielle Modifikation auf eine bestimmte Ressource aus (Dusseault & Snell, 2010). Die Modifikation wird in der Form *JavaScript Object Notation (JSON) Patch* durchgeführt (P. Bryan, 2013).

1.1.3 HTTP Status-Codes

Eine Antwort auf eine HTTP-Anfrage enthält jeweils einen Status-Code (R. Fielding & Reschke, 2014, Abschnitt 6). Bei der PEAX API kommen u.a. folgende Status-Codes häufig zum Einsatz:

- `200 OK`: Die Anfrage hat funktioniert.
- `201 Created`: Die Anfrage hat funktioniert, und dabei wurde eine neue Ressource erzeugt.
- `204 No Content`: Die Anfrage konnte ausgeführt werden, liefert aber keinen Inhalt zurück (etwa in einer Suche mit einem Begriff, zu dem keine Ressource gefunden werden kann).
- `204 Partial Content`: Der zurückgelieferte Payload repräsentiert nur einen Teil der gefundenen Informationen. Wird etwa beim Paging eingesetzt.
- `400 Bad Request`: Die Anfrage wurde fehlerhaft gestellt (ungültige oder fehlende Feldwerte).
- `401 Unauthorized`: Der Benutzer ist nicht autorisiert, d.h. nicht eingeloggt im weitesten Sinne (fehlende Authentifikation).
- `403 Forbidden`: Der Benutzer ist zwar eingeloggt, hat aber keine Berechtigung mit der gewählten Methode auf die jeweilige Ressource zuzugreifen (fehlende Autorisierung).

- `404 Not Found`: Die Ressource wurde nicht gefunden; deutet auf eine fehlerhafte URL hin.
- `405 Method Not Allowed`: Die Ressource unterstützt die gewählte Methode nicht.
- `415 Unsupported Media Type`: Das Format des mitgelieferten Payloads wird nicht unterstützt. In der PEAX API sind dies etwa Dokumentformate, die beim Hochladen nicht erlaubt sind (z.B. `.exe`-Dateien).
- `500 Internal Server Error`: Obwohl die Anfrage korrekt formuliert und angenommen worden ist, kommt es bei der Verarbeitung derselben zu einem serverseitigem Fehler.[1]
- `380 Unknown`: Dieser Status ist nicht Teil der HTTP-Spezifikation, wird aber nach einem Login-Versuch verwendet, wenn eine Zwei-Faktor-Authentifizierung (SMS-Code, Time-Based One Time Password) verlangt wird, und ist somit für die vorliegende Arbeit von hoher Relevanz.

1.1.4 OAuth 2.0

Im Hinblick auf das Wirtschaftsprojekt hat sich der Autor dieser Arbeit bereits im Vorsemester im Rahmen des Moduls *Computer Science Hot Topics* (INFKOL) mit dem Thema OAuth 2.0 befasst (Bucher & Christensen, 2019). Detaillierte Informationen zu OAuth 2.0 können diesem Paper und v.a. den dort zitierten Primärquellen entnommen werden.

An dieser Stelle sollen nur die Grundlagen beschrieben werden, die dann im Umsetzungsteil (siehe Abschnitt 5 *Realisierung*, Seite 40) bei Bedarf vertieft werden.

Bei einem Login-Vorgang mit OAuth 2.0 sendet der Benutzer seine Credentials (Benutzername, Passwort, optionaler zweiter Faktor wie SMS-Code) an den Token-Endpoint eines *Identity Providers* (IDP). Stimmen diese Angaben mit den Informationen auf dem IDP überein, erhält der Benutzer ein *Token Pair* bestehend aus *Access Token* und *Refresh Token*. Der Access Token dient zum Zugriff auf eine geschützte Ressource und ist in der Regel kurzlebig. (Bei PEAX läuft er nach fünf Minuten ab.) Mithilfe des Refresh Tokens kann

[1] In der PEAX API kommt es gelegentlich zu solchen Fehlern, die stattdessen mit dem Status `400 Bad Request` und einer aussagekräftigen Fehlermeldung beantwortet werden müssten. Wird z.B. bei der Einlieferung von Dokument-Metadaten zu einer Rechnung eine syntaktisch fehlerhafte IBAN mitgegeben, tritt der Fehler erst bei der internen Verarbeitung, und nicht schon bei der Validierung der Anfrage auf. Hier besteht Handlungsbedarf aufseiten der Backend-Entwicklung.

der Benutzer einen neues Token Pair vom IDP anfordern. Der Refresh Token ist darum langlebiger (30 Minuten bei PEAX).

Werden die Tokens jeweils vor Ablauf dieser Zeitspanne aktualisiert, kann eine Session beliebig lange aufrecht erhalten werden (siehe Abschnitt 5.4 *Retry-Mechanismus*, Seite 45). Aufgrund der Handhabung dieser Tokens (sehr lange Base64-codierte Strings) ist das Ansteuern der PEAX API mit Programmen wie `curl` und POSTMAN sehr umständlich.[2]

Die Anfragen werden vom Client (d.h. Frontend – oder der hier beschriebene Command Line Client) nicht direkt an das Backend geschickt, sondern an einen Proxy-Server. Dieser entscheidet aufgrund der Anfrage, ob die Kombination aus Ressource und Methode einen gültigen Access Token benötigt.[3]

Der Access Token wird auf seine Gültigkeit geprüft; die Anfrage im Erfolgsfall an das Backend weitergeleitet. Die Prüfung, ob der mitgelieferte Access Token für die jeweilige Aktion über den richtigen Scope verfügt, wird teilweise vom Proxy-Server anhand des URL-Schemas (eindeutige PEAX ID des Benutzers im Ressourcenpfad) geprüft. Für andere Aktionen, v.a. im Backoffice-Bereich[4], finden zusätzliche Scope-Prüfungen anhand von Benutzerattributen statt.

1.1.5 Umgebungen

Bei PEAX gibt es verschiedene Umgebungen oder «Stufen», welche den ganzen PEAX-Stack (Datenbank, Backend, Frontend, IDP, Proxy) für einen bestimmten Zweck zur Verfügung stellen:

local Die Entwickler können den PEAX-Stack zum Entwickeln und Testen lokal ausführen. Da diese Umgebung von jedem Entwickler selbständig aufgesetzt und konfiguriert wird, kann sie an dieser Stelle nicht genauer beschrieben werden.[5]

dev Dies ist das Entwicklungssystem, worauf die Entwickler ihre Änderungen deployen, sobald diese vom jeweiligen Feature-Branch in den `develop`-Branch gemerged wurden. Diese Umgebung ist tendenziell sehr aktuell, aber dafür auch instabil.

[2] Eine beispielhafte Analyse ergab, dass ein Access Token 1604 Zeichen lang ist. Refersh Token sind in der Regel etwas kürzer, da sie weniger Informationen zum Gültigkeitsbereich des Tokens enthalten.

[3] Eine `POST`-Anfrage auf den `account`-Endpoint, der zur Registrierung dient, benötigt etwa keinen Access Token, da man einen solchen vor der Registrierung noch nicht haben kann.

[4] Als *Backoffice* wird in diesem Zusammenhang nicht die Organisationseinheit, sondern die Web-Oberfläche zur Administration der PEAX-Anwendung verstanden.

[5] Aufgrund der unterschiedlichen lokalen Konfigurationen soll die lokale Umgebung im Rahmen dieser Arbeit nicht unterstützt werden.

test Auf diese Stufe werden Änderungen übertragen, die auf Stufe `dev` erfolgreich getestet werden konnten. Diese Umgebung wird vom Product Owner zur Abnahme von User Stories verwendet, ist in der Regel eher stabil und repräsentiert nach jedem Sprint einen potenziell releasefähigen Stand.

stage Diese Stufe dient für die Regressionstests. Hier wird nach jedem Sprint der letzte Stand von `test` übertragen. Bei einem Release wird der Stand von hier verwendet. Diese Umgebung ist stabil und jeweils maximal zwei Wochen alt.

prod Von `stage` werden die Änderungen mehrmals pro Jahr (mittelfristiges Ziel: nach jedem Sprint) auf die Produktivumgebung übertragen. Dies ist die einzige Umgebung, auf der produktive Kundendaten abgelegt werden. Datenschutz und Sicherheit spielen auf dieser Umgebung eine besonders hohe Rolle. (Die Konsequenzen daraus sind in Abschnitt **??** **??**, Seite **??** und Abschnitt **??** **??**, Seite **??** beschrieben.)

devpatch Dies ist die Entwicklungsumgebung für den Hotfix-Pfad. Nach einem Release wird der aktuelle Stand von `prod` auf diese Stufe deployed. Bis zum nächsten Release können hier dringende Fehlerkorrekturen vorgenommen werden.

testpatch Dies ist die Testumgebung für den Hotfix-Pfad. Dringende Fehlerkorrekturen werden von `devpatch` auf diese Stufe übernommen und hier abgenommen. Die Änderungen werden von hier aus direkt auf `prod` deployed.

prototype Hierbei handelt es sich um eine Umgebung, die sporadisch für Prototypen und Demos verwendet wird.

perf Diese Umgebung wurde vor dem grossen v3-Release im Frühling 2019 für Last- und Performance-Tests verwendet und ist seither nur sporadisch in Betrieb, etwa zum Ausprobieren neuer Konfigurationen oder Technologien.

1.1.6 Arten von Parametern

Ein HTTP-Request hat verschiedene Parameter: Dies sind einerseits Header-Parameter, wie z.B. `Content-Type`, womit der MIME-Type für den Request-Body festgelegt wird, oder `Accept`, womit dem Server mitgeteilt wird, welcher MIME-Type der Response-Body haben soll (R. T. Fielding et al., 1999).

Auch in der Authentifizierung und Autorisierung spielen Request-Header eine wichtige Rolle, zumal Access Tokens per `Authorization`-Header an den Server übermittelt werden (Hilt, Camarillo & Rosenberg, 2012, Kapitel 7.1).

Andererseits gibt es auch Query-Parameter, welche direkt an die URL angehängt werden. Letztere werden oft für die Navigation im Portal verwendet, zumal bei `GET`-Requests kein Request-Body übermittelt werden kann.

1.1.7 Benutzer

Es gibt verschiedene Gruppen von Benutzern, die px gewinnbringend einsetzen können:

Backend-Entwickler Diese entwickeln, erweitern und korrigieren die RESTful APIs, die das Backend von PEAX ausmachen. Von ihnen kann px einerseits für schnelle Tests und das Erstellen von Testdaten verwendet werden, andererseits kann px auch dabei hilfreich sein, die API (gerade Datenstrukturen) explorativ kennenzulernen.

Frontend-Entwickler Ist die Spezifikation eines Endpoints unvollständig, unklar oder gar fehlerhaft, kann darauf kein funktionierendes Frontend aufsetzen. Hier kann px dabei hilfreich sein, das tatsächliche Verhalten des Backends zu überprüfen, und die Struktur der zurückgelieferten Payloads zu betrachten.

Tester Die manuellen Regressionstests finden direkt auf dem Portal bzw. auf der App statt. Oftmals wäre es hilfreich, Testdaten grösseren Umfangs für einen neu registrierten Benutzer zu erstellen. Mithilfe von px können hierzu einfache Skripts zur Verfügung gestellt werden. (Die Skripts werden tendenziell eher von Entwicklern zur Verfügung gestellt, aber die Tester können diese nach Instruktion selbständig ausführen.)

Projektleiter Sollen grosse Verzeichnisse von Dokumenten eines Firmenkunden auf das PEAX-Portal übertragen werden, ist dies per Web-Interface sehr aufwändig. Hier könnte px zur Automatisierung herangezogen werden.

1.1.8 Betriebssysteme

Zu Beginn des Projekts (September 2019) waren auf den persönlichen Computern der Entwickler macOS und Windows im Einsatz. Mittlerweile (Stand Oktober 2019) wurden alle Windows-Rechner durch Geräte mit macOS ausgetauscht. Mit den Testern und Projektleitern gibt es dennoch einige potenzielle Benutzer (siehe Abschnitt 1.1.7 *Benutzer*, Seite 10), die px immer noch auf Windows einsetzen würden.

Auf zahlreichen virtuellen Maschinen der PEAX-Infrastruktur (etwa für Datenbanken) läuft Linux als Betriebssystem. Hier könnte px für verschiedene Service-Tasks (Monitoring, Alerting) eingesetzt werden.

Es sind somit die Betriebssysteme macOS, Windows und Linux für die Ausführung von px relevant. Was die Architektur betrifft, kommen derzeit nur Intel-Prozessoren mit 64-Bit-Architektur (x86_64) zum Einsatz.

1.1.9 Shells

Verschiedene Betriebssysteme haben verschiedene Shells. Im UNIX-Bereich gibt es auch zahlreiche Shells mit unterschiedlichen Merkmalen, die parallel zueinander installiert werden können.

BASH ist nicht nur die Standard-Shell vieler LINUX-Distributionen, sondern kommt bei Entwicklern auch auf WINDOWS als GIT BASH zum Einsatz. Auf MACOS gehört BASH ebenfalls zum Lieferumfang, wobei die mächtigere ZSH seit MACOS CATALINA standardmässig zum Einsatz kommt. [6] Andere mehr oder weniger populäre UNIX-Shells wie FISH, KSH und TCSH haben unterschiedliche Merkmale, jedoch den POSIX-Standard als kleinsten gemeinsamen Nenner (American National Standards Institute, 1993).

Auf WINDOWS spielen zudem die POWERSHELL sowie cmd.exe eine Rolle.

1.2 Risikoanalyse

Auf den ersten Blick scheint px ein risikoarmes Unterfangen zu sein, handelt es sich doch hierbei um ein Zusatztool für eine bestehende Softwarelandschaft, die Abläufe erleichtern soll. Sämtliche Abläufe können bereits mit den bestehenden Werkzeugen durchgeführt werden, wenn auch vielleicht weniger effizient und weniger komfortabel.

Ein Projekt bringt aber immer das Risiko des Scheiterns mit sich. Solche Risiken beziehen sich auf die Ziele des Projekts, bzw. darauf, dass diese Ziele nicht erreicht werden können, oder das Projekt trotz formal erreichter Ziele das gestellte Problem nicht löst.

Für das vorliegende Projekt könne folgende Risiken identifiziert werden:

Projektrisiken Diese Risiken beziehen sich auf den Erfolg des Projekts als Ganzes:

- Fehlende Adaption: Auch wenn alle Anforderungen formell erfüllt sind, kann es dennoch sein, dass Entwickler und andere potenzielle Benutzer px nicht verwenden wollen. Eine Kombination aus mangelndem Mehrwert, Gewohnheit und Bequemlichkeit könnte der Grund dafür sein. Es gilt also nicht nur ein gutes Werkzeug zu erstellen, es muss auch dessen Mehrwert überzeugend demonstriert werden.

[6] https://support.apple.com/en-us/HT208050

- Mangelnde Abdeckung der API: Werden signifikante Teile der API nicht durch `px` abgedeckt, ist man wiederum zur Verwendung der herkömmlichen Werkzeuge gezwungen. Diese Situation gilt es zu vermeiden, indem eine weite Abdeckung der API erreicht werden soll.

Sicherheitsrisiken Gegenüber `cUrl`, POSTMAN und dergleichen soll `px` eine höhere Sicherheit schaffen. Die folgenden Sicherheitsrisiken können diesem Ziel abträglich sein:

- Token-Verwahrung: Werden die Tokens von `px` nicht angemessen sicher verwahrt, könnten diese von einem Angreifer verwendet werden.
- Payment-Schnittstelle: Über die PEAX API können – ein hinterlegtes Bankkonto vorausgesetzt – Zahlungen durchgeführt werden. In Kombination mit unsicher verwahrten Tokens könnte diese API für betrügerische Überweisungen verwendet werden.

technische Risiken Diese Risiken beziehen sich auf die Implementierung der einzelnen Features von `px`. Sie werden im Rahmen der einzelnen `User Stories` besprochen und behandelt.

2 Stand der Praxis

Dieses Kapitel beschreibt die Ausgangslage vor dem Projektstart. Diese Situation bezieht sich einerseits auf die gängige Praxis beim Ansprechen der PEAX API, andererseits auf State-of-the-Art-Kommandozeilenprogramme, die u.a. auch bei PEAX zum Einsatz kommen.

2.1 Ansprechen der PEAX API

Für das direkte (d.h. nicht durch ein Web-Frontend vermittelte) Ansprechen einer RESTful API haben sich verschiedene Werkzeuge etabliert. Im Folgenden werden verschiedene Werkzeuge besprochen, die sich bei PEAX etabliert haben.

2.1.1 Postman

Die Anwendung POSTMAN[7] erfreut sich bei PEAX-Entwicklern grosser Beliebtheit. POSTMAN wird v.a. als HTTP-Client eingesetzt.

Requests können mit URL, Parametern und Body definiert und manuell in sogenannte Collections gruppiert werden. Diese Gruppierungen werden beispielsweise nach Umgebung (`dev`, `test`) oder nach API-Bereich (Profile, Document) vorgenommen.

Es besteht auch die Möglichkeit, Variablen zu definieren. So kann beispielsweise nach einer erfolgreichen Authentifizierung der zurückgelieferte Access Token abgespeichert werden. Diese Token-Variable kann dann im einen weiteren Request zwecks Autentifikation im *Authentication*-Header mitgegeben werden.

Obwohl POSTMAN einen hohen Komfort bietet und als generischer HTTP-Client eine Abdeckung der gesamten API ermöglicht, gibt es einige Probleme in dessen Handhabung:

Austausch von Collections POSTMAN-Collections können als JSON-Dateien abgespeichert werden. Dieses Format eignet sich zur Speicherung in einem GIT-Repository. Im Gegensatz zu Programmcode oder Konfigurationsdateien sind die einzelnen Änderungen jedoch schwer nachzuvollziehen. Da die Entwickler verschiedene Bedürfnisse haben (API-Abdeckung, Organisation, verwendete Daten), gibt es kein zentrales Repository für POSTMAN-Collections. Stattdessen werden diese Collections als Dateien herumgereicht, was zu einem Wildwuchs führt.

[7] `https://www.getpostman.com/product/api-client`

Speicherung von Credentials Die Zugangsdaten werden zumeist im Klartext abgespeichert und mit den Collections herumgereicht. Hier könnten Variablen Abhilfe schaffen. Fehlendes Wissen über deren Handhabung und Bequemlichkeit führen aber immer wieder dazu, dass wieder Passwörter im Klartext in POSTMAN-Collections auftauchen.

Fehlende Automatisierung Obwohl sich POSTMAN mithilfe einer JAVASCRIPT-Runtime[8] automatisieren liesse, werden Abläufe damit in der Praxis meistens als manuelle Klick-Sequenzen durchgeführt. Der Aufwand, sich in das Scripting-Framework von POSTMAN einzuarbeiten, scheint sich für viele Entwickler offensichtlich nicht zu lohnen. Ein möglicher Grund dafür ist das Lock-In: Skripts können nur innerhalb von POSTMAN ohne Zusatzaufwand ausgeführt werden. Von einem Server oder Container aus, wo kein GUI zur Verfügung steht, sind die Skripts also nicht ohne grossen Zusatzaufwand ausführbar.

2.1.2 cUrl

Bei `curl` handelt es sich um ein Kommandozeilenprogramm zum Ansprechen verschiedenster Protokolle[9], wobei im Kontext von PEAX nur HTTP relevant ist. Verglichen mit POSTMAN wird `curl` von weniger Entwicklern eingesetzt, es kommt aber immer dann zum Einsatz, wenn ein grösserer Ablauf automatisiert werden soll.[10]

`curl` wird oft in Skripts verwendet. Hierzu wird ein High-Level-Skript geschrieben, das die eigentliche Aufgabe übernimmt (Dokument einliefern, Daten abrufen), welches ein Low-Level-Skript für die jeweilige Umgebung aufruft, das sich um das Token-Handling kümmert. Diese Skripts rufen meistens `jq`[11] auf, um relevante Informationen aus den JSON-Payloads zu extrahieren.

Im Gegensatz zu den POSTMAN-Collections sind diese Skripts und deren Änderungen in einem GIT-Kontext einfach zu verstehen. Die Automatisierung ist nicht nur wie bei POSTMAN eine ungenutzte Möglichkeit, sondern wird auch tatsächlich genutzt.

Die Problematik der Credentials, die im Klartext herumgereicht werden, besteht dennoch. Ausserdem werden Ansammlungen solcher Skripts oftmals unübersichtlich und sind nur noch vom ursprünglichen Entwickler versteh- und wartbar.

[8] `https://learning.getpostman.com/docs/postman/scripts/intro_to_scripts`

[9] `https://curl.haxx.se/`

[10] Beispiel: Einem bestimmten Benutzer für Testzwecke 2000 Dokumente per Delivery-API ins Portal stellen, was aufgrund der langen Laufzeit periodische Token-Aktualisierung erfordert.

[11] `https://stedolan.github.io/jq/`

`curl` benötigt zudem einiges an Einarbeitungszeit, um die einzelnen Kommandozeilenparameter zu verstehen – und zusätzliches Verständnis von Kommandozeilenkonzepten (Pipes, Stdin/Stdout) sowie weiterer Kommandozeilenwerkzeuge (`js`, `grep`), um sinnvoll einsetzbar zu sein.

2.1.3 httpie

HTTPie ist eine sehr einstiegsfreundliche Alternative zu `curl`, das sich im Gegensatz dazu auf das Protokoll HTTP konzentriert. Das Absetzen von Multipart-Payloads ist damit wesentlich komfortabler als mit `curl`, was etwa beim Testen der Thumbnailer-Komponente sehr hilfreich war.

Da es sich bei `http` – dem Executable von HTTPie – um ein Python-Skript handelt, ist die Installation einer entsprechenden Laufzeitumgebung vorausgesetzt. Ansonsten hat HTTPie in der Anwendung die gleichen Vor- und Nachteile wie `curl`.

2.2 Kommandozeilenprogramme

Bei Kommandozeilenprogrammen grösseren Umfangs hat sich ein Bedienungsmuster etabliert, bei dem der Programmname als der Hauptbefehl und der erste Parameter als Unterbefehl angegeben wird. Hierzu einige Beispiele:

```
$ git add *.c
$ git commit -m 'added C files'
$ git push

$ docker rm -f my-container
$ docker rmi -f my-image
$ docker run redis

$ go run cmd/px.go
$ go build cmd/px.go -o builds/linux/px
$ go vet
```

Codebeispiel 1: Einie Kommandozeilenbeispiele mit Haupt- und Unterbefehl

Ähnliche Bedienungsmuster findet man auch in `oc` (OpenShift Command Line Client), `pacman` und `brew` (Paketmanager für Arch Linux bzw. macOS), `npm` und `pip` (Paketmanager für Node und Python). Obwohl sich für dieses Bedienungsmuster noch kein allgemein gebräuchlicher Name etabliert hat[12], soll es im Rahmen der vorliegenden Ar-

[12] `https://superuser.com/q/1020583`

beit als *Swiss Army Knife*-Ansatz bezeichnet werden, zumal diese Bezeichnung im Standardwerk zu Go für das go-Tool verwendet wird (Donovan & Kernighan, 2015).

Ein etwas anderer Ansatz wird in Rust für das sehr umfassende Werkzeug `cargo` (u.a. für die Kompilierung und das Ausführen von Tests) verwendet. Hier gibt es zwar auch Unterbefehle (`cargo build`, `cargo test`), diese sind aber nicht zwingend Teil des `cargo`-Binaries, sondern separate Binaries, die im Verzeichnis `/.cargo/bin/` abgelegt sind, und der Namenskonvention `cargo-[subcommand]` folgen (Klabnik & Nichols, 2019, Kapitel 14.5).

Der Ansatz von Rust passt zwar deutlich besser zur Unix-Philosophie – *«Make each program do one thing well.»* (McIlroy Doug, Pinson, E.N, Tague, B.A., 1978, S. 3) – führt aber im Fall von Go aufgrund der statischen Kompilierung zu einer ganzen Reihe grosser Binaries, anstelle nur eines grossen Binaries, was wiederum das Deployment und Setup erschwert.

2.3 Ausgangslage und Vorleistungen

Das Projekt `px` wurde bereits am 11. Juni 2019 auf dem GitLab von PEAX erstellt[13]. Als erstes wurde eine CI-Pipeline bestehend aus den Schritten `build` und `test` eingerichtet. Die Pipeline wurde mittels eines Dummy-Tests überprüft, der einmal erfolgreich durchlaufen und einmal scheitern sollte, um einen Positiv- und einen Negativtest durchführen zu können.

Es wurde eine Hallo-Welt-Programm im `cmd`-Unterverzeichnis (Donovan & Kernighan, 2015, S. 293) erstellt, welches dazu diente, die Kompilierung für verschiedene Plattformen zu testen. Obwohl auch umfassende Go-Programme mittels `go build` kompiliert werden können und keine weitere Build-Konfiguration benötigen, wurde ein `Makefile` erstellt, das ausführbare Programme für verschiedene Plattformen im `build`-Unterverzeichnis erstellt, also z.B. `build/windows/px.exe` für Windows oder `build/linux/px` für Linux.

Das `Makefile` wurde später um ein `release`-Target erweitert, womit die kompilierten Artefakte jeweils in eine Zip-Datei verpackt werden, die den aktuellen Versionstag (z.B. `v0.0.3`[14]) im Dateinamen enthält.

Das Target `coverage` führt die Testfälle durch, misst die Testabdeckung und generiert eine HTML-Ausgabe des getesteten Codes. Rote Zeilen sind nicht durch einen Testfall abgedeckt, grüne Zeilen hingegen schon (Donovan & Kernighan, 2015, Kapitel 11.3).

[13] `https://gitlab.peax.ch/peax3/px`
[14] `https://semver.org/`

Weiter sind bis am 31. Juli 2019 folgende Features implementiert worden:

`px help` zeigt eine einfache Hilfeseite auf der Kommandozeile an.

`px login` führt einen Loginversuch mit den angegebenen Credentials durch. Benutzername und Passwort können entweder als Kommandozeilenparameter oder mittels interaktiver Eingabe (`stdin`) entgegengenommen werden. Im letzteren Fall wird das eingegebene Passwort nicht angezeigt, was mit einem externen SSH-Terminal-Modul[15] erreicht wird. Bei einem erfolgreichen Login-Versuch werden `access_token` und `refresh_token` aus dem Response-Payload gelesen und im `$HOME`-Verzeichnis des jeweiligen Benutzers in eine JSON-Datei namens `.px-tokens` abgespeichert.

`px logout` löscht ein Token Pair für eine bestimmte Umgebung. Pro Umgebung kann es zu jedem Zeitpunkt nur ein aktives «Login», d.h. Token Pair geben. Es besteht auch die Möglichkeit, sämtliche Tokens auf einmal zu löschen. Hierbei wird `$HOME/.px-tokens` nicht gelöscht, sondern nur das Property `tokens` geleert. (Die Datei enthält ein Initialisierungsdatum, das nicht verlorengehen soll.)

`px upload` lädt eine Datei (z.B. PDF) auf das PEAX-Portal hoch. Diese Funktionalität wurde eingebaut, um die Funktionsweise von `px` vor dem Ideation-Gremium zu demonstrieren.[16]

Für die Evaluierung des Prototypen werden zudem die Anzahl Aufrufe und das Datum des letzten Aufrufs von `px` in eine JSON-Datei `$HOME/.px-usage` geschrieben.

Insgesamt wurden ca. 20 Arbeitsstunden in den Prototyp investiert. Ein grosser Teil des Codes kann für die Weiterentwicklung übernommen werden, muss jedoch umstrukturiert werden. So ist zuviel Logik im Hauptmodul `cmd/px.go`, die zwecks Wiederverwendbarkeit in das Library-Modul `px` überführt werden soll. Am Prototypen mangelhaft sind Testabdeckung und Dokumentation auf Quellcodeebene.

[15] `https://godoc.org/golang.org/x/crypto/ssh/terminal#Terminal.ReadPassword`

[16] Die hochgeladene Datei erschien Sekunden später im Web-Portal, was die Anwesenden von der Funktionsweise überzeugte.

3 Ideen und Konzepte

Der PEAX Command Line Client px soll zwei grundlegenden Design-Prinzipien folgen:

1. der Unix-Philosophie, und
2. dem *Swiss Army Knive*-Ansatz, der bereits in Abschnitt 2.2 *Kommandozeilenprogramme*, Seite 15 beschrieben wurde.

3.1 Unix-Philosophie

Die Unix-Philosophie wird oftmals mit den folgenden beiden Grundsätzen wiedergegeben (*The Code*, 2001, 12:51):

1. *Everything is a file.*[17]
2. *When you build something, you write things that are for a single purpose, but that do that purpose well.*[18]

Für den ersten Grundsatz lassen sich schwer Quellen finden. Die Unix-Pioniere Doug McIlroy und Brain Kernighan beschreiben aber den bei der Entwicklung von Unix eingeschlagenen Weg mit einem hierarchischen Dateisystem als revolutionär und als Mitgrund für den Erfolg von Unix und dessen Nachfolger (Kernighan, 2019, Kapitel 4.1, S. 62-62, und Kapitel 9.1, S. 166). Das experimentelle Betriebssystem Plan 9 verfolgte diesen Grundsatz noch konsequenter (Kernighan, 2019, Kapitel 8.4, S. 161).

Der zweite Grundsatz geht auf Doug McIlroys erste Stil-Maxime (McIlroy Doug, Pinson, E.N, Tague, B.A., 1978, S. 3) *«Make each program do one thing well.»*[19] zurück. Die zweite Maxime *«Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. [...]»* ist im Zusammenhang mit Kommandozeilenprogrammen ebenfalls zu beachten. Doug McIlroys *Pipe*-Idee von 1964 und Ken Thompsons Implementierung davon 1972 haben dazu geführt, dass in Unix praktisch jedes Programm mit jedem anderen Programm zusammenarbeiten kann, sofern diese zweite Maxime eingehalten wird.[20] Auf die weiteren Maximen soll an dieser Stelle

[17] Alles ist eine Datei.
[18] Wenn man etwas erstellt, macht man es zu einem einzigen Zweck, den es gut erfüllen soll.
[19] Jedes Programm soll eine Sache gut erfüllen.
[20] Ken Thompson soll den Pipe-Mechanismus nach der ersten Verwendung als «mind-blowing» bezeichnet haben (Kernighan, 2019, Kapitel 4.4, S. 68-69).

nicht eingegangen werden.[21]

Jahre später – und um etliche Erfahrung mit Unix reicher – sammelte Mike Gancarz neun Grundsätze (*tenets*) zur Unix-Philosophie, die hier zusammenfassend wiedergegeben werden (Gancarz, 1995):

1. *Small is beautiful.* Grosse Programme entstehen dann, wenn das zu lösende Problem nicht vollends verstanden und eingegrenzt wird. Bei grossen Programmen wird eine signifikante Menge an Code geschrieben, die nicht das eigentliche Problem löst, sondern andere, nicht-essentielle Sachen macht. Kleine Programme hingegen sind einfach zu verstehen, einfach zu warten, benötigen weniger Systemressourcen – und können einfacher mit anderen Programmen kombiniert werden.

2. *Make each program do one thing well.* Ein Programm soll nur zur Lösung eines Problems entwickelt werden, und nicht damit zusammenhängende oder gar ganz andere Probleme zu lösen versuchen. Dieses eine Problem sollte aber so gut wie möglich gelöst werden. Das Formatieren von Ein- und Ausgaben kann oft von anderen Programmen übernommen werden. Interaktive Programme können oft vermieden werden, indem die Aufbereitung der Eingabedaten mit einem anderen Programm gelöst wird.

3. *Build a prototype as soon as possible.* Eine Idee sollte zunächst anhand eines Prototypen getestet werden, bevor viel Aufwand in eine umfassende Spezifikation und anschliessende Implementierung fliesst, die nicht gebraucht wird. Das Problem ist in dieser frühen Phase oft noch nicht gänzlich verstanden, und ein entsprechendes Konzept würde bloss am Ziel vorbeischiessen. Ein früher Prototyp minimiert das Risiko für unnütze Aufwände und spart somit viel Zeit. Überhaupt entsteht gute Software über iterative Annäherungen an ein Ziel, nicht auf Basis eines einzigen Konzepts.

4. *Choose portability over efficiency.* Ein Programm, das auf vielen Plattformen läuft, ist besser als ein Programm, das die Vorzüge einer einzigen Plattform vollends ausnützt, jedoch nur auf dieser Plattform läuft. Performanceoptimierung für eine einzige Plattform führt oft dazu, dass ein Programm auf anderen Plattformen nur noch schlecht oder gar nicht mehr läuft. Es ist einfacher, eine plattformübergreifende

[21] McIlroys dritte Maxime, man solle Software idealerweise innert Wochen gestalten und erstellen, um sie früh ausprobieren zu können, wird diese Arbeit bereits aufgrund ihres Rahmens und des vorangegangenen Prototypen gerecht.

Codebasis zu warten, als verschiedene plattformabhängige. Da die Hardware ständig schneller wird, lösen sich Performanceprobleme oftmals von selber – sofern ein Programm nicht unnötig aufgeblasen wird.

5. *Store numerical data in flat ASCII files.* Daten müssen früher oder später auf ein anderes System übertragen werden. Bei Textdateien ist dies in der Regel kein Problem. Bei proprietären Binärformaten ist aber oftmals eine Konvertierung erforderlich, die Zeit und Geld kostet. Textdateien können problemlos manuell mit einem Texteditor und automatisiert mit anderen Programmen bearbeitet werden. *Unix* bietet eine Vielzahl solcher Programme.[22]

6. *Use software leverage to your advantage.* Programmierer sollen sich auf das eigentliche Problem konzentrieren, und nicht auf solche, die bereits von anderen Programmierern zufriedenstellend gelöst worden sind. Um dies zu ermöglichen, muss Programmcode zwischen Programmierern ausgetauscht werden, und nicht als streng gehütetes Geheimnis behandelt werden. Bestehende Software – in der Form von Code oder kompilierter Software – kann mit einer gewaltige Hebelwirkung für die eigene Arbeit eingesetzt werden.

7. *Use shell scripts to increase leverage and portability.* Mithilfe von Shell-Skripts kann in wenigen Zeilen Software genutzt werden, die hundertausenden Zeilen – getestetem und von anderen Programmierern gewartetem – C-Code entsprechen. Der Entwicklungszyklus von Shell-Skripts ist schneller als derjenige von C-Programmen, da der Kompilierungsschritt entfällt. Zudem sind Shell-Skripts portabler als Programme, die in C oder in einer anderen Hochsprache geschrieben sind.

8. *Avoid captive user interfaces.* Interaktive Programme haben ihre eigene Interaktionssprache, die sich von der Shell unterscheidet, und daher zunächst für jedes Werkzeug gelernt werden muss. Solche Programme gehen davon aus, dass der Benutzer ein Mensch ist, was die Verwendung via Skripts – und somit die Hebelwirkung von Software – verunmöglicht. Interaktive Programme tendieren dazu, Features aus der Umgebung, von der sie sich abkapseln, in eigener, meist schlechterer Implementierung anzubieten, was die Codebasis aufbläht und zu einem inkonsistenten Verhalten führt.

[22] Heutzutage ist UTF-8 das am weitesten verbreitete Format für Textdateien. Dessen Erfolg geht nicht zuletzt darauf zurück, dass es mit dem Ziel entwickelt worden ist, eine Obermenge von ASCII zu sein, wodurch ASCII komplett vorwärtskompatibel zu UTF-8 ist. UTF-8 wurde von Rob Pike und Ken Thompson entwickelt.

9. *Make every program a filter.* Die meisten Programme nehmen Eingabedaten entgegen, transformieren diese, und produzieren daraus Ausgabedaten. Sie produzieren keine originären Daten, sondern verarbeiten Daten, die zumeist von Menschen prodiziert worden sind. Programme, die als Filter konzipiert sind, werden diesem Umstand gerecht. Auf UNIX werden Eingabedaten über die Standardeingabe ('stdin') entgegengenommen und über die Standardausgabe (`stdout`) wieder ausgegeben. Mithilfe der Pipe können so beliebig lange Filterketten erstellt werden, wovon jedes Programm eine Transformation auf den Datenstrom vornimmt. Fehlermeldungen und andere Informationen, die sich von den Nutzdaten unterscheiden, sollen in die Standardfehlerausgabe (`stderr`) geschrieben werden, was eine gesonderte Verarbeitung solcher Meldungen ermöglicht.

3.1.1 Anwendung auf px

Für px werden die Grundsätze der UNIX-Philosophie folgendermassen angewendet:

1. *Small is beautiful.* px soll nicht jeden Anwendungsfall mit einem einfachen, benutzerfreundlichen Befehl abdecken, sondern mit folgender Doppelstrategie mit möglichst wenig Aufwand zu einem möglichst für alle Benutzergruppen befriedigenden Ergebnis kommen:
   - Entwickler sollen eine möglichst weite hohe und feingranulare Abdeckung der API bekommen, indem px die HTTP-Methoden `GET`, `PUT`, `POST`, `PATCH` und `DELETE` anbietet, und quasi als PEAX-spezifisches `curl` mit transparentem Token-Handling fungiert.
   - Andere Benutzergruppen sollen Befehle bekommen, die sie häufig benötigen und ihnen einen hohen Nutzen bringen, wie z.B. das rekursive Hochladen von Verzeichnissen mit Dokumenten.
2. *Make each program do one thing well.* px soll sich nicht um die Inhalte der Payloads (Dokumente, Metadaten) kümmern und diese validieren oder auswerten, sondern nur sicherstellen, dass die Payloads an den richtigen Ort mit den richtigen Optionen übertragen werden. Beispielsweise soll das Durchsuchen und Auswerten von JSON-Payloads *nicht* in px eingebaut werden, da es hierfür mit `jq` bereits ein sehr mächtiges Tool gibt.
3. *Build a prototype as soon as possible.* Ein minimaler Prototyp wurde bereits im Sommer entwickelt. Auf Basis dieses Prototypen wird nun px im Rahmen dieser Arbeit

weiterentwickelt.

4. *Choose portability over efficiency.* Mit Go wurde eine Programmiersprache gewählt, die das Kompilieren für andere Betriebssysteme und Architekturen *out of the box* unterstützt.[23]

5. *Store numerical data in flat ASCII files.* Die unsicher verwahrten Tokens sollen in einer JSON-Datei namens `.px-tokens` im `HOME`-Verzeichnis des Benutzers abgelegt werden. JWT-Tokens sind numerische Daten im weitesten Sinn, sprich Base64-codiertes JSON. Zwar lässt sich JSON nicht wie eine «flache» ASCII-Datei komfortabel mit Tools wie `grep` und Konsorten bearbeiten, dafür bietet Go sehr komfortable *Marshaling*-Mechanismen um Datenstrukturen aus und zu JSON zu serialisieren (Donovan & Kernighan, 2015, Kapitel 4.5). Wichtig ist, dass zur Speicherung der Tokens keine Binärdateien, sondern Textdateien *im weitesten Sinn* (JSON) zum Einsatz kommen.

6. *Use software leverage to your advantage.* Die Hebelwirkung von *bestehender* Software soll für `px` auf verschiedenen Stufen genutzt werden. Die sehr umfassende und hochwertige Standard Library von Go, gerade das sehr mächtige Package `net/http` unterstützt das Ansprechen einer RESTful HTTP-API. Eine Fremdkomponente von Zalando (`go-keyring`) bietet plattformübergreifende Unterstützung für den nativen Keystore des Betriebssystems. Das sehr mächtige `go`-Tool kommt u.a. zur statischen Quellcodeanalyse (`go vet`), zum Testen (`go test`), Verwalten von Abhängigkeiten (`go mod`) und kompilieren (`go build`) zum Einsatz. In der Entwicklung werden Hilfstools wie `golint` (Quellcodeanalyse), `goimports` (Formatierung von Code und Importieren von Packages) und `gopls` (automatische, Texteditor-unabhängige Code-Vervollständigung).

7. *Use shell scripts to increase leverage and portability.* Die Teststrategie (siehe Abschnitt 4.2.2 *Q2: automatisiert und manuell*, Seite 36) setzt stark auf Shell-Skripts zum Erreichen einer hohen Testabdeckung mit aussagekräftigen Tests. Die Skripts könnten auch dann noch verwendet werden, sollte `px` dereinst mit einer anderen Programmiersprache neu implementiert werden (siehe auch Abschnitt 8.2.4 *Entscheidung Programmiersprache*, Seite 63).

8. *Avoid captive user interfaces.* Grundsätzlich soll `px` interaktiv bedienbar und in Skripten verwendet werden können. So können Benutzername und Passwort per Kom-

[23] Mit Ken Thompson und Rob Pike sind zwei der drei Schöpfer dieser Programmiersprache Unix-Pioniere erster und zweiter Stunde, was man ihr an verschiedensten Stellen anmerkt.

mandozeilenparameter gesetzt und nur interaktiv erfragt werden, falls ersteres unterlassen wird. Bei der Zwei-Faktor-Authentifizierung ist jedoch eine interaktive Eingabe vonnöten, da der SMS-Code zum Zeitpunkt der initialen Login-Anfrage noch nicht bekannt ist.[24]

9. *Make every program a filter.* `px` soll keine unnötigen Ausgaben machen und nur Nutzdaten auf `stdout` ausgeben. Werden Vollzugs- und Erfolgsmeldungen benötigt, sollen diese per speziellem Parameter verlangt und nach `stderr` ausgegeben werden (siehe auch die Diskussion bei Abschnitt 6.1.1 *Sprint 1*, Seite 49).

3.2 Swiss Army Knive

Die «Swiss Army Knive»-Befehlsstruktur (Donovan & Kernighan, 2015, S. 290), wie sie im vorhergehenden Kapitel beschrieben (Abschnitt 2.2 *Kommandozeilenprogramme*, Seite 15) und im Prototyp (Abschnitt 2.3 *Ausgangslage und Vorleistungen*, Seite 16) verwendet worden ist, soll beibehalten werden. Befehle in `px` sollen etwa folgendermassen aussehen:

```
# Login-Vorgang: Tokens vom IDP holen
px login -env test -verbose -user john.doe@acme.org

# Upload-Vorgang: Ein Dokument hochladen
px upload -e test document.pdf

# Logout-Vorgang: lokale Tokens entfernen
px logout -e test
```

Codebeispiel 2: Beispielhafter Befehl mit Command, Subcommand, Flags, Parametern

Die Struktur ist immer dieselbe: `[Befehl] [Unterbefehl] [Flags] [Parameter]`, wobei der `Befehl` im Rahmen dieser Arbeit immer `px` lautet. Die einzelnen Unterbefehle werden in den folgenden Abschnitten erläutert.

Die Flags treten in zweierlei Ausprägung auf: Einerseits als *Switches*, wobei eine Option durch das Vorhandensein eines Flags aktiviert und durch das Fehlen deaktiviert wird. Das `-verbose`-Flag ist ein Beispiel dafür. Andererseits als *Key-Value-Parameter*, wobei das Flag einen zusätzlichen Wert erfordert. Beispiele hierfür sind die Flags `-env` und `-user` im Beispiel oben.

[24] Mit einem One-Time Password würde dies theoretisch funktionieren, da die Codes jeweils vorweg für eine bestimmte Zeitspanne gültig sind. Der Nutzen hiervon ist jedoch äusserst gering, da der Benutzer ja doch am Terminal sitzen muss, um den Code *vor* der Ausführung von `px` einzugeben.

Zu den Flags soll es jeweils eine lange Version (`-verbose`, `-user`, `-env`) und eine zugehörige kurze Version (bestehend aus dem ersten Buchstaben der langen Version) geben (`-v`, `-u`, `-e`). Treten dabei Kollisionen auf (mehrere Flags, die mit dem gleichen Buchstaben beginnen), muss improvisiert werden.

Parameter werden nicht von allen Befehlen erwartet/unterstützt. Beim `upload`-Unterbefehl ist etwa ein Dokument (als relativer Dateipfad) erforderlich. Bei anderen Befehlen sind es Ressourcenpfade.

3.2.1 Befehle für das Token-Handling

Die grundlegende Voraussetzung zur Interaktion mit der PEAX API und die wichtigste Erleichterung durch `px` ist das Token-Handling. Bei einem Login-Vorgang wird ein Token Pair bestehend aus Refresh Token und Access Token vom Identity Provider geholt. Dabei wird zwischen drei Bereichen der API unterschieden: User API, Agent API und Admin API (siehe auch Abschnitt 8.1.1 *Systemkontext*, Seite 56).

Die User API deckt die Funktionen des PEAX-Portals ab. Für das Login wird ein Benutzername (E-Mail-Adresse), ein Passwort und ein optionaler zweiter Faktor verwendet. Letzteres kann ein SMS-Code oder eine Time-based One-time-Password (TOTP) sein.

Die Quelle für diesen zweiten Faktor wird im Portal konfiguriert: Entweder als Mobiltelefonnummer (SMS-Code) oder mittels einer entsprechenden Authenticator-App. Stimmen Benutzername und Passwort im ersten Schritt mit den Angaben auf dem Identity Provider überein, wird interaktiv nach dem zweiten Faktor gefragt, der dann wiederum auf dem Identity Provider überprüft wird.

Bei einem erfolgreichen Login-Vorgang werden die Tokens via HTTP-Response an den Client übertragen, wo sie verwahrt werden können.

Der Login-Vorgang wird mit dem Unterbefehl `login` durchgeführt. Als Flags muss eine Umgebung (`-env`/`-e`), ein Benutzername (`-u`/`-user`) und ein Passwort (`-p`/`-pass`) angegeben werden. Benutzername und Passwort werden interaktiv nachgefragt, sollten diese nicht via Flag spezifiziert worden sein.

Mithilfe der Flags `-s`/`-secure` und `-i`/`-insecure` kann die Umgebungseinstellung bezüglich sicherer Tokenverwahrung übersteuert werden. Die beiden Flags schliessen sich gegenseitig aus. Ist das `-v`/`-verbose`-Flag gesetzt, wird nach einem erfolgreichen Login eine entsprechende Meldung ausgegeben.

Die Tokens können mithilfe des Unterbefehls `logout` wieder entfernt werden. Mit dem Flag `-a`/`-all` werden alle lokalen Tokens entfernt. Mit dem Flag `-e`/`-env` werden die Tokens zu der jeweiligen Umgebung entfernt. Das `-v`/`-verbose`-Flag dient wiederum zur

Ausgabe von Erfolgsmeldungen.

In der Anwendung sehen die Befehle folgendermassen aus:

```
# Login auf Umgebung test mit Benutzername und Passwort
$ px login -e test -u john.doe@acme.org -p TopSecret1337!

# Das gleiche Login mit interaktiver Passworteingabe
$ px login -e test -u john.doe@acme.org
Password: [versteckt]

# Das gleiche Login mit interaktiver Benutzer- und Passworteingabe
$ px login -e test
Username: john.doe@acme.org
Password: [versteckt]

# Login mit Zwei-Faktor-Authentifizierung und verbose-Flag
$ px login -e test -u john.doe@acme.org -p TopSecret1337! -v
SMS Code: 123467
login for user john.doe@acme.org to environment test was successful

# Logout von Umgebung test mit verbose-Flag
$ px logout -e test -v
user logout from environment test was successful

# Logout von allen Umgebungen
$ px logout -a
```

Codebeispiel 3: Anwendung der Befehle für Login und Logout

Die Agent API bietet Partnern und Zulieferern – die *Agents* – die Möglichkeit, beliebigen Benutzern Dokumente und Metadaten ins Portal zu stellen. Sie wird für verschiedene Input-Kanäle von PEAX verwendet.

Im Gegensatz zur User API erfolgt die Authentifizierung nicht mit Benutzername und Passwort, sondern mit einer *Client ID* und einem *Client Secret*.[25] Jeder Agent verfügt über seine eigene Kombinatation von ID und Secret.

Der Agent-API-Nutzers verwendet Client ID und Client Secret somit wie Benutzername und Passwort. Dementsprechend sollen auch die Kommandozeilenparameter entsprechend lauten: `-u`/`-user` für die Client ID und `-p`/`-pass` für das Client Secret.

[25] Bei OAuth 2.0 wird dies als der *OAuth 2.0 Client Credentials Grant* bezeichnet. Bei der User API gibt es die Client ID `peax.portal`, die für alle Benutzer gleich ist.

Die Befehle für das Login und Logout erhalten das Prefix `agent`. In der Anwendung sehen diese Befehle folgendermassen aus:

```
# Agent-Login auf Umgebung test mit Client Secrets
$ px agent-login -u PEAX-Agent-ACME -p 123456-789abc-...

# Agent-Login mit interaktiver Eingabe des Secrets:
$ px agent-login -e test -u PEAX-Agent-ACME
Client Secret: 123456-789abc...

# Logout
$ px agent-logout -e test
```

Codebeispiel 4: Anwendung der Befehle für Agent-Login und -Logout

Für die Admin API wird der gleiche Login-Mechanismus wie für die User API verwendet. Der einzige Unterschied im Gebrauch dieser APIs ist, dass die Benutzer für die Admin API über zusätzliche Berechtigungen verfügen, welche im Access Token entsprechend kodiert sind.

3.2.2 Generische Befehle für Entwickler

Damit Entwickler oftmals an einzelnen Endpoints arbeiten, sollen sie einen möglichst feingranularen Zugriff auf die API erhalten, sprich über die HTTP-Methoden `GET`, `POST`, `PATCH` usw.[26] Die Befehle werden dementsprechend (mit Kleinbuchstaben) bezeichnet.

Befehle für HTTP-Methoden, die über einen Body verfügen (`POST`, `PUT`, `PATCH`), verfügen über den Parameter `-p`/`-payload`, womit eine entsprechende Datei angegeben werden kann.

```
# Abrufen der Anzahl Dokumente eines Benutzers
$ px get document/api/v3/account/455.5462.5012.69/collection

# Hinzufügen einer Organisation (für spätere Postabonnierung)
$ px post -p testdata/organization.json \
    network/api/v3/network/455.5462.5012.69/userorganizationrelationship

# Aktualisierung der Metadaten eines Dokuments
$ px patch -p document-patch.json \
    document/api/v3/document/123456-789abc...
```

[26] px wurde im Ideenstadium als «`curl` *for the PEAX API*» bezeichnet.

```
# Hochladen eines Profilbildes
$ px put -p pic.jpg \
    profile/api/v3/profile/455.5462.5012.69/picture/avatar

# Löschen des Profilbildes
$ px delete \
    profile/api/v3/profile/455.5462.5012.69/picture/avatar
```

Codebeispiel 5: Anwendung der feingranularen HTTP-Befehle

3.2.3 Spezifische Befehle wichtiger Funktionen

Eine der wichtigsten Funktionen von PEAX ist das Einstellen von Dokumenten ins Portal. Dies geschieht einerseits über die Agent API, wo ein Agent einem beliebigen Nutzer ein Dokument (mitsamt Metadaten) ins Portal stellen kann. Andererseits kann ein Benutzer Dokumente für sich selber hochladen.[27]

Ein Agent soll den Benutzern über den Befehl `deliver` Dokumente in den Posteingang stellen können. Der Endpoint ist immer der gleiche. Der Empfänger (und Sender) muss mithilfe von Metadaten (JSON-Format) spezifiziert werden, die mit dem Parameter `-m`/`--meta` mitgegeben werden können. Diese Metadaten sehen beispielsweise folgendermassen aus (`meta.json`):

```
{
    "type": "DOCUMENT",
    "source": "SCANCENTER",
    "title": "Testdokument",
    "senderUid": "CHE-123.456.789",
    "senderName": "px: PEAX Command Line Client",
    "receiverPeaxId": "455.5462.5012.69"
}
```

Codebeispiel 6: Metadaten für die Dokument-Einflieferung

Die `senderUid` ist die MWST-Nummer des Dokument-Absenders. Der Empfänger ist mit der `receiverPeaxId` definiert. Die Felder `type` und `source` sind spezielle Konstanten, die serverseitig geprüft werden, und eine besondere Bedeutung haben. Die Felder `title`

[27] Die anderen Eingangskanäle (App, E-Mail, Postumleitung via Scanning-Center, Sammelcouvert) sind allesamt mit diesen beiden Schnittstellen umgesetzt.

und `senderName` sind Freitextfelder für den Dokumenttitel und den Absendernamen. Eine Vielzahl anderer Angaben (z.B. komplette Zahlungsinformationen für Rechnungen) können auf diese Weise gemacht werden. Die API ist PEAX-intern entsprechend dokumentiert.

Als Parameter erwartet der `deliver`-Befehl den Pfad zu einer Dokumentdatei:

```
# Dokument mit Metadaten einliefern
$ px deliver -e test -m meta.json document.pdf
```

Codebeispiel 7: Dokument-Einlieferung über die Delivery-Schnittstelle

Einem Benutzer der User API soll der `upload`-Befehl zum Hochladen von Dokumenten zur Verfügung stehen. Rekursives Hochladen von Dokumentordnern kann mit dem Flag `-r`/`-recursive` und einer entsprechenden Ordnerangabe als Parameter bewerkstelligt werden. Fehlt dieses Flag, wird als Parameter der Pfad zu einer Dokumentdatei erwartet:

```
# Upload eines einzelnen Dokuments
$ px upload document.pdf

# rekursiver Upload eines Ordners, der Dokumente enhält
$ px upload -r docfolder/
```

Codebeispiel 8: Hochladen von Dateien

3.2.4 Hilfefunktion

Damit sich der Benutzer bei Unklarheiten selber helfen kann, und damit ihm der Einstieg in die Verwendung von `px` erleichtert wird, soll ihm eine Hilfefunktion zur Verfügung stehen. Mit dem Unterbefehl `help` soll er einführende Erläuterungen und einen Überblick über alle zur Verfügung stehenden Befehle erhalten. Gibt er zusätzlich als Parameter den Namen eines Befehls an, soll er Hilfe zu diesem jeweiligen Befehl erhalten. Hilfe zu den einzelnen Flags eines Befehls soll er mit Befehlen der Struktur `px [subcommand] -h` erhalten.

```
# Allgemeine Hilfe (Ausgabe gekürzt)
$ px help

px – the PEAX Command Line Client

This tool is supposed to make dealing with the PEAX API easier [...]
```

```
The following subcommands are supported at the moment:

    px login: log into a PEAX environment (fetch tokens)
    px get: execute a HTTP GET request against a given resource
    [...]

# Befehlsspezifische Hilfe (Ausgabe gekürzt)
$ px help env

px env: get or set the default environment for next commands

This command can be used in two ways: ...

Sample usage:

    # print the current default environment
    px env

    # set prod as the default environment
    px env prod

Run 'px env -h' to get information on the individual flags.

# Informationen über die Flags eines Befehls
$ px env -h
Usage of env:
    -v    print success message to STDERR (shorthand)
    -verbose
          print success message to STDERR
```

Codebeispiel 9: Verwendung der Hilfefunktion

Zu jedem Unterbefehl sollen allgemeine Informationen und Beispielaufrufe beschrieben sein. Bei manchen Befehlen ist es sinnvoll, wenn spezifische Zusatzinformationen ausgegeben werden. Beim `login`-Unterbefehl ist dies etwa eine Liste mit allen zur Verfügung stehenden Umgebungen.

3.2.5 Weitere Befehle

Damit der Benutzer nicht bei jedem Aufruf eine Umgebung mit dem Flag `-e`/`--env` spezifizieren muss, soll es eine Standardumgebung geben. Diese ist nicht konstant, sondern soll sich auf das jeweils letzte Login beziehen: Hat sich der Benutzer beispielsweise zuletzt auf `prod` eingeloggt, soll dies seine Standardumgebung sein.

Da ein Benutzer zu einem gegebenen Zeitpunkt mehrere aktive Logins, d.h. gültige Tokens, auf verschiedenen Umgebungen haben kann, soll er die Standardumgebung wechseln können; nicht nur mit dem `-e`/`--env`-Flag jedes Aufrufs, sondern über mehrere Aufrufe hinweg mithilfe des Unterbefehls `env`. Wird dieser mit einem Parameter (Umgebungsname) aufgerufen, wird die Standardumgebung entsprechend gesetzt, sofern ein Token für diese Umgebung vorhanden ist. Wird der Unterbefehl ohne Parameter aufgerufen, wird die derzeit aktive Standardumgebung ausgegeben.[28]

```
# Login auf Umgebung test
$ px login -e test -u ... -p ...

# Ausgabe der aktuellen Standardumgebung
$ px env
test

# Login auf Umgebung prod
$ px login -e prod -u ... -p ...

# Ausgabe der aktuellen Standardumgebung
$ px env
prod

# Wechseln der Standardumgebung
$ px env test
$ px env
test

# Wechseln auf eine Umgebung, für die kein Login besteht
$ px env prototype
```

[28] Dieses Verhalten ist dem Befehl `oc project` des OpenShift Command Line Clients nachempfunden (`https://docs.openshift.com/container-platform/3.11/cli_reference/basic_cli_operations.html#project`).

```
env: no tokens for environment 'prototype', you must login first
```

Codebeispiel 10: Ausgeben und Wechseln der Standardumgebung

Damit ein Benutzer weiss, welche Version von px er gegenwärtig verwendet, soll ihm der Unterbefehl `version` zur Verfügung stehen, der ihm die semantische Versionsnummer ausgibt. Diese Angabe soll fix ins Programm hineinkompiliert sein.

```
# Ausgeben der Versionsnummer von px
$ px version
v0.3.0
```

Codebeispiel 11: Ausgeben der Versionsnummer

3.3 Token Handling

Wie die Befehle der vorhergehenden Abschnitte (besonders Abschnitt 3.2.2 *Generische Befehle für Entwickler*, Seite 26) gezeigt haben, kann px ähnlich wie `curl` (Abschnitt 2.1.2 *cUrl*, Seite 14) verwendet werden, ohne sich um `Authorization`-Headers, Tokens und dergleichen kümmern zu müssen. Dies, weil px sich um das Abholen (`px login`) und Verwalten der Tokens kümmert.

3.3.1 Token Store

OAuth-Tokens erlauben den Zugriff auf möglicherweise sensitive Informationen. Sie sind ähnlich mächtig, wie ein zeitlich begrenzt gültiges Passwort. Sie sollen darum sicher verwahrt werden können. Andererseits müssen sich Entwickler bei der Fehlersuche die Tokens mit wenig Aufwand anschauen können. Dieser Zielkonflikt kann folgendermassen gelöst werden:

- Es soll zwei Arten der Token-Verwahrung geben: Eine sichere und eine unsichere.
- Für die sichere Token-Verwahrung soll der betriebssystem-eigene Token-Store verwendet werden. Bei macOS ist dies die Anwendung Keychain Access. Bei Windows heisst die entsprechende Anwendung Credential Manager. Auf Linux gibt es verschiedene Varianten, z.B. die Anwendung Seahorse.
- Für die unsichere Token-Verwahrung dient die JSON-Datei `.px-tokens`, die im `HOME`-Verzeichnis des Benutzers abgelegt wird. Diese Art der Verwahrung ist insofern unsicher, dass jedes Programm mit Leseberechtigungen für das entsprechende Ver-

zeichnis auf die Tokens zugreifen kann – was auch für die privaten Dokumente gilt, die ein Benutzer auf seinem Computer ablegt.

- Für Entwicklungs- und Testsysteme ist der komfortable Zugriff auf die Tokens wichtig. Tokens von diesen Systemen sollen in `$HOME/.px-tokens` verwahrt werden.
- Für das Produktivsystem ist die sichere Verwahrung der Tokens wichtig. Sie sollen im nativen Keystore abgelegt werden.
- Mit den Flags `-s`/`-secure` und `-i`/`-insecure` kann das Verhalten vom Benutzer übersteuert werden.

3.3.2 Token Refresh

Der Access Token ist in der Regel kurzlebig. Bei IDP von PEAX beträgt seine Gültigkeitsdauer fünf Minuten ab Ausstellungszeitpunkt. Refresh Tokens sind langlebiger. Sie dienen dazu, ein neues Paar von Token vom IDP zu holen. Bei PEAX beträgt deren Lebensdauer 30 Minuten.

Nach dem Login kann der Benutzer fünf Minuten lang mit dem Access Token Anfragen an die API stellen. Nach diesen fünf Minuten werden seine Anfragen mit dem HTTP-Status `401 Unauthorized` quittiert.

Wird nun mit dem Refresh-Token ein neues Token Pair beim IDP geholt, können nicht für fünf weitere Minuten weitere API-Anfragen an den Server gestellt werden, sondern es kann auch für 30 Minuten ein neues Token Pair geholt werden. Damit kann die Sitzungsdauer beliebig lange sein, sofern die einzelnen Anfragen nie länger als 30 Minuten auseinanderliegen.

Für die Aktualisierung des Token Pairs gibt es zwei Ansätze:

1. proaktive Variante: Die Gültigkeit des Access Tokens wird `vor` jedem Request anhand der Zeitinformationen im Token überprüft. Ist der Token abgelaufen, wird ein neues Token Pair anhand des Refresh Tokens geholt. Ist der Refresh Token, dessen Gültigkeitsdauer ebenfalls vorher geprüft werden soll, auch abgelaufen, wird der Prozess mit einem Fehler abgebrochen.
2. «lazy»-Variante: Ein Request soll mit dem jeweils aktuellen Access Token gestellt werden. Wird er mit dem HTTP-Status `401 Unauthorized` quittiert, soll ein Token Refresh vorgenommen werden. Scheitert der Token Refresh aufgrund eines abgelaufenen Refresh Tokens (wiederum `401 Unauthorized`), soll dies dem Benutzer angezeigt werden.

Die proaktive Variante hat den Vorteil, das keine unnötigen Anfragen an den Server gestellt werden. Die «lazy»-Variante hat den Vorteil, dass clientseitig weniger Logik implementiert werden muss.

Mag die proaktive Variante technisch ausgereifter erscheinen, führt ihr zugrundeliegender Gedanke in eine Sackgasse: Soll `px` *vor* einer Anfrage prüfen, ob diese vom Server akzeptiert werden wird, müsste die serverseitige Validierungslogik im Client abgebildet werden. Um Antworten mit HTTP-Status `400 Bad Request` verhindern zu können, müssten die erlaubten Payload-Schemas und -Restriktionen dem Client bekannt sein. Möchte man zusätzlich Antworten mit HTTP-Status `404 Not Found` verhindern, müsste eine lokale Liste der möglichen Endpoints geführt und gepflegt werden.

Da `px` mitunter ein Entwickler-Werkzeug für Backend-Entwickler ist, und Backend-Entwickler das Verhalten des Servers – und hierzu gehört die Validierungslogik – prüfen möchten, sind lokale Validierungen nur hinderlich. Darum soll die «lazy»-Variante umgesetzt werden, die dem Anwender das bereits zitierte «`curl` *for PEAX*» an die Hand gibt.

4 Methoden

Dieses Kapitel beschreibt nicht das WAS, sondern das WIE: Hier soll es nicht um einzelne Features und deren Implementierung gehen, sondern um das Vorgehen beim Planen, Konzipieren, Umsetzen, Testen, Validieren.

4.1 Vorgehen

Bei PEAX hat sich die agile Softwareentwicklung nach SCRUM durchgesetzt. Da es sich bei px um ein Einzelprojekt handelt, können hier nicht sämtliche Aspekte davon abgebildet werden. Die üblichen SCRUM-Rollen wie PRODUCT OWNER und SCRUM MASTER fallen dabei weg, bzw. bestehen nur in Personalunion des einzigen Entwicklers.

Mit dem Auftraggeber und den Anwendern (Beta-Tester) gibt es aber dennoch Stakeholder, denen der Fortschritt der Arbeit regelmässig berichtet und vorgeführt wird. Von diesen werden auc Rückmeldungen zum Prototyp und zur Projektdokumentation eingefordert, um im weiteren Verlauf des Projekts darauf reagieren zu können.

Die Anwendung soll in mehreren zweiwöchigen Sprints entwickelt werden. Nach jedem Sprint wird der aktuelle Stand an das Entwicklungsteam ausgeliefert. Dabei erhält das Programm einen Versionstag entsprechend der Sprint-Nummer, wozu die Minor-Version erhöht wird (Beispiel: `v0.1.0` ist der Tag nach dem ersten Sprint, `v0.3.0` der Stand nach dem dritten Sprint). Bis zum darauffolgenden Sprint ist nun eine Woche Zeit zum Ausprobieren umd zum Erteilen von Rückmeldungen. Diese fliessen in die Planung zum nächsten Sprint ein, der eine Woche nach dem vorausgegangenen Sprint-Ende startet.

Das Backlog wurde zu Beginn des Projekts nur grob skizziert, wird aber vor Beginn eines jeden Sprints dafür detailliert ausgearbeitet. Neben der der Story-Beschreibung nach der Form *Als [Rolle] möchte ich [Funktion], damit [Nutzen]* werden zu jeder Story mehrere Akzeptanzkriterien festgehalten.

Während dem Umsetzen und Testen werden dann zu jeder Story Umsetzungsnotizen und ein Testprotokoll festgehalten. Teile dieser Notizen fliessen nach dem Sprint in die Dokumentation ein. Ein Story-übergreifender Testplan soll nicht erstellt werden. Die Teststrategie wird im folgenden Unterkapitel beschrieben. Die Systemarchitektur und das Komponentendesign sind im Kapitel zur Realisierung beschrieben (Abschnitt 5 *Realisierung*, Seite 40).

Das Projekt entstammt dem PEAX-Ideation-Prozess, und soll auch in diesem Rahmen validiert werden. Dieser Hintergrund ist im ersten Meilensteinbericht (siehe Abschnitt 8.4 *Weitere Dokumente*, Seite 63) genauer beschrieben.

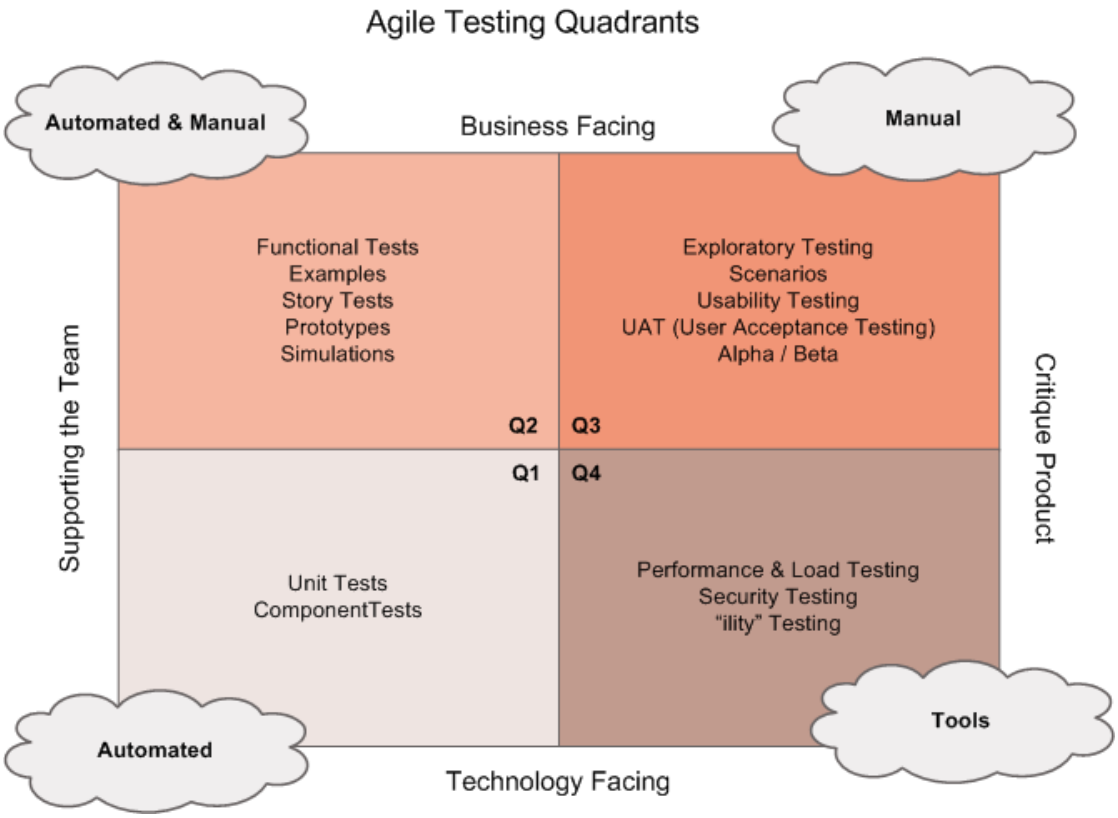


Abbildung 1: *Agile Testing Quadrants* nach Lisa Crispin (`https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/`)

4.2 Teststrategie

Wie im Vorsemester im Modul *Software Testing* eingeübt, sollen die *Agile Test Quadrants* (Abbildung 1, Seite 35) als Grundlage zur Erarbeitung einer Teststrategie dienen (Crispin & Gregory, 2008, S. 242).

Für die vier Quadranten bieten sich für das vorliegende Projekt verschiedene Arten von Tests an. Diese werden im Folgenden für die einzelnen Quadranten beschrieben.

4.2.1 Q1: automatisiert

Im ersten Quadranten geht es um Tests, die vollautomatisch ausgeführt werden können. Diese Tests sollen in einer CI-Umgebung nach jedem Push und vor jedem Merge durchlaufen. Scheitert ein Test, wird der Entwickler notifiziert. Ein Merge-Request soll nicht ausgeführt werden können, wenn Testfälle im Feature-Branch scheitern, sodass die Tests

im Master-Branch immer durchlaufen.

Einzelne Funktionen können durch *Unit Tests* abgedeckt werden. Da die Sichtbarkeitsregeln in Go anders geregelt sind als in Java, und ein Unit Test jeweils zum gleichen Modul wie der zu testende Code gehört, können auch interne Funktionen getestet werden, und nicht nur die vom Modul exportierte Schnittstelle (Donovan & Kernighan, 2015, S. 311). Dies erlaubt ein feingranulareres Testing auf Stufe Unit Test.

Die Komponententests sind dann als Tests einzelner Module und somit als Black-Box-Tests zu verstehen, wobei die exportierte (d.h. öffentliche) Schnittstelle angesprochen wird. Auf Mocking soll hierbei verzichtet werden, da das Schreiben *Test Doubles* (Mocks, Spies, Fakes) Kenntnisse der Implementierung und nicht nur der Schnittstelle erfordert. Ändert sich die Implementierung bei gleichbleibendem (und weiterhin erfülltem) Schnittstellenvertrag, sollte auch ein Komponententest weiterhin funktionieren. Dies ist beim Einsatz von Mocks oft nicht gegeben. Eric Elliot bezeichnet Mocking gar als ein *Code Smell*, das die Struktur des Codes verkompliziere, wo doch *Test Driven Development* dabei helfen solle, Code zu vereinfachen (Elliot, 2019, S. 205),

Ziel der Unit- und Komponententests ist nicht eine möglichst hohe Codeabdeckung, sondern ein optimales Verhältnis von Aufwand und Ertrag: Zeigt es sich, dass für einzelne Tests mit viel Aufwand eine umfassende Umgebung (im weitesten Sinne) aufgebaut werden muss, soll stattdessen geprüft werden, ob der Code nicht besser mittels Tests des zweiten Quadranten getestet werden soll.

4.2.2 Q2: automatisiert und manuell

Im zweiten Quadranten geht es um Tests auf der funktionalen Ebene, die teils automatisch, teils manuell ausgeführt werden.

Da `px` nicht nur interaktiv, sondern auch in Skripten verwendet werden soll, können komplette Workflows ebenfalls vollautomatisiert durchgetestet werden. Ein folgendes Skript könnte etwa folgenden Ablauf beschreiben:

- Einloggen auf einen System mit speziellen Test-Zugangsdaten
- Hochladen mehrerer Dokumente inklusive Metadaten mit Abspeicherung der dabei generierten Dokument-UUIDs
- Tagging der hochgeladenen Dokumente
- Herunterladen der zuvor hochgeladenen Dokumente und Vergleich mit dem ursprünglich hochgeladenen Dokument (etwa per SHA-2-Prüfsumme)

- Abfragen der zuvor mitgeschickten Metadaten und Tags; Prüfung derselben gegenüber den Ausgangsdaten

Ein solcher Test muss zwangsläufig gegen eine laufendes Umgebung durchgeführt werden. (Ähnliche, aber wesentlich einfachere Abläufe werden bereits mit Uptrends durchgeführt, um die Verfügbarkeit der produktiven Umgebung automatisch zu überprüfen.) Hierbei besteht die Gefahr, dass Testfälle aufgrund eines Fehlers in der entsprechenden Umgebung scheitern, und nicht aufgrund der am Code von px vorgenommenen Änderungen. Im schlimmsten Fall müsste ein Merge-Vorgang auf den Master-Branch aufgrund einer nicht funktionierender Umgebung verzögert werden. Eine pragmatische Lösung wäre es, wenn diese Tests gegen die produktive Umgebung ausgeführt würden. Diese Umgebung ist hoch verfügbar, und bei den seltenen Ausfällen derselben könnte auch notfalls mit einem Merge-Vorgang auf den Master-Branch zugewartet werden.[29] Die Zugangsdaten für ein entsprechendes Testkonto können innerhalb der CI-Umgebung als verschlüsselte Variablen und lokal als Umgebungsvariablen abgelegt werden.

Das Schreiben von Skripts ist meistens ein Prozess, dem üblicherweise mehrere manuelle Durchgänge der auszuführenden Arbeitsschritte vorangeht. Skripts werden häufig dann geschrieben, wenn ein manueller Vorgang die Finger stärker beansprucht als den Kopf, und aufgrund der nachgebenden Achtsamkeit die Gefahr für Flüchtigkeitsfehler besteht. So soll es auch im vorliegenden Projekt gehandhabt werden: Wird ein neues Feature eingebaut, d.h. eine neue *User Story* umgesetzt, und sich das Testing desselben als aufwändig herausstellt, soll ein Testskript geschrieben werden, das dann sogleich in die CI-Pipeline eingebaut werden kann. Manuelle Tests sollen so möglichst bald und unkompliziert in automatisierte überführt werden.

4.2.3 Q3: manuell

Nach jedem Sprint erhalten die Entwickler bei PEAX Zugang zum aktuellen Stand der Software mit einem Changelog. Sie haben nun eine Woche Zeit, sich mit den neuen Features vertraut zu machen, und auszuprobieren, ob sich die Software wie gewünscht verhält.

Hier geht es weniger um die Korrektheit gemäss Spezifikation (Akzeptanzkriterien in den User Stories), welche bereits durch Tests in Q1 und Q2 gewährleistet werden sollte,

[29] Der Autor dieser Zeilen ist u.a. auch für die Verfügbarkeit des Produktivsystems zuständig. Ist diese nicht gegeben, werden Entwicklungsarbeiten erfahrungsgemäss unterbrochen, bis das System wieder vollumfänglich verfügbar ist.

sondern um *User Acceptance Tests*. Hiermit wird geprüft, ob das Inkrement die Bedürfnisse der Benutzer erfüllt und ihren Zielen dient. «Another kind of acceptance testing is *user acceptance testing*. Commonly used in Agile environments, user acceptance testing (UAT) checks that the system meets the goals of the users and operates in a manner acceptable to them (Laboon, 2017, S. 85).

Auch soll im Rahmen dieser Tests die *Usability* des Inkrements überprüft werden. Stossen mehrere Tester auf die gleichen Probleme? Wurde ein Feature (z.B. ein Subcommand) schlecht benannt, sodass dessen Semantik unklar ist? Ist die angebotene Hilfe-Funktion zu einem Befehl unklar oder schlecht formuliert?

Die Tests in Q3 sind jeweils in der ersten Wochenhälfte nach einem Sprint durchzuführen, sodass die Rückmeldungen für die Planung des nächsten Sprints, der in der darauffolgenden Woche startet, berücksichtigt werden kann.

4.2.4 Q4: Tools

Für die Qualitätssicherung können verschiedene Werkzeuge zum Einsatz kommen:

Benchmarks Bieten sich bei der Implementierung einer performancekritischen Funktion mehrere Varianten an, ist die bessere Variante mithilfe von Benchmarks zu bestimmen. Die integrierte Benchmark-Funktion des go-Tools[30], die eine Funktion so oft laufen lässt, bis eine statistisch relevante Aussage über deren Performance gemacht werden kann, ist hierzu völlig ausreichend (Donovan & Kernighan, 2015, S. 321).

Profiling Im Profiling geht es darum, die kritischen Stellen im Code im Bezug auf Rechenzeit, Speicherverbrauch und blockierende Operationen zu ermitteln, um Aufgrund dieser Erkenntnis effektive Optimierungen am Code vornehmen zu können (Flaschenhalsoptimierung). Hierzu bietet das go-Tool[31] wiederum sehr mächtige Werkzeuge out-of-the-box an (Donovan & Kernighan, 2015, S. 324).

Quellcodeanalyse Kompilierbarer und korrekter Quellcode ist nicht automatisch auch guter Quellcode im Bezug auf Klarheit, Einfachheit, Eleganz und Best Practices. Beispielsweise sollen nach Möglichkeit keine veralteten und unsicheren APIs verwendet werden. Exportierte Funktionen, d.h. die öffentliche Schnittstelle eines Moduls,

[30] `go test -bench=[pattern]`
[31] `go test -cpuprofile/-memprofile/-blockprofile=[Ausgabedatei]`

muss dokumentiert sein. Hierzu gibt es einerseits das Tool `go vet`[32], das zum Lieferumfang von Go gehört, und potenzielle Fehler im Code meldet. Das Zusatztool `golint`[33] meldet stilistische Unschönheiten im Code.

Da es bei diesen Tools nicht um kategorische Qualitätskriterien (richtig oder falsch), sondern eher um kontinuierliche (schnell/langsam, hoher/tiefer Speicherverbrauch, hohe/tiefer Quellcodequalität) handelt, die einer subjektiven Interpretation bedürfen, sollen diese Tools nicht Teil der CI-Pipeline sein, sondern kontinuierlich (`go vet` und `golint`) und bei konkretem Bedarf (Benchmarking, Profiling) im Entwicklungsprozess eingesetzt werden. Um eine gewisse Quellcodeanalyse gewährleisten zu können, ist die Ausgabe von `go vet` und `golint` am Ende eines jeden Sprints summarisch zu dokumentieren.[34]

[32] Vet examines Go source code and reports suspicious constructs, such as Printf calls whose arguments do not align with the format string.

[33] `https://github.com/golang/lint`

[34] Da die Ausformulierung von Rechtfertigungen oftmals anstrengender ist als die notwendigen Korrekturen am Code vorzunehmen, ist diese Massnahme als Anreiz zu verstehen, die erhaltenen Warnungen zu behandeln statt zu ignorieren.

5 Realisierung

Wie im Anhang (Abschnitt 8.2.4 *Entscheidung Programmiersprache*, Seite 63) beschrieben und begründet, wird px mit Go umgesetzt. Da die Standard Library von Go sehr umfassend ist, die mit dem Package `net/http` über einen mächtigen HTTP-Client (und Server) verfügt, wird nur eine Fremdkomponente (Zugriff auf den Keystore) eingesetzt.

Enorm hilfreich in der Umsetzung war Standardwerk zu Go (Donovan & Kernighan, 2015), das nicht nur die Programmiersprache beschreibt, sondern auch deren idiomatischen Gebrauch.

5.1 Architektur: Package-Übersicht

Go-Code wird in sogenannte Packages aufgeteilt. Da das Projekt in GitLab px heisst, wird das Hauptpackage als px benannt. Im Root-Verzeichnis befindet sich kein Go-Code.[35] Dieser ist in verschiedenen Unterverzeichnissen (`requests`, `tokenstore`, usw.) abgelegt. Die Dateien in den Unterpackages deklarieren ihre Packagezugehörigkeit jeweils mit dem unmittelbaren Überverzeichnis, also beispielsweise nicht `px/tokenstore`, sondern nur `tokenstore`. Bei der Verwendung der Packages hingegen wird der ganze Pfad angegeben: `px/tokenstore`.

Der Code für das ausführbare Programm (`px.go`) befindet sich gemäss Konvention[36] im `cmd`-Unterverzeichnis (Gerardi, 2019, S. 12). Das Package heisst jedoch nicht `cmd`, sondern `main`, und verfügt über eine Funktion namens `main` als Haupteinstiegspunkt. Somit ist px als Library, und `cmd/px.go` als Client dieser Library zu verstehen.[37] Die Projektstruktur ist auf der linken Seite der Abbildung 2, Seite 41 zu sehen. Die einzelnen Packages haben folgende Verantwortlichkeiten:

env Die Umgebungen (Abschnitt 1.1.5 *Umgebungen*, Seite 8) sind in diesem Package aufgelistet. Jede Umgebung verfügt über ein URL-Schema, womit (via Proxy) auf das Backend und direkt auf den IDP zugegriffen werden kann. Verschiedene Funktionen bieten die Möglichkeit, URLs für den Ressourcenzugriff anhand der jeweiligen Umgebung und weiterer Parameter zu erzeugen, was mit dem Unit Test `env_test.go` überprüft wird. Die Standardeinstellung, welcher Keystore (sicher/unsicher) zu verwenden ist, wird statisch pro Umgebung definiert.

[35] `px.go` beinhaltet bloss eine Package-Deklaration mit einem entsprechenden Kommentar.

[36] `https://github.com/golang-standards/project-layout#cmd`

[37] Da sich nur knapp ein Viertel des Programmcodes im Client-Teil befinden, könnte auf Basis der px-Library recht einfach ein alternativer Client umgesetzt werden.

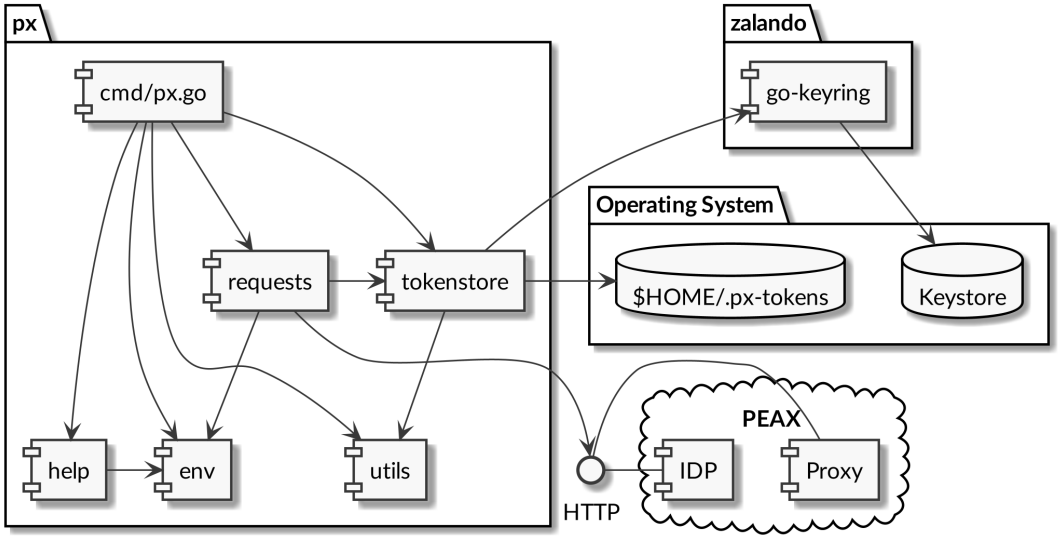


Abbildung 2: Die Package-Struktur von px (Komponentendiagramm)

help Dieses Package umfasst die Hilfetexte. Zu jedem Befehl gibt es jeweils eine kurze, einzeilige Beschreibung, und einen ausführlichen Hilfetext. Dieses Package importiert das env-Package, damit bei der Hilfe zum Login-Befehl (`px login`) die verfügbaren Umgebungen automatisch aufgelistet werden können.

tokenstore Hier sind zentrale Datenstrukturen für das Token-Handling mit dazugehörigen Funktionen definiert. Die Datenstruktur `TokenPair` dient zur lokalen, persistenten Speicherung von Tokens. Sie wird auf Basis einer `TokenResponse` aufgebaut, indem Informationen extrahiert und anhand von Kontextinformationen ergänzt. Die Datenstruktur `TokenStore` speichert pro Umgebung und Token-Typ (`agent`, `user`) ein `TokenPair` ab. Der `TokenStore` wird als JSON-Struktur serialisiert und im `$HOME`-Verzeichnis des Benutzers abgespeichert. Der Zugriff auf den sicheren Keystore ist ebenfalls hier implementiert.

requests In diesem Package sind die eigentlichen Zugriffe auf die API via HTTP umgesetzt. Hier sind Datenstrukturen für die Credential-Payloads mit dazugehörigen Funktionen definiert, die für Login-Vorgänge verwendet werden (`credentials.go`. Für die verschiedenen Befehle (`login`, `get`, `upload` usw.) sind hier dazugehörige Funktionen definiert. Der transparente Retry-Mechanismus ist ebenfalls hier implementiert (`requests.go`). Das Package `tokenstore` wird verwendet, um die Requests mit

den notwendigen Authentifizierungsinformationen (`Authentication`-Header) auszustatten.

`utils` Dieses Package umfasst verschiedene Funktionen, die von verschiedenen anderen Packages und vom Hauptprogramm verwendet werden, jedoch nicht direkt zu den jeweiligen anderen Packages gehören. Die Eingabeaufforderung für Passwörter (sicher über ein SSH-Terminal, d.h. ohne Echo) ist etwa hier umgesetzt (`pwinput.go` und `pwinput_windows.go`[38]). Weiter gibt es Funktionen zum automatischen Ermitteln des MIME-Types einer Datei, und eine Funktion zur rekursiven Auflisten lesbarer Dateien in einem Unterverzeichnis.

`cmd/px.go` Dies ist der eigentliche Command Line Client mit der Einstiegsfunktion `main`. Die zur Verfügung stehenden Befehle weden in einer Map namens `commands` abgespeichert, wobei der Befehlsname den Key bildet, und der Wert eine Datenstruktur bestehend aus einer Funktionsreferenz, dem einzeiligen Infotext und dem längeren Hilfetext (siehe Package `help`) zusammengesetzt ist. Die `main`-funktion prüft, ob der eingegebene Befehl in der Map gefunden wurde, und führt diese dann aus. Jede dieser Command-Funktionen nimmt den `TokenStore` als Argument entgegen, und gibt einen optionalen Fehler zurück.[39] Der `TokenStore` wird zu Beginn von `main` aus `/.px-tokens` geladen, und am Ende (mit möglichen Änderungen) wieder in diese Datei zurückgeschrieben.[40] Die einzelnen Command-Funktionen sind selber für ihre Seiteneffekte (Ausgabe möglicher Payloads) verantwortlich. Die Fehlerausgabe wird über die Rückgabe eines Fehlers in `main` abgehandelt. Die Flags sind ebenfalls für jeden Befehl eigens definiert, wobei Gemeinsamkeiten in Hilfsfunktionen und Konstanten (Beschreibungstexte) ausgelagert sind. Das Parsen der Kommandozeilenargumente wird vom sehr mächtigen und komfortablen `flag`-Package übernommen, das Teil der Standard Library von Go ist. Tritt bei der Programmausführung ein Fehler auf, wird einerseits eine Fehlermeldung auf `stderr` geschrieben, andererseits der Rückgabewert `1` an den aufrufenden Prozess zurückgegeben.[41] Dieser Rückgabecode wird in den Testskripts über die Shell-Variable `$?` geprüft.

[38] Aufgrund eines offenen Fehlers (`https://github.com/golang/go/issues/34461`) funktioniert die sichere Passworteingabe nicht unter Windows. Aus diesem Grund wird mithilfe eines Build Constraints (`https://golang.org/pkg/go/build/#hdr-Build_Constraints`) für den Windows-Build eine unsichere Variante, für alle anderen Betriebssysteme die sichere Variante verwendet.

[39] Die Befehle `help` und `env` bilden eine Ausnahme.

[40] Die sicher verwahrten Tokens werden bei Bedarf, d.h. bei einem entsprechenden Request, nachgeladen.

[41] Im Gegensatz zu `0`, das für eine erfolgreiche Ausführung steht.

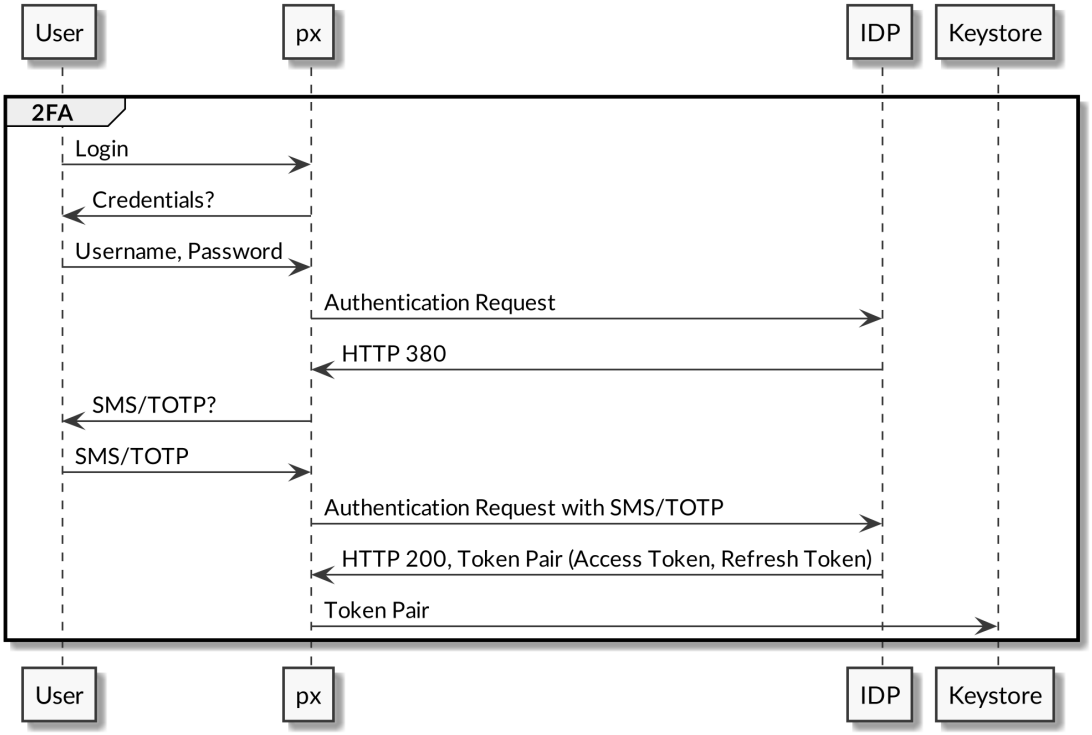


Abbildung 3: Der Ablauf der Zwei-Faktor-Authentifizierung mit SMS oder OTP (Sequenzdiagramm)

5.2 Zwei-Faktor-Authentifizierung

Hat ein Benutzer im Portal die Zwei-Faktor-Authentifizierung über SMS oder TOTP aktiviert, wird der initiale Login-Request mit Benutzername und Passwort mit dem HTTP-Status 380 quittiert. Diese Antwort enthält auch die Information, welcher zweite Faktor (SMS oder TOTP) verlangt wird.

Dem Benutzer wird eine entsprechende Eingebeaufforderung angezeigt. Nach der Eingabe wird der initiale Request mit dem eingegebenen zweiten Faktor ergänzt und erneut an den IDP gesendet. Hat der Vorgang funktioniert, erhält der Benutzer das Token-Pair, das im Keystore für die spätere Verwendung abgelegt wird.

Dieser Vorgang ist in Abbildung 3, Seite 43 mit einem Sequenzdiagramm veranschaulicht.

5.3 Token Store

Für jede Umgebung ist mit dem Flag `Confidential` definiert, ob die Tokens standardmässig sicher oder unsicher verwahrt werden sollen. Wie im Abschnitt Abschnitt 3.3.1 *Token Store*, Seite 31 beschrieben kann diese Einstellung mit den entsprechenden Flags übersteuert werden.

Wurde die unsichere Variante gewählt bzw. beibehalten, werden die beiden Tokens direkt in die JSON-Struktur geschrieben, die vor Beendigung des Clients nach `/.px-tokens` geschrieben wird. Für jede Kombination von Token-Art (`user` oder `agent`) und Umgebung (`test`, `stage` usw.) wird je ein Feld `refresh_token` und `access_token` gespeichert. Als zusätzliches Feld wird `use_keystore` mit dem Wert `false` abgespeichert.

Wird jedoch die sichere Variante gewählt bzw. beibehalten, wird unter der Kombination von Token-Art und Umgebung kein Token Pair abgespeichert, sondern bloss das Feld `use_keystore` mit dem Wert `true`, was bei künftigen Token-Zugriffen als Verweis auf den nativen Keystore zu verstehen ist.

Das folgende Codebeispiel zeigt einen verkürzten Auszug aus der Datei `/.px-tokens`, wobei für die Umgebung `test` die unsichere und für die Umgebung `stage` die sichere Token-Verwahrung verwendet worden ist.

```
{
        "tokens": {
                "user_test": {
                        "access_token": "eyJhbGciOiJSUz...",
                        "refresh_token": "eyJhbGciOiJIUz...",
                        "use_keystore": false
                },
                "user_stage": {
                        "use_keystore": true
                }
        },
        "default_environment": "stage"
}
```

Codebeispiel 12: Die JSON-Struktur für den Keystore (Auszug)

Der sichere Keystore ist als Key-Value-Store mit globalem Namensraum zu verstehen. Die Keys müssen also vollständig qualifiziert sein, d.h. alle Informationen enthalten, die etwa einen Refresh Token von `px` für einen Benutzer auf der Umgebung `prod` von einem auf dem Computer hinterlegten E-Mail-Passwort oder dergleichen unterscheidet. Hierzu

wurde das folgende Namensschema (Zeile 1) gewählt:

```
1 px:[environment]:[(user|agent)]:[(refresh|access)Token]
2 px:test:user:refreshToken
3 px:prod:agent:accessToken
```

Codebeispiel 13: Namensschema für die Keys auf dem nativen Keystore (mit Beispielen)

Zeile 2 zeigt den Key für einen Refresh Token für die User API auf der Umgebung `test`. Auf Zeile 3 ist ein Key für einen Access Token für die Agent API auf der Umgebng `prod` zu sehen.

5.3.1 Fremdkomponente `zalando/go-keyring`

Da sich der Zugriff auf den sicheren Keystore von Plattform zu Plattform unterscheidet, wurde dieser Mechanismus nicht selber implementiert. Das Package `go-keyring` von Zalando[42] löst dieses Problem bereits – und funktionierte auf Anhieb.

Das Package `go-keyring` wird nur im Package `tokenstore` verwendet, und sonst nirgends in `px`. Mithilfe der Funktionen `keyring.Set`, `keyring.Get` und `keyring.Delete` kann Secret abgespeichert, gelesen und gelöscht werden. Zu jedem Key wird nicht nur ein Passwort (das eigentliche Secret), sondern auch ein Benutzername abgelegt. Als Passwort dient jeweils der Token. Als Benutzername wurde der Vollständigkeit halber der String `"px"` verwendet.[43]

Diese drei Funktionen werden durch die eigenen Funktionen `storeSecret`, `loadSecret` und `deleteSecret` (`tokenstore/tokenpair.go`) gewrappt, wobei zusätzlich ein einheitliches Key-Format über die Funktion `tokenKeys` sichergestellt wird.

5.4 Retry-Mechanismus

Im Konzept (Abschnitt 3.3.2 *Token Refresh*, Seite 32) wurde beschrieben, nach welcher Strategie die Tokens aktualisiert werden sollen. Das Sequenzdiagramm Abbildung 4, Seite 46 zeigt, wie dies umgesetzt worden ist.

Einige Implementierungsdetails, die auf dem Sequenzdiagramm nicht ersichtlich sind, sollen an dieser Stelle noch erläutert werden.

[42] `https://github.com/zalando/go-keyring`

[43] Für `px` spielt es keine Rolle, *welcher* Benutzer eingeloggt ist. Validierungen auf den Scope zur Verhinderung von Fremdzugriffen werden serverseitig geprüft. Der Benutzer von `px` soll auch solche Zugriffe versuchen können, ohne schon clientseitig abgeblockt zu werden, um etwa Angriffe simulieren zu können.

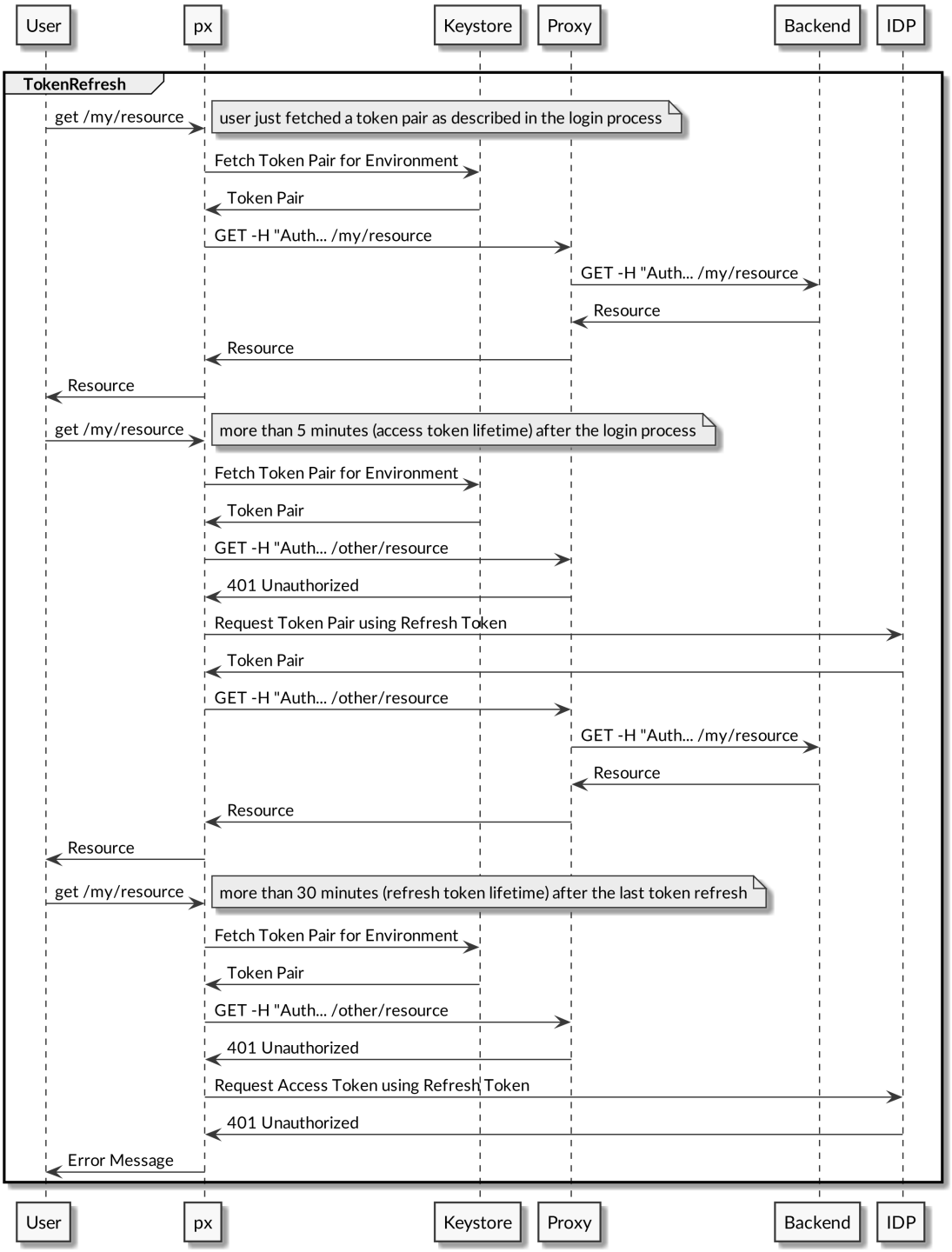


Abbildung 4: Der für den Benutzer transparente Retry-Mechanismus mit einem Token Pair, das im Hintergrund automatisch aktualisiert wird (Sequenzdiagramm)

Der erste, naive Implementierungsversuch, der darin bestand, einen gescheiterten Request (mit HTTP-Status `401 Unauthorized`) mit aktualisiertem `Authorization`-Header einfach erneut abzuschicken, scheiterte für Requests, die über einen Body verfügen, was bei `POST`, `PUT` und `PATCH` der Fall ist. Grund dafür ist, dass die Repräsentation des Bodies bei einem ausgeführten Request konsumiert wird, und anschliessend nicht mehr verwendet werden darf.

Technisch war dies zunächst so implementiert worden, dass die Funktion `doWithTokenRefresh` (`requests/requests.go`) als Parameter (nebst Kontextinformationen zum Auffinden der richtigen Tokens) einen Request erwartete. Die einzelnen Request-Funktionen (`Get`, `Post`, `Put`, `Patch` usw.) erstellten den initialen Request und übergaben ihn an `doWithTokenRefresh`, welches dann anhand des Token Stores einen Token Refresh durchführte, und den initialen Request mit aktualisiertem `Authorization`-Header erneut abschickte.

Die Lösung für das Problem mit den konsumierten Bodies konnte gelöst werden, indem für jeden erneuten Versuch ein neuer Request aufgebaut wird. Da `doWithTokenRefresh` jedoch nicht über alle Informationen verfügen kann, welche zum Erstellen eines Requests nötig sind, musste ein Weg gefunden werden, diese Kontextinformationen für alle möglichen Arten von Requests mitliefern zu können. Da Go *first class functions* unterstützt, konnte dies sehr elegant mit einer Closure gelöst werden (Donovan & Kernighan, 2015, S. 136). Die Client-Funktion übergibt nicht mehr einen Request, sondern eine Funktion, die einen Request erstellt. Die Kontextinformationen zur Erstellung des Requests erhält sie von der umschliessenden (*enclosing*) Funktion; den aktualisierten Access Token erhält sie zu einem späteren Zeitpunkt als Parameter, nachdem `doWithTokenRefresh` das Token Pair aktualisiert hat.[44]

5.5 Umgang mit Risiken

Von den ermittelten Risiken (Abschnitt 1.2 *Risikoanalyse*, Seite 11 sind bei der Umsetzung v.a. die technischen und sicherheitsrelevanten Risiken relevant. Mit ihnen wurde folgendermassen verfahren:

Token-Verwahrung Mit dem nativen Keystore wurde eine Lösung gefunden, die so sicher ist, wie der Keystore des zugrundeliegenden Betriebssystems. Möchte der Benutzer

[44] Wäre `px` mit einer stark objektorientierten Programmiersprache wie Java oder C++ umgesetzt worden, dürfte an dieser Stelle wohl ein Klassendiagramm mit `AbstractRequestFactory`, `GetRequestFactoryImpl` und dergleichen stehen. Peter Norvig hat demonstriert, dass in funktionalen Programiersprachen 16 der 23 GoF-Patterns *«invisible or simpler»* sind (`http://www.norvig.com/design-patterns/design-patterns.pdf`). Die Implementierung von `doWithTokenRefresh` veranschaulicht dieses Prinzip.

von px die Tokens der Produktivumgebung im Klartext abspeichern, muss er das explizit angeben. Die Kombination aus sicheren Grundeinstellungen und der Möglichkeit, diese bei Bedarf zu übersteuern, hat der Anwender eine standardmässig sichere, jedoch ausreichend flexible Lösung.

Payment-Schnittstelle Hat der Benutzer ein Bankkonto hinterlegt, und loggt er sich mit px auf das Produktivsystem ein, kann er Zahlungen mit den entsprechenden Requests in Auftrag geben. Es bestehen die gleichen Risiken wie bei der Bedienung des Portals. Auf eine unnötige und willkürliche Einschränkung der API-Abdeckung wurde verzichtet. Das Risiko liegt ganz beim Benutzer; die Sicherheit steht und fällt mit der Token-Verwahrung, welche sicher genug umgesetzt worden ist.

5.6 Notizen zur Implementierung und zum Testing

Während der Umsetzung von px wurden zu jeder User Story Umsetzungsnotizen und Testprotokolle geschrieben, die im Zusatzdokument *Backlog* (Abschnitt 8.4 *Weitere Dokumente*, Seite 63) zu finden sind. Da dieses Dokument recht umfassend ist, ist es als Referenz zu lesen. Hierbei ist jedoch zu beachten, dass das Dokument, gerade die Auflistung der User Stories, historisch gewachsen ist: Notizen früherer Stories wurden teils durch Notizen späterer Stories obsolet gemacht. Es empfiehlt sich also, bei der Lektüre so weit unten wie möglich – d.h. bei der jeweiligen Story, die von Interesse ist – einzusteigen, und bei Unklarheiten weiter oben nachzulesen, um den Hintergrund besser erfassen zu können.

Aufgrund der agilen Vorgehensweise gibt es keine umfassenden Testpläne und Testprotokolle, zumal das Testen nicht als von der Implementierung getrennte Phase praktiziert worden ist. Eine Ausnahme bilden die wenigen manuellen Tests (Regressionstests), die ab Sprint 3 jeweils durchgeführt worden und am Ende des *Backlog*-Dokuments verzeichnet sind.

6 Evaluation und Validierung

Das vorliegende Projekt entstammt dem PEAX-Ideation-Prozess, worin sich Phasen der Entwicklung (*Ideate*) und der Validierung (*Validate*) abwechseln. Innerhalb des Projekts wechselten sich Sprints zur Entwicklung mit einzelnen Wochen zur Dokumentation ab, wobei während letzterer jeweils Rückmeldungen von Kollegen (v.a. Entwickler) aufgenommen wurden (siehe Folgeabschnitt). Aus der Perspektive des Ideation-Prozesses war `px` jedoch für die letzten drei Monate in der *Ideate*-Phase, worauf nach Abschluss der vorliegenden Projektarbeit eine Validierung im Rahmen des Ideation-Gremiums folgen soll.

6.1 Rückmeldungen von Entwicklern

Nach jedem Sprint wird ein Inkrement an die Entwickler ausgeliefert (siehe Abschnitt 4.1 *Vorgehen*, Seite 34). Die Rückmeldungen werden hier gesammelt und kommentiert – und fliessen ins Backlog und nach Möglichkeit ins den jeweils nachfolgenden Sprint ein.

6.1.1 Sprint 1

- Michael Buholzer wünscht sich Erfolgs- und Vollzugsmeldungen nach dem Log-in oder dem Upload eines Dokuments.
  - `stdout` sollte grundsätzlich «sauber» bleiben, d.h. frei von unnötigen Ausgaben, die ein nachgelagertes Programm wieder herausfiltern müsste. Eine wichtige Maxime von Unix-Programmen lautet: *«Expect the output of every program to become the input to another, as yet unknown, program.»* (McIlroy Doug, Pinson, E.N, Tague, B.A., 1978, S. 3). Siehe dazu auch *Rule of Silence* (Raymond, 2004, S. 20) und *Silence is Golden* (Gancarz, 1995, S. 111).[45]
  - `stderr` wird nicht nur als Ausgabekanal für Fehlermeldungen verwendet, sondern für Meldungen allgemein. Für Vollzugsmeldungen ist `stderr` vorzuziehen.
  - Da `stderr` in `px` bisher grundsätzlich für Fehlermeldungen verwendet wird,

[45] Brian W. Kernighan berichtet von der Zeit, als die Pipe Einzug in Unix hielt, womit die Ausgabe eines Programms zur Eingabe eines anderen Programms gemacht werden konnte: *«Ken [Thompson] and Dennis [Ritchie] upgraded every command on the system in a single night. [...] Overall, the job was not hard—most programs required nothing more than eliminating extraneous messages that would have **cluttered a pipeline**, and sending error reports to stderr.»* (Kernighan, 2019, S. 69)

sollen Erfolgsmeldungen über ein zusätzliches Flag `-v`/`-verbose` aktiviert werden müssen.

  - Bei anderen Anwendungsfällen signalisiert die Ausgabe des Payloads auf `stdout` den Erfolg der Operation. Beim Dokument-Upload besteht dieser etwa aus der generierten UUID des hochgeladenen Dokuments.

- Patrick Roos sieht die Möglichkeit, `px` auch zur Handhabung der *Vault Secrets*[46] (zur Verschlüsselung und Entschlüsselung von Benutzernamen, Passwörtern etc.) zu verwenden.
  - Im Arbeitsalltag von PEAX stellt die Handhabung von Vault Secrets tatsächlich eine teils mühsame und langwierige Aufgabe dar. Hier besteht durchaus Automatisierungsbedarf.
  - `px` ist als «skriptbare» Anwendung für die PEAX API konzipiert und so potenziell für jeden PEAX-Anwender einsetzbar.
  - Die Verwaltung und Verwendung von Vault Secrets ist hingegen eine Aufgabe im DevOps-Bereich und betrifft nur interne Entwickler bei PEAX.
  - Eine der obersten Maximen von Unix lautet: *«Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new ‹features.›* (McIlroy Doug, Pinson, E.N, Tague, B.A., 1978, S. 3) Die Verwaltung von Vault Secrets und das Ansprechen der PEAX API sind klar zwei verschiedene Sachen und somit nicht «one thing». Die genannte Idee muss also anderweitig weiterverfolgt werden.

6.1.2 Sprint 2

- Stefano Pellegrini fände einen Befehl `px version` sinnvoll, der die aktuelle Versionsnummer ausgibt. Damit könne man sicherstellen, dass man nicht etwa eine veraltete Version verwendet und (hinfällige) Rückmeldungen auf diese gibt.
  - Tatsächlich stellen die meisten Kommandozeilentools eine solche Möglichkeit zur Verfügung, wie z.B. Go oder Docker mit den Befehlen `go version` und `docker version`. Andere Werkzeuge, wie etwa der Command Line Client von Heroku oder das Tool Ripgrep stellen ein entsprechendes Flag zur Verfügung (`heroku -v` bzw. `rg --version`).

[46] `https://docs.ansible.com/ansible/latest/user_guide/vault.html`

  - Ein Versionstag ist bereits über das GIT-Repository verfügbar. Die jeweils aktuelle Version kann mittels `git describe --tags` abgerufen werden.
  - Der Linker von GO[47] erlaubt es mit dem Parameter `-ldflags` uninitialisierte Strings im Programmcode mit einem Wert zu belegen.
  - Heisst die Variable im `main`-Modul `Version`, kann die aktuelle Versionsnummer folgendermassen in die Binärdatei hineinkompiliert werden: `go build -ldflags="-X main.Version=$(git describe --tags)" cmd/px.go`
  - Der Befehl `px version` soll wie gewünscht umgesetzt werden.

- Weiter hat STEFANO PELLEGRINI vorgeschlagen, dass `logout` sich auf die jeweils aktuelle Standardumgebung beziehen soll, damit man nicht immer eine Umgebung mit dem `-e`/`-env`-Parameter angeben muss.
  - Der Vorschlag ist sinnvoll und soll entsprechend umgesetzt werden.

- MICHAEL BUHOLZER wünscht sich, dass die Ausgabe von JSON-Datenstrukturen automatisch formatiert wird (*pretty print*).
  - Grundsätzlich lässt sich die JSON-Ausgabe sehr einfach formatieren, indem sie mittels Pipe durch ein Programm wie `jq` gesendet wird[48].
  - GO bietet mit `json.Indent` eine sehr komfortable Funktion, womit ein beliebiger JSON-Payload[49] einfach formatiert werden kann.
  - Der Vorschlag kommt ins Backlog – jedoch mit tiefer Priorität, da das Problem mithilfe von `jq` einfach gelöst werden kann. Weitere Features im Zusammenhang mit generischer JSON-Verarbeitung sollen der UNIX-Philosophie entsprechend an `jq` oder ähnliche Programme mittels Pipe delegiert werden.

- Zudem schlägt MICHAEL BUHOLZER im Zusammenhang mit dem `get`-Befehl vor, dass die PEAX ID automatisch anhand des eingeloggten Benutzers ergänzt werden soll, und man so nicht beispielsweise `profile/api/v3/profile/785.2120.8339.75` sondern bloss `profile/api/v3/profile` eingeben muss. Die PEAX ID sei für den Benutzer von `px` nirgends ersichtlich.

[47] `https://golang.org/cmd/link/`

[48] Siehe `https://stedolan.github.io/jq/`, das auch die Möglichkeit bietet, Teile mittels einer DSL aus der Datenstruktur zu extrahieren.

[49] D.h. nicht nur ein JSON-Payload, dessen Struktur mittels einer `struct` und den entsprechenden Annotations beschrieben ist, was zu einem unverhältnismässigen Mehraufwand führen würde, zumal dann jeder mögliche Payload statisch beschrieben sein müsste.

  - Eine generische `GET`-Schnittstelle kann nur angeboten werden, wenn die Ressourcenpfade für den Client transparent sind. So ist es nicht möglich, die PEAX ID automatisch zu ergänzen, zumal sie nicht zwingend am Ende, sondern auch mitten im Ressourcenpfad auftreten kann.
  - Eine bereits angedachte Lösung sind Variablen im Ressourcenpfad, die vom Client automatisch ergänzt werden, z.B. `profile/api/v3/profile/{peaxid}`. Hiermit kann eine generische Schnittstelle gewährleistet werden – und für den Benutzer wird die Handhabung einfacher.
  - Evtl. wäre es sinnvoll, zu jeder Umgebung, die ein Login repräsentiert, ergänzende Token-Informationen ausgeben zu können, z.B. die PEAX ID.

- Stephan Korner meldet eine Reihe spezifischer Use-Cases, die ihm beim Testen der Mobile App (iOS) nützlich sein würden:
  1. den Dokumentstatus eines eingelieferten Dokuments (Agent API) setzen
  2. den Check-In-Prozess nach der Registrierung wiederholbar machen, d.h. zurücksetzen
  3. den Status von Rechnungen überschreiben
  4. die Handhabung von Organisationen ermöglichen
  - Diese Funktionalitäten werden nach der User Story 13 (Einlieferung von Dokumenten per Agent API) für den ersten Punkt bzw. mit den User Stories 14-17 (generische Schnittstellen für `POST`, `PUT`, `PATCH`, `DELETE`) unterstützt.

6.1.3 Sprint 3

- Patrick Roos würde `px` gerne auch gegen seine lokale Entwicklungsumgebung verwenden können.
  - Technisch ist dies möglich, die Umgebung `local` benötigt jedoch eine spezielle Konfiguration. So werden hier keine Domains verwendet, sondern bloss Hostnamen und Ports.
  - Der lokale Proxy ist unter Port `8050`, der IDP unter Port `8080` verfügbar. Der Host heisst jeweils `localhost`.
  - Das Keycloak-Realm heisst `peax-id-local`; die `clientId` `peax.local`.
  - Die Idee wird ins Backlog aufgenommen.

6.2 Ergebnisse

In den vergangenen 14 Wochen ist aus dem kleinen, undokumentierten und etwas holprig zu bedienenden Prototyp `px` eine Software geworden, die den ursprünglich gesteckten Zielen gerecht wird: Ein grosser Teil der API von PEAX kann über generische HTTP-Befehle (`GET`, `POST` usw.) abgedeckt werden; mit spezifischen Befehlen (`upload`, `deliver`) können häufige Use Cases komfortabel abgedeckt werden.[50]

Mit dem sicheren Keystore wird `px` den gestellten Sicherheitsanforderungen gerecht, ohne den Anwender dabei zu bevormunden. Die Handhabung von Access und Refresh Tokens wird dem Benutzer dank deren transparenter und automatischer Aktualisierung im Hintergrund abgenommen.

Mit dem Hochladen ganzer Verzeichnisstrukturen mit automatischem Tagging ist `px` nicht nur ein Hilfsprogramm für die Weiterentwicklung des PEAX-Portals, sondern auch ein mächtiger Prototyp für Kundenprojekte.

Insgesamt wurden ca. 3000 Zeilen Go-Code geschrieben. Die pragmatische Teststrategie, die zu einem grossen Teil auf bedienungsnahe Testskripts setzt, brachte ca. 700 Zeilen Bash-Code hervor. Mit einem kaum 50 Zeilen fassenden `Makefile` wurden die meisten wiederkehrenden Aufgaben (Quellcodeanalyse, Unit Tests starten, Kompilieren) mit wenig Aufwand automatisiert.

Die vorliegende Dokumentation ist überraschend schnell gewachsen. Gerade die Umsetzungsnotizen und Testprotokolle (Backlog-Dokument) wuchsen bei der Umsetzung praktisch um eine Druckseite pro Tag.

Das agile Vorgehen brachte wenige Projektmanagement-Artefakte hervor. Ein schlanker Projekt- und Meilensteinplan, ein konsequent nachgeführtes Arbeitsjournal und die Meilensteinberichte sind das Ergebnis davon.

Die fehlende Adaption, die das bisher letzte in Abschnitt 1.2 *Risikoanalyse*, Seite 11 angesprochene Risiko darstellt, kann zu diesem Zeitpunkt noch nicht eingeschätzt werden. Es wurden Rückmeldungen aufgenommen und – wenn sinnvoll – ins Backlog aufgenommen und/oder umgesetzt, sodass die Beteiligten die Software erhalten, die sie benötigen. Die Validierungsphase im Ideation-Prozess wird Aufschluss über die Adaption geben.

[50] Da die «signifikante Abdeckung» im Projektauftrag nicht quantifiziert worden ist, soll hier auch keine quantitative Auswertung vorgenommen werden. Im Bereich der User API konnten während der Entwicklung alle Endpoints mit den generischen HTTP-Befehlen angesteuert werden. Der Multipart-Request der Delivery-API zum Einliefern von Dokumenten wurde mit dem spezifischen Befehl `px deliver` umgesetzt. Zugriffe auf die Admin API wurden während des Projekts nie gewünscht – und deshalb nie unternommen.

7 Ausblick

Nach drei Monaten des Konzipierens, Implementierens und Dokumentierens liegt px in Version 0.4.1 vor.[51] Der Sprung auf die Versionsnummer 1.0.0 wurde nicht vollzogen, zumal es sich hierbei um ein Zwischenergebnis handelt, und noch viele nicht umgesetzte Ideen im Backlog gibt, deren Umsetzung px zu einem wesentlich «runderen» Anwendungserlebnis machen könnten.

7.1 Reflexion der Arbeit

Nach einem etwas holprigen Start mit einigen Unklarheiten in der Projekt- und Meilensteinplanung konnte bald mit der Umsetzung angefangen werden. Der bestehende Prototyp bot einerseits ein Framework, das die Entwicklung leitete, andererseits aber auch einiges an Code, der noch zu überarbeiten war. So konnten im ersten Sprint nicht alle geplanten User Stories umgesetzt werden. Die Investitionen in das Refactorig zu Beginn des Projekts zahlten sich aber im weiteren Verlauf aus, sodass bei den drei weiteren Sprints jeweils alle gesteckten Ziele erreicht werden konnten.

Das vorgängige Aufsetzen der Dokumentation mit LaTeXerwies sich als lohnende Investition. Es ist sehr motivierend, wenn Erweiterungen am Text per Knopfdruck[52] ein druckreifes Dokument zum Ergebnis haben. Die Nebendokumente (Backlog, Arbeitsjournal usw.) wurden mithilfe von Markdown und Pandoc erzeugt. Die meisten Grafiken wurden mithilfe von Graphviz[53] und PlantUML[54] erstellt. Dadurch konnten mit Ausnahme der Meilensteingrafik alle Artefakte in textueller Representation in die Versionskontrolle aufgenommen werden.

Dies erlaubt etwa eine automatisierte Auswertung des Arbeitsjournals, welche die Aufwände pro User Story und Arbeitsbereich (allgemeine Projektaufgaben, Recherche, Dokumentation, Umsetzung) mithilfe eines AWK-Skripts aufsummiert.[55] Die Unix-Philosophie war somit nicht nur ein Leitfaden für die Softwareentwicklung, sondern für das ganze Projekt.

[51] Nach dem letzten Meilenstein wurde noch ein Fehler behoben, der eine automatische Aktualisierung von sicher verwahrten Tokens beeinträchtigte.

[52] Für die Dokumentation wurde ebenfalls ein `Makefile` verwendet.

[53] `https://www.graphviz.org/`

[54] `https://plantuml.com/de/`

[55] Bisher wurden 56.5 Stunden in die Dokumentation, 26 Stunden in allgemeine Projektaufgaben, 7.5 Stunden in die Recherche und 73.5 Stunden in die Umsetzung investiert; d.h. total 163.5 Stunden Aufwand (Stand: Montag, 16.12.2019).

Mit der gewählten Programmiersprache Go habe ich sehr viele positive Erfahrungen gemacht. Ja: das manuelle Error-Handling ist teils anstrengend, mühsam und führt zu repetitivem Code – der jedoch mit etwas Disziplin grundsolide und sehr gut lesbar ist. Der Entwickler wird dazu gezwungen, sich mit jeder Fehlermöglichkeit einzeln auseinanderzusetzen, wodurch es selten Überraschungen – und solche nur mit aussagekräftigen Fehlermeldungen gibt. Das oft monierte «Fehlen» von parametrischem Polymorphismus (vulgo «Generics») in Go stellte nie ein Problem dar.[56] Während der Arbeit lernte ich Go viel besser kennen, auch spezielle Features wie Linker-Flags und Build-Annotations, sowie die ganze Toolchain. Vim mit dem Plugin `vim-go`[57] erwies sich als schlanke und komfortable Entwicklungsumgebung.

Die Arbeit wurde grösstenteils in der Freizeit durchgeführt; meistens frühmorgens vor der Arbeit im Büro. Die skriptbasierte Teststrategie, die ein funktionierendes Backend voraussetzt, führte einmal dazu, dass ich die Arbeit am Wirtschaftsprojekt unterbrechen musste, um mich im Büro um die nicht funktionierende Testumgebung zu kümmern.[58] Trotz solcher Episoden würde ich die gewählte Teststrategie wieder auf ein ähnliches Problem anwenden, wenn auch mit einem etwas grösseren Fokus auf Unit Tests.

Insgesamt war das Wirtschaftsprojekt sehr lehrreich, wenn auch aufgrund der hohen Arbeitsbelastung sehr anstrengend. Für die übrigen Module musste ich in diesem Semester viele Abstriche machen. Die inkrementell wachsende Software bot die nötige Motivation; das inkrementell wachsende Arbeitsjournal mit der automatischen Zeitauswertung sorgte für den nötigen Druck. Mit dem Ergebnis bin ich zufrieden, und ich habe bei der Arbeit viel gelernt.

7.2 Ungelöste Probleme

- Adaption - mögliches Refactoring: lange Parameterlisten, teils duplizierter Code - evtl. bessere Testabdeckung möglich

7.3 Weitere Ideen

- siehe Backlog - OpenSource - mit der Entwicklung schritthalten

[56] Für das Schreiben von Libraries dürfte das Fehlen von Generics durchaus ein Problem darstellen; weniger bei Client-Anwendungen.

[57] `https://github.com/fatih/vim-go`

[58] Lauffähige Testumgebungen haben auch für das Tagesgeschäft eine höhere Priorität als die Weiterentwicklung von px.

8 Anhang

Der Anhang besteht aus den folgenden drei Teilen:

1. der Systemspezifikation mit Kontextdiagramm,
2. der Technologieevaluation, v.a. bezogen auf die Programmiersprache, und
3. einer Liste von weiteren Dokumenten, die mit dem vorliegenden Dokument zusammen abgegeben werden.[59]

8.1 Systemspezifikation

Bei px handelt es sich um ein modular aufgebautes Kommandozeilenprogramm. Der Code ist in zwei Teile eingeteilt: das px-Modul (Library), das die Funktionalität anwendungsneutral zur Verfügung stellt, und das Kommandozeilenprogramm `cmd/px.go`, das die kommandozeilenspezifischen Operationen (Interpretation der Parameter, Ein- und Ausgabe) übernimmt, und Gebrauch vom px-Modul macht.

8.1.1 Systemkontext

Im Kontextdiagramm (Abbildung 5, Seite 57) wird die zu entwickelnde Komponente px im Systemkontext von PEAX dargestellt. Andere Komponenten sind als Ellipsen, Schnittstellen als Rechtecke dargestellt. Ausserhalb vom Systemkontexts befindet sich die irrelevante Umgebung (d.h. die Frontend-Anwendungen). Die im Kontextdiagramm verwendeten Begriffe haben folgende Bedeutungen:

- px: Der PEAX Command Line Client (Gegenstand der vorliegenden Arbeit)
- Developer: Ein Softwareentwickler (im weitesten Sinne) bei PEAX, der px verwendet.
- Backoffice User: Ein PEAX-Angestellter mit administrativen Befugnissen (Benutzerverwaltung).
- Portal User: Ein Benutzer des PEAX-Portals (Kunde).
- Scanning Center: Zulieferfirma, welche die umgeleitete Papierpost der PEAX-Kunden erhält, diese einscannt und dem betreffenden Kunden ins Portal stellt.

[59] Die Dokumente wurden nicht physisch zusammengehängt, damit die Outline der digitalen Version zwecks einfacherer Navigierbarkeit erhalten bleibt.

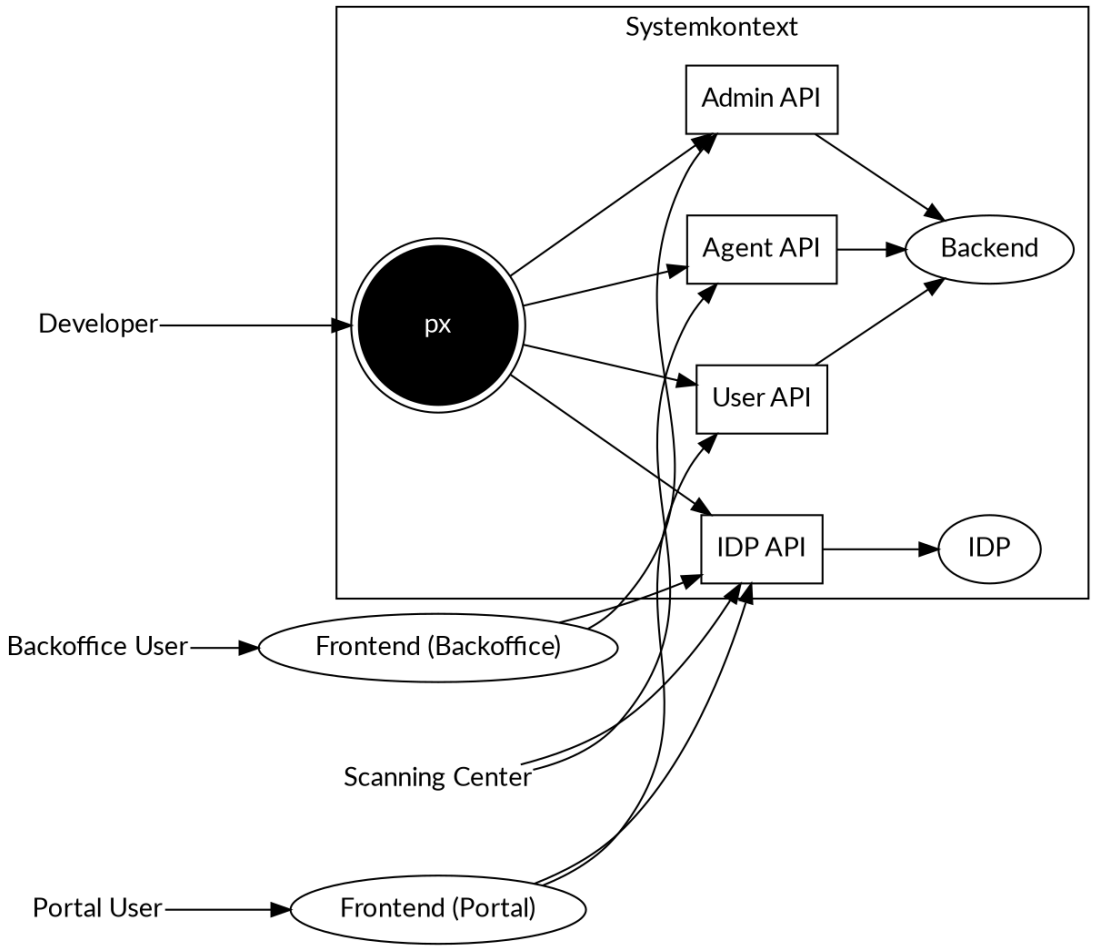


Abbildung 5: Kontextdiagramm: px als der Gegenstand der Arbeit innerhalb des Systemkontexts

- Frontend (Backoffice): Ein Web-GUI für administrative Tätigkeiten zum internen Gebrauch.
- Frontend (Portal): Ein Web-GUI für die Kunden von PEAX (das eigentliche Portal).
- Admin API: RESTful API für administrative Aufgaben
- Agent API: RESTful API zum Einliefern von Dokumenten über Zulieferer
- User API: RESTful API für die Operationen der Kunden
- IDP API: RESTful API für das Token-Management (OAuth 2.0/OpenID Connect)
- Backend: Serverseitige Software mit Businesslogik und Datenspeicher
- IDP: Identity Provider, API-übergreifende Benutzer- und Zugangsverwaltung (AuthN/AuthZ)

8.1.2 Architektur und Designentscheide

Die Architektur der Anwendung ist im Abschnitt 5 *Realisierung*, Seite 40, Designentscheide sind im Abschnitt 3 *Ideen und Konzepte*, Seite 18 beschrieben.

Limitierende Mengengerüste sind im relevanten Systemkontext aufseiten des Backends zu verorten und liegen daher nicht im Scope der vorliegenden Arbeit. Lokal fallen pro Umgebung maximal zwei Token Pairs (Access und Refresh Token; User- und Agent-API) an, d.h. vier Tokens. Mit den bis dato acht verfügbaren Umgebungen ergibt dies eine maximale Anzahl von 32 gleichzeitig abzuspeichernden Tokens. Ein Access Token, der grössere der beiden Tokens, hat eine Länge von ca. 1800 Bytes; der kürzere Refresh Token wiegt die zusätzlichen abgespeicherten Informationen (PEAX ID, Token Type usw.) und den JSON-Overhead bei weitem auf, sodass lokal maximal 57'600 Bytes abzuspeichern sind. Die ausführbaren `px`-Dateien sind mit ca. 7.5 bis 9.5 MB (je nach Plattform) wesentlich grösser.

8.1.3 Schnittstellen

Als externe Schnittstellen verwendet `px` die im Kontextdiagramm (Abbildung 5, Seite 57) verwendeten APIs des Backends und des IDPs. Diese Interaktionen sind im Kapitel Abschnitt 5 *Realisierung*, Seite 40 beschrieben. Weiter interagiert `px` mit dem Dateisystem (Token Store `~/.px-tokens`, temporäre Dateien für Payloads) des Betriebssystems. Die

Schnittstelle zum nativen Keystore wurde mit der Fremdkomponente `go-keyring` von ZALANDO (Abschnitt 5.3.1 *Fremdkomponente* `zalando/go-keyring`, Seite 45) gelöst.

Die internen Schnittstellen sind am besten in der Package-Übersicht (Abschnitt 5.1 *Architektur: Package-Übersicht*, Seite 40) ersichtlich.[60]

Die Benutzerschnittstelle ist im Kapitel Abschnitt 3.2 *Swiss Army Knive*, Seite 23 konzeptionell und anhand konkreter Beispiele erläutert.

8.1.4 Environment-Anforderungen

An die Umgebung stellt `px` nur geringe Anforderungen. Zur Ausführung wird MACOS, WINDOWS oder LINUX in einer mehr oder weniger aktuellen Version benötigt. Tests wurden auf MACOS CATALINA, WINDOWS 10 und ARCH LINUX erfolgreich ausgeführt. Zur Bedienung wird eine UNIX-Shell wie z.B. BASH empfohlen, wobei `px` auch mit `cmd.exe` oder der POWERSHELL funktioniert. Eine funktionierende Internetverbindung wird angenommen. Soll die sichere Verwahrung der Tokens auf LINUX funktionieren, ist eine Software wie SEAHORSE zu installieren. Die `README`-Datei im Repository gibt Auskunft über deren Konfiguration.

Grundsätzlich kann `px` auf allen Betriebssystemen und Plattformen kompiliert werden, auf denen GO läuft.[61] Die aktuelle Build-Konfiguration geht von der `amd64`-Architektur aus. Mithilfe der Umgebungsvariablen `GOOS` und `GOARCH` kann `px` auf eine Vielzahl von Umgebungen cross-kompiliert werden.[62]

Dateien werden jeweils mit dem Encoding `UTF-8` gelesen und geschrieben. Für Quellcodedateien ist dies bei GO sogar eine verbindliche Vorgabe.[63]

8.2 Technologie-Evaluation

Im Bereich der Technologie gilt es vor allem eine passende Programmiersprache auszuwählen. Die API von PEAX wird bereits durch den Projektrahmen vorgegeben, und dadurch auch die Protokolle OAuth 2.0 und HTTP.

[60] Führt man im Quellcodeverzeichnis von `px` den Befehl `godoc` aus, kann die Quellcodedokumentation komfortabel im Browser unter `http://localhost:6060/pkg/px/` betrachtet werden.

[61] siehe `https://github.com/golang/go/wiki/MinimumRequirements`

[62] `https://gist.github.com/asukakenji/f15ba7e588ac42795f421b48b8aede63` bietet eine gute Übersicht.

[63] siehe `https://golang.org/ref/spec#Source_code_representation`

8.2.1 Programmiersprache

Aus der Aufgabenstellung und dem Umfeld bei PEAX ergeben sich folgende nicht-funktionale Anforderungen an die zu erstellende Software:

Installation Die Software soll sich einfach installieren lassen.

Umgebung Es dürfen keine besonderen Anforderungen an die Umgebung gestellt werden, auf der `px` läuft.

Plattformen Die Software soll auf allen gängigen, d.h. bei PEAX eingesetzten, Betriebssystemen (Windows, macOS, Linux) lauffähig sein.

Einheitlichkeit Der Client soll überall die gleiche Befehlssyntax haben.

Performance Ein Command Line Client soll in Skripten verwendet werden können, wodurch das Programm sehr oft in kurzem Zeitraum aufgestartet werden muss.

Java, das bei PEAX im Backend-Bereich zum Einsatz kommt, erfordert die lokale Installation einer JRE in der richtigen Version, was bei Frontend-Entwicklern nicht gegeben ist. Ausserdem werden Wrapper-Skripts benötigt (`java -jar px.jar` ist nicht praktikabel).

Python, Ruby, Perl und andere Skriptsprachen benötigen ebenfalls einen vorinstallierten Interpreter in der richtigen Version.

Zwar gibt es mit Mono eine Variante von .Net, die überall lauffähig ist, hier werden aber wiederum eine Laufzeitumgebung bzw. vorinstallierte Libraries benötigt.

Für die Problemstellung am besten geeignet sind kompilierte Sprachen (C, C++, Go, Rust, Nim). Mit einer statischen Kompilierung lässt sich das ganze Programm in eine einzige Binärdatei überführen, welches denkbar einfach zu installieren ist (Kopieren nach einem der Verzeichnisse innerhalb von `$PATH`).

Für JavaScript, das bei PEAX im Frontend zum Einsatz kommt, gibt es mit QuickJS[64] seit kurzem die Möglichkeit, JavaScript zu Binärdateien zu kompilieren. Dies funktioniert aber nicht auf allen Plattformen, ausserdem ist QuickJS noch experminentell und noch nicht für den produktiven Einsatz geeignet.

Um ein Projekt vom gegebenen Umfang innerhalb eines Semesters umsetzen zu können, sind Vorkenntnisse in der einzusetzenden Programmiersprache zwar nicht zwingend, können das Risiko des Scheiterns aber erheblich senken. Gerade bei der Abschätzung von Aufwänden ist Vertrautheit mit den einzusetzenden Werkzeugen sehr hilfreich.

[64] `https://bellard.org/quickjs/`

Was (statisch) kompilierte Programmiersprachen betrifft, konnte der Autor dieser Arbeit bereits Erfahrungen mit C, Go und Rust sammeln. Das manuelle Speichermanagement in C (u.a. auch bei Strings) ist einerseits ein grosses Risiko (Buffer Overflows, Segmentation Faults), und wirkt sich andererseits negativ auf das Entwicklungstempo aus. In die engere Auswahl kommen somit Go und Rust.

Im Folgenden werden die gemachten Erfahrungen und die dabei empfundenen Vor- und Nachteile mit den Programmiersprachen Go und Rust einander gegenübergestellt.

8.2.2 Go

Mit Go konnte der Autor dieser Arbeit bereits einges an Erfahrung sammeln. So wurde neben dem Prototyp zu px bereits die Testat-Aufgabe im Modul Software Testing[65], ein Thumbnailer[66] sowie zahlreiche Utilities[67] (viele darunter als HTTP-Clients) in Go entwickelt. Dabei wurden folgende Vor- und Nachteile ermittelt:

+ aufgrund weniger Keywords und Features einfach zu lernen
+ hervorragendes Tooling out-of-the-box
+ Cross-Compilation ohne Zusatztools auf alle unterstützte Plattformen möglich
+ schnelle Kompilierung
+ umfassende Standard-Library, die u.a. ein hervorrangendes HTTP-Package beinhaltet
+ persönlich bereits viel (positive) Erfahrungen damit gesammelt
+ wird bereits für andere bei PEAX gebräuchliche CLI-Tools eingesetzt (`oc`, `docker`)
+ fügt sich sehr gut in die Unix-Philosophie ein (Tooling, Libraries)
+ Einfaches Interface für nebenläufige Programmierung (Goroutines und Channels)
+ geringer Memory-Verbrach bei relativ hoher Performance[68]
- keine Features wie Generics, Exceptions und `filter`/`map`/`reduce`

[65] `https://github.com/patrickbucher/getting-to-philosophy`
[66] `https://github.com/patrickbucher/thumbnailer`
[67] `https://github.com/patrickbucher/go-scratch`
[68] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go.html`

- Binaries fallen relativ gross aus[69]
- Error-Handling aufwändig und teils repetitiv

8.2.3 Rust

Der Autor dieser Arbeit konnte sich bereits letztes Jahr im Rahmen des Moduls *Programming Concepts and Paradigms* an der HSLU Informatik mit Rust befassen (Arnold & Bucher, 2018, S. 12). Nach selbständiger Beschäftigung mit dieser Programmiersprache im Sommer können (teils ergänzend) folgende Vor- und Nachteile genannt werden:

+ viele moderne Features (Generics, `filter`/`map`/`reduce`)
+ hervorragendes Typsystem
+ gutes und ausgereiftes Tooling
+ weder manuelles Memory-Management noch Garbage Collector nötig
+ Pattern Matching führt zu sehr solidem Code
+ gegenüber Go schlankere Binaries
+ kommt bereits in der Form einiger CLI-Tools persönlich zum Einsatz (`rg`, `bat`, `hexyl`, `battop`)
+ erstklassige Performance (im Bereich von C/C++) bei geringem Speicherverbrauch[70]
- hohe Einstiegshürde und lange Einarbeitungszeit
- Cross-Compilation benötigt Zusatztools
- noch keine praxisnahe Erfahrung damit gesammelt
- aufgrund schlanker Standard Library auf viele Dependencies angewiesen

[69] `https://golang.org/doc/faq#Why_is_my_trivial_program_such_a_large_binary`
[70] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-go.html`

8.2.4 Entscheidung Programmiersprache

RUST hat gegenüber GO einige unbestreitbare Vorzüge (Memory Management, Typsystem, Ausdrucksstärke, Eliminierung ganzer Fehlerklassen, Performance, schlankere Binärdateien). Bezogen auf das umzusetzende Projekt haben jedoch einige davon kaum einen wichtigen Stellenwert (etwa Performance und Zero-Cost Abstractions). Hier fallen die Vorzüge von GO (umfassende Standard Library, Cross-Compilation) wesentlich stärker ins Gewicht.

Gerade die absichtlich schlank gehaltene Standard Library von RUST, die etwa zur Generierung von Zufallszahlen bereits externe Abhängigkeiten erfordert[71], dürfte sich im vorliegenden Projektrahmen negativ auswirken, zumal die Evaluation verschiedener Libraries einen sehr hohen Zusatzaufwand erfordert.

Da GO bereits bei der Entwicklung des Prototypen von px erfolgreich zum Einsatz kam, und einige Projektaspekte (grundlegende CI-Pipeline, `Makefile` für Cross-Compilation und Packaging) bereits damit implementiert werden konnten, soll GO für das vorliegende Projekt den Vorzug erhalten.

Eine spätere Neuimplementierung von px in RUST wäre ein technisch durchaus interessantes, wenn auch praktisch wenig dringendes – als Fallstudie aber durchaus lohnendes – Unterfangen.

8.3 Libraries

Wie bereits in Abschnitt 5.3.1 *Fremdkomponente* `zalando/go-keyring`, Seite 45 geschildert, wird nur eine einzige Fremdkomponente verwendet: Das Package `go-keyring` von ZALANDO. Für die sichere Passworteingabe wird ein SSH-Terminal (`golang.org/x/crypto`) verwendet, das zu den halboffiziellen GO-Packages gehört.[72]

8.4 Weitere Dokumente

Projektauftrag Im Projektauftrag (`Anhang/Projektauftrag.pdf`) ist die Aufgabe beschrieben, wie sie zu Beginn des Projekts definiert worden ist.

Projektplan Der Projektplan (`Anhang/Projektplan.pdf`) besteht aus einem Rahmenplan, einem Meilensteinplan und einem Wochenplan.

[71] `https://doc.rust-lang.org/book/ch02-00-guessing-game-tutorial.html#using-a-crate-to-get-more-functionality`

[72] Die GO-Packages unter `golang.org/x` sind vergleichbar mit den JAVA-Libraries unter `javax`.

Backlog Das Backlog (`Anhang/Backlog.pdf`) enthält die einzelnen User Stories, die Sprint-Planung, Umsetzungsnotizen zu einzelnen User Stories mit Testprotokolle dazu, sowie Informationen zu manuellen Tests und gefundenen Bugs.

Meilensteinbericht 1 Der erste Meilensteinbericht (`Anhang/Meilensteinbericht-1.pdf`) beschreibt die Vorgeschichte des Projekts und die Phase der Projektinitialisierung.

Meilensteinbericht 2 In diesem Meilensteinbericht (`Anhang/Meilensteinbericht-2.pdf`) geht es um die ersten beiden Sprints.

Meilensteinbericht 3 In diesem Meilensteinbericht (`Anhang/Meilensteinbericht-3.pdf`) geht es um die letzten beiden Sprints.

Arbeitsjournal Im Arbeitsjournal (`Anhang/Arbeitsjournal.pdf`) sind die einzelnen Aufwände auf halbe Stunden gerundet nach Bereich – Projekt(administration), Dokumentation, Umsetzung – rapportiert. Mithilfe eines AWK-Skripts können die Aufwände nach Bereich und User Story ausgewertet werden.

Glossar

Admin API API zur Verwaltung der PEAX-Benutzer und zur Konfiguration benutzerübergreifender Einstellungen (z.B. Bankfeiertage, verfügbare Organisationen für Postabonnierungen), die über das sogenannte Backoffice von entsprechend befugten PEAX-internen Mitarbeitern verwendet wird.

Backoffice Web-Anwendung für administrative Aufgaben, siehe *Admin API*.

IDP Identity Provider: Service, der die Benutzerkonti mit persönlichen Angaben, Identifikation, Authentifizierungsmechanismen (Passwörter, Tokens), Berechtigungen usw. verwaltet und diesen Authentifizierungsmechanismen zur Verfügung stellt.

PEAX ID GTIN-13-Identifikation[73] eines PEAX-Benutzers, z.B. `123.4567.8901.23`. Wird oft als Teil des Ressourcenpfades von HTTP-Endpoints verwendet.

[73] `https://www.gin.info/`

Literatur

American National Standards Institute. (1993). *IEEE standard for information technology: Portable Operating System Interface (POSIX) : part 2, shell and utilities*. IEEE Computer Society Press.

Arnold, L. & Bucher, P. (2018, December). *Rust* (Bericht). (https://github.com/patrickbucher/pcp-project/raw/master/Rust-Arnold-Bucher/paper.pdf)

Bucher, P. & Christensen, C. (2019, May). *OAuth 2* (Bericht). (https://github.com/patrickbucher/inf-stud-hslu/raw/master/infkol/thesis/OAuth2_Bucher-Christensen.pdf)

The Code. (2001). (https://www.youtube.com/watch?v=XMm0HsmOTFI)

Crispin, L. & Gregory, J. (2008). *Agile Testing*. Addison-Wesley.

Donovan, A. A. A. & Kernighan, B. W. (2015). *The Go Programming Language*. Addison-Wesley.

Dusseault, L. & Snell, J. (2010, March). *PATCH Method for HTTP* (RFC Nr. 5789). RFC Editor. Internet Requests for Comments. Zugriff auf http://www.rfc-editor.org/rfc/rfc5789.txt (http://www.rfc-editor.org/rfc/rfc5789.txt)

Elliot, E. (2019). *Composing Software*. Leanpub. https://leanpub.com/composingsoftware.

Fielding, R. & Reschke, J. (2014, June). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content* (RFC Nr. 7231). RFC Editor. Internet Requests for Comments. Zugriff auf http://www.rfc-editor.org/rfc/rfc7231.txt (http://www.rfc-editor.org/rfc/rfc7231.txt)

Fielding, R. T., Gettys, J., Mogul, J. C., Nielsen, H. F., Masinter, L., Leach, P. J. & Berners-Lee, T. (1999, June). *Hypertext transfer protocol – http/1.1* (RFC Nr. 2616). RFC Editor. Internet Requests for Comments. Zugriff auf http://www.rfc-editor.org/rfc/rfc2616.txt (http://www.rfc-editor.org/rfc/rfc2616.txt)

Gancarz, M. (1995). *The UNIX Philosophy*. Digital Press.

Gerardi, R. (2019). *Powerful Command-Line Applications in Go*. The Pragmatic Programmers.

Hilt, V., Camarillo, G. & Rosenberg, J. (2012, December). *A Framework for Session Initiation Protocol (SIP) Session Policies* (RFC Nr. 6794). RFC Editor. Internet Requests for Comments. Zugriff auf https://tools.ietf.org/html/rfc6794 (https://tools.ietf.org/html/rfc6794)

Kernighan, B. W. (2019). *UNIX*. Kindle Direct Publishing.

Klabnik, S. & Nichols, C. (2019). *The Rust Programming Language*. no starch press.

Laboon, B. (2017). *A Friendly Introduction to Software Testing*.

McIlroy Doug, Pinson, E.N, Tague, B.A. (1978). *UNIX Time-Sharing System* (Bericht). Bell System Technical Journal 57. (https://archive.org/details/bstj57-6-1899)

Mozilla Developer Network. (o. J.). *Cross-Origin Resource Sharing (CORS).* https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. (Abgerufen am 26.10.2019)

P. Bryan, M. N. (2013, April). *JavaScript Object Notation (JSON) Patch* (RFC Nr. 6902). RFC Editor. Internet Requests for Comments. Zugriff auf https://tools.ietf.org/html/rfc6902 (https://tools.ietf.org/html/rfc6902)

Raymond, E. S. (2004). *The Art of UNIX Programming.* Addison-Wesley.

Abbildungsverzeichnis

Verzeichnis der Codebeispiele