px: PEAX Command Line Client

Wirtschaftsprojekt, Herbstsemester 2019

Patrick Bucher

11. November 2019

Abstract

Die Software px ist ein Kommandozeilenprogramm, das den Zugriff auf die API von PEAX für Benutzer mit einem technischen Hintergrund vereinfachen soll. PEAX ist ein digitaler Posteingang, womit der Endanwender seine Post über verschiedene Kanäle empfangen und in einem komfortablen Web-Portal verwalten kann. 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 eine Reihe intuitiv bedienbarer Befehle für das Ansteuern verschiedener Endpoints zur Verfügung gestellt wird.

Inhaltsverzeichnis

1 Problemstellung

Die Problemstellung setzt sich einerseits aus dem gestellten Projektauftrag (siehe Anhang) und andererseits aus der damit implizierten Umgebung (Systeme, Technologien, Benutze, etc.) zusammen (siehe dazu auch Abschnitt 8.1.1 *Systemkontext*, Seite 28).

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:

Technologie das Protokoll HTTP und der Authentifizierungsmechanismus OAuth 2.0

Server-Umgebung die Umgebungen, die eine PEAX API anbieten

Client-Umgebung 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 welchen Dokumente hochgeladen werden können.

1.1.2 HTTP-Methoden

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

- `GET`: Erfragt eine Repräsentation einer bestimmten Ressource; greift nur schreibend auf diese zu.
- `HEAD`: Analog zu `GET`, es wird jedoch nur der Header und nicht der Body der Ressource angefragt.
- `POST`: Übermittelt eine Ressource 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 werden u.a. folgende Status-Codes häufig verwendet:

- `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.
- `403 Forbidden`: Der Benutzer ist zwar eingeloggt, hat aber keine Berechtigung mit der gewählten Methode auf die jeweilige Resource zuzugreifen.
- `404 Not Found`: Die Resource wurde nicht gefunden; deutet auf eine fehlerhafte URL hin.

- `405 Method Not Allowed`: Die Resource 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, kam 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, 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 den dort zitierten Primärquellen entnommen werden.

An dieser Stelle sollen nur die Grundlagen beschrieben werden, die dann im Umsetzungsteil (siehe Abschnitt 5 *Realisierung*, Seite 23) bei Bedarf genauer eingegangen wird.

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 sich der Benutzer einen neuen Access Token vom IDP holen. Der Refresh Token ist darum langlebiger (30 Minuten bei PEAX).

[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 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.

Es ist die Handhabung dieser Tokens (sehr lange, Base64-codierte Strings), die das Ansteuern der PEAX API mit Programmen wie `curl` und POSTMAN so mühsam machen.[2]

1.1.5 Umgebungen

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

local Die Entwickler können den PEAX-Stack zum Entwickeln und Testen lokal ausführen.

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.

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 (Ziel: einmal pro Monat) 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.

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.

[2] Eine beispielhafte Analyse ergab, dass ein Access Token 1604 Zeichen lang ist.

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 für Performance-Tests verwendet und ist seither nur sporadisch in Betrieb.

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 der Request-Bodys 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 über 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-API, 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.

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 gibt es dennoch einige potenzielle Benutzer (siehe Abschnitt 1.1.7 *Benutzer*, Seite 8), 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 (Apple Support, 2019). Andere populäre UNIX-Shells wie Fish, Ksh und Tcsh haben zwar 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, das den Entwickleralltag 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 die Entwickler 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.
- **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.

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.

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[3] 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 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 in eine Variable `token` abgespeichert werden. Diese Variable kann dann im einen weiteren Request 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.

[3] `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[4] automatisieren liesse, werden Abläufe damit in der Praxis als 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 Servern oder Containern aus, wo kein GUI zur Verfügung steht, sind die Skripts also nicht ausführbar.

2.1.2 cUrl

Bei `curl` handelt es sich um ein Kommandozeilenprogramm zum Ansprechen verschiedenster Protokolle[5], 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[6].

`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`[7] 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 eine ungenutzte Möglichkeit, sondern wird auch tatsächlich genutzt.

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

`curl` benötigt zudem einiges an Einarbeitungszeit, um die einzelnen Kommandozeilenparameter zu verstehen – und zusätzliches Verständnis von Kommandozeilenkon-

[4] `https://learning.getpostman.com/docs/postman/scripts/intro_to_scripts`
[5] `https://curl.haxx.se/`
[6] Beispiel: Einem bestimmten Benutzer für Testzwecke 2000 Dokumente per Delivery-API ins Portal stellen
[7] `https://stedolan.github.io/jq/`

zepten (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`.

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 Name etabliert hat[8], soll es im Rahmen der vorliegenden Arbeit als *Swiss Army Knife*-Ansatz bezeichnet werden, zumal diese Bezeichnung im Standardwerk zu Go für das `go`-Tool verwendet wird (Donovan & Kernighan, 2015).

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

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, p. 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[9]. Als erstes wurde eine CI-Pipeline bestehend aus den Schritten 'build' und 'test' erstellt. 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, p. 293) erstellt, welches dazu diente, die Kompilierung für verschiedene Plattformen zu testen. Obwohl 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`[10]) 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)

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

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

[9] `https://gitlab.peax.ch/peax3/px`
[10] `https://semver.org/`

`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[11] erreicht wird. Bei einem erfolgreichen Login-Versuch werden `access_token` und `refresh_token` aus dem Response-Payload gelesen und im `$HOME`-Verzeichnis des jeweiligen Betriebssystem-Benutzers in eine JSON-Datei namens `.px-tokens` abgespeichert.

`px logout` löscht ein Token-Paar für eine bestimmte Umgebung. Pro Umgebung kann es zu jedem Zeitpunkt nur ein aktives «Login», d.h. Token-Paar 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[12].

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.

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

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

3 Ideen und Konzepte

TODO: swiss army knive (Donovan & Kernighan, 2015, p. 290) für CLI

TODO: token storage (sicher und unsicher)

TODO: automatisches Retry

4 Methoden

Dieses Kapitel beschreibt nicht das WAS, sondern das WIE: Hier soll es nicht um einzelne Features und deren Implementierun 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. Die Umsetzungsnotizen fliessen jeweils nach dem Sprint in die Dokumentation ein. Ein Story-übergreifender Test-Plan soll nicht erstellt werden. Die Teststrategie wird im folgenden Unterkapitel beschrieben.

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 34) genauer beschrieben.

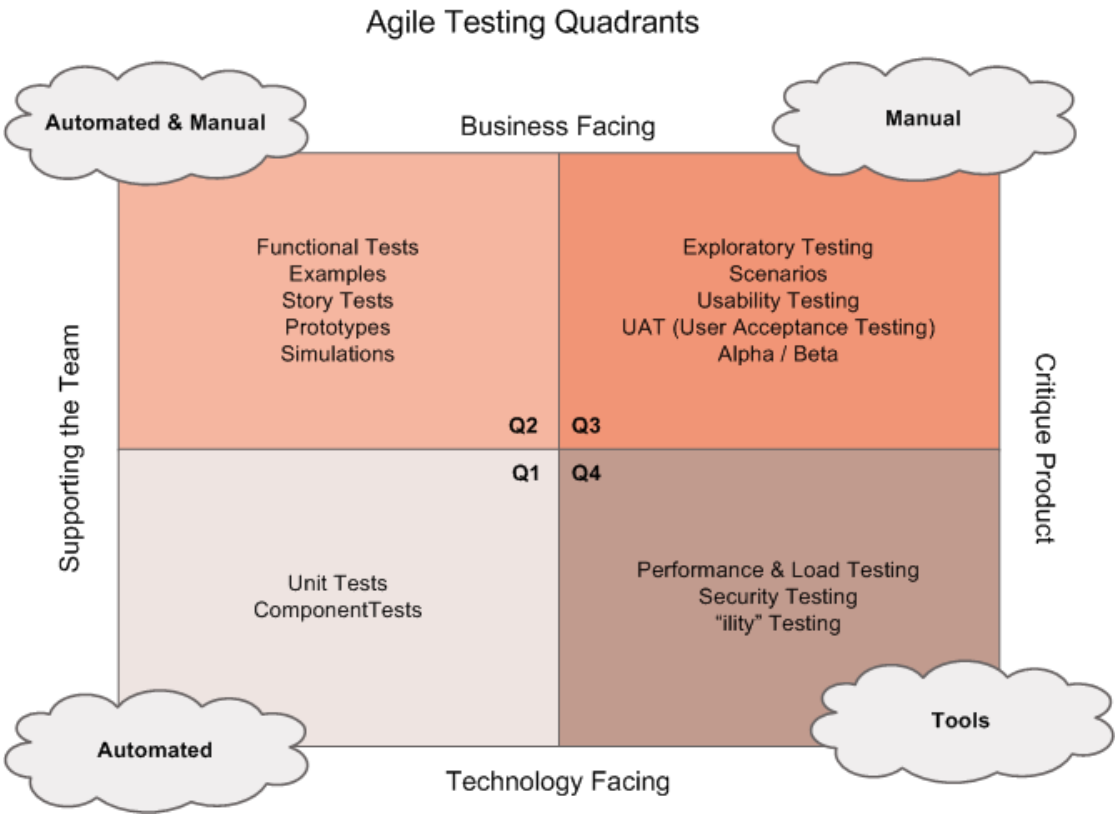

Abbildung 1: *Agile Testing Quadrants* nach Lisa Crispin (`https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/`)

4.2 Teststrategie

Wie im Vorsemester im Modul *Software Testing* eingeübt, sollen die *Agile Test Quadrants* (Abbildung 1, Seite 18) als Grundlage zur Erarbeitung einer Teststrategie dienen (Crispin & Gregory, 2008, p. 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, p. 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, p. 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.[13] 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,

[13] 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, p. 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[14], 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, p. 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[15] wiederum sehr mächtige Werkzeuge out-of-the-box an (Donovan & Kernighan, 2015, p. 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,

[14] `go test -bench=[pattern]`
[15] `go test -cpuprofile/-memprofile/-blockprofile=[Ausgabedatei]`

muss dokumentiert sein. Hierzu gibt es einerseits das Tool `go vet`[16], das zum Lieferumfang von Go gehört, und potenzielle Fehler im Code meldet. Das Zusatztool `golint`[17] 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.[18]

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

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

[18] 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.

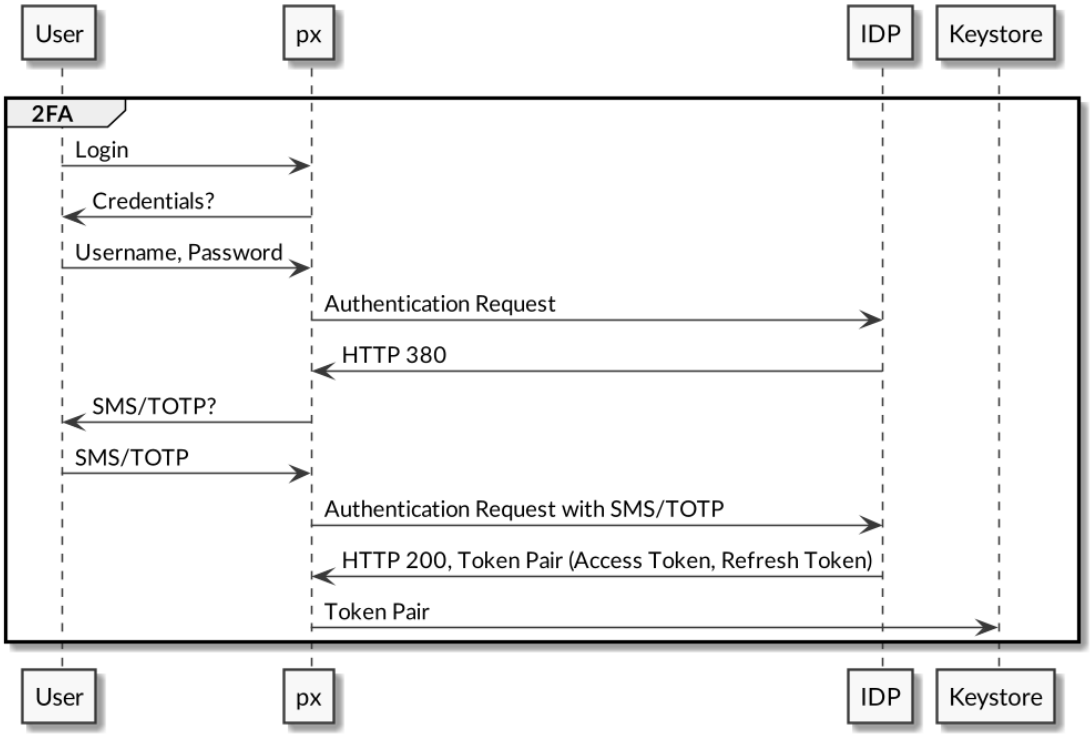

Abbildung 2: Sequenzdiagramm: Der Ablauf der Zwei-Faktor-Authentifizierung

5 Realisierung

TODO: Libraries, Umgebung, Architektur

5.1 Zwei-Faktor-Authentifizierung

5.2 Retry-Mechanismus

5.3 Token-Store

TODO: sicher und unsicher, Datenstruktur

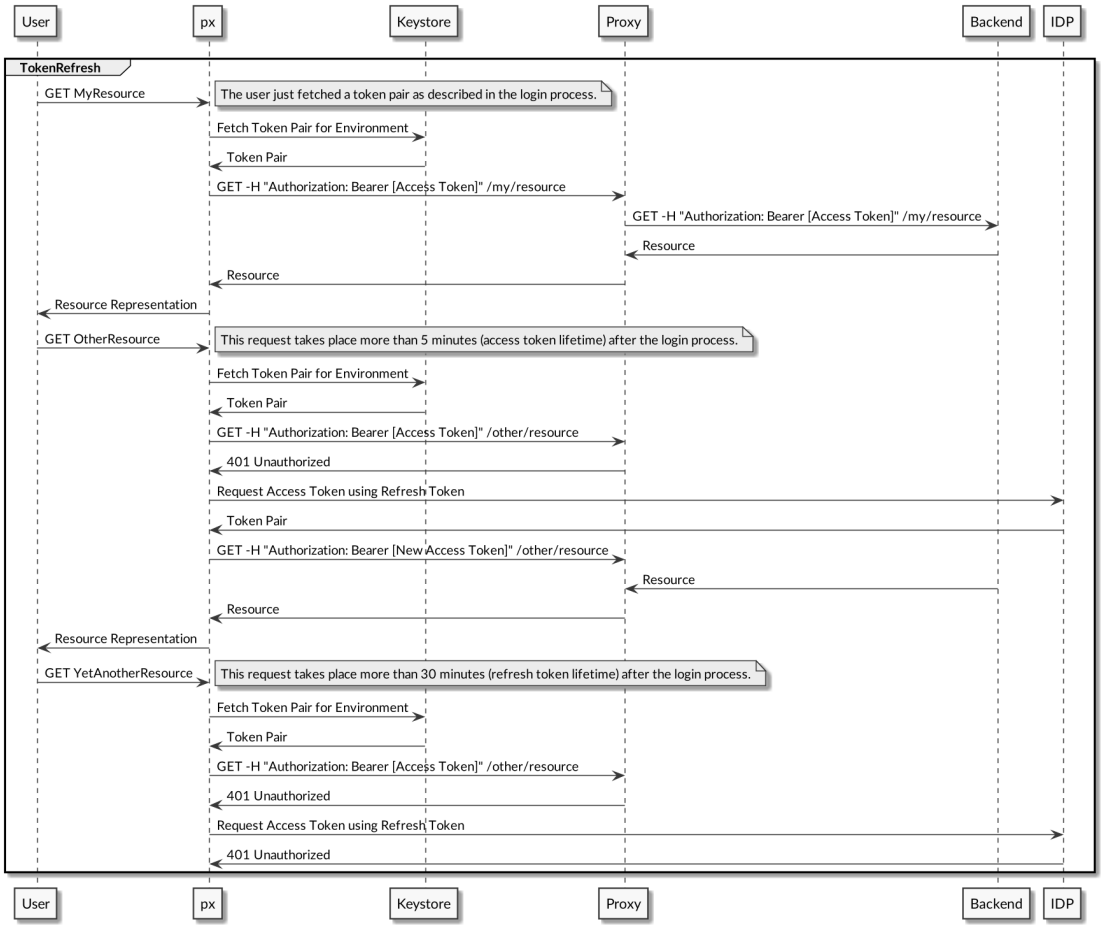

Abbildung 3: Sequenzdiagramm: Der für den Benutzer transparente Retry-Mechanismus mit aktualisiertem Access Token

6 Evaluation und Validierung

Das vorliegende Projekt entstammt dem PEAX-Ideation-Prozess, worin sich Entwicklungs- und Validierungsphase abwechseln.

6.1 Rückmeldungen von Entwicklern

Nach jedem Sprint wird ein Inkrement an die Entwickler ausgeliefert (siehe Abschnitt 4.1 *Vorgehen*, Seite 17). Die Rückmeldungen werden hier gesammelt und kommentiert – und fliessen in 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, p. 3). Siehe dazu auch *Rule of Silence* (Raymond, 2004, p. 20) und *Silence is Golden* (Gancarz, 1995, p. 111).[19]
  - `stderr` wird nicht nur als Ausgabekanal für Fehlermeldungen verwendet, sondern für Meldungen allgemein. Für Vollzugsmeldungen wäre `stderr` vorzuziehen.
  - Da `stderr` in `px` bisher grundsätzlich für Fehlermeldungen verwendet wird, sollen Erfolgsmeldungen über ein zusätzliches Flag `-verbose`/`-v` aktiviert werden müssen.
  - Bei anderen Anwendungsfällen signalisiert die Ausgabe des Payloads auf `stdout` den Erfolg der Operation. Beim Dokument-Upload besteht dieser beispielsweise aus der generierten UUID des hochgeladenen Dokuments.

[19] 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, p. 69)

- Patrick Roos sieht die Möglichkeit, px auch zur Handhabung der *Vault Secrets*[20] (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, p. 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.

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

7 Ausblick

TODO: OpenSource, auf dem Laufenden halten, unterschiedliche API-Versionen, wie weiter?

8 Anhang

Im Anhang sind verschiedene Teile der Dokumentation gesammelt, die den Lesefluss im Hauptteil unnötig erschweren würden, und eher als Referenz denn zum Durchlesen gedacht sind.

Weiter sind hier im Anhang Dokumente aufgelistet, die mit der Arbeit im Zusammenhang stehen, aber nicht physisch mit diesem Dokument verbunden sein sollen.

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 4, Seite 29) 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.
- Frontend (Backoffice): Ein Web-GUI für administrative Tätigkeiten zum internen Gebrauch.

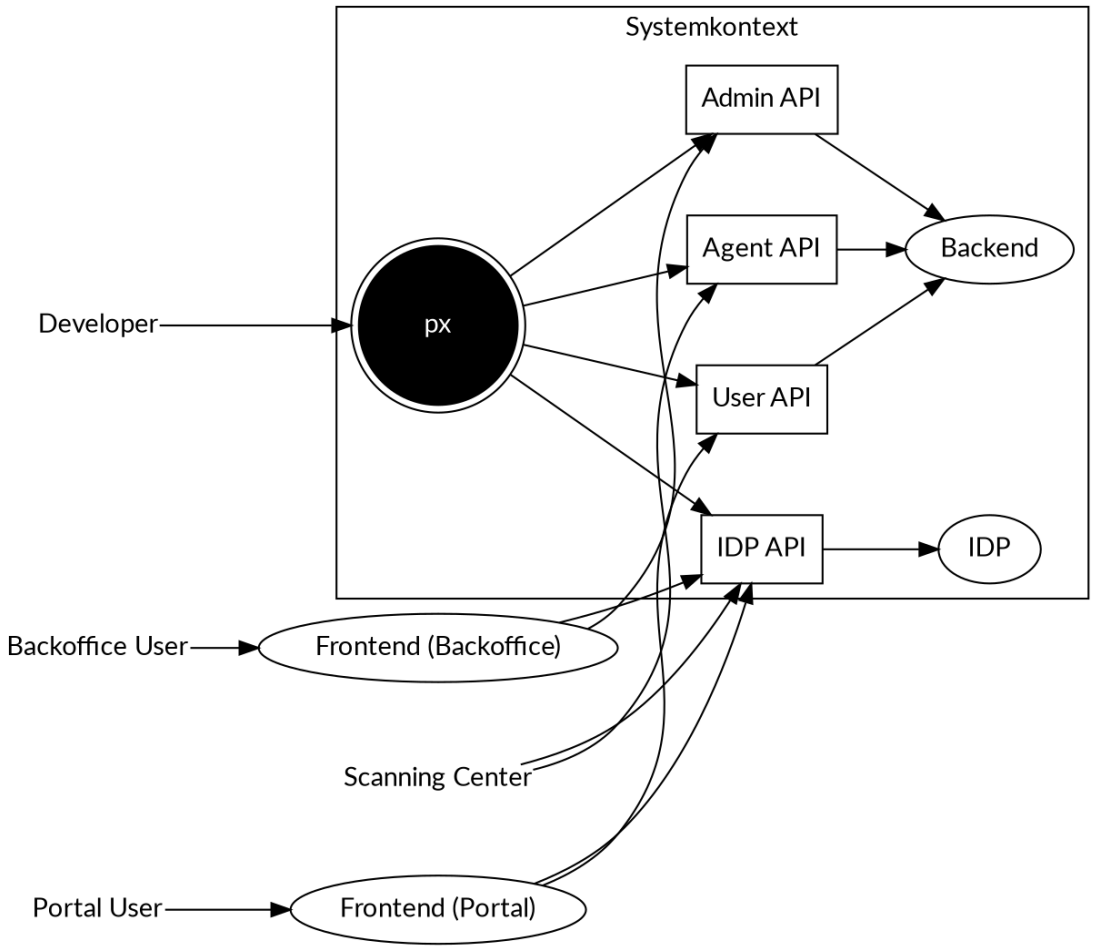

Abbildung 4: Kontextdiagramm: px als der Gegenstand der Arbeit innerhalb des Systemkontexts

- 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 (Auth-N/AuthZ)

8.2 Technologie-Evaluation

- Vorgabe: PEAX API (RESTful)

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[21] 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.

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[22], ein Thumbnailer[23] sowie zahlreiche Utilities[24] (viele darunter als HTTP-Clients) in Go entwickelt. Dabei wurden folgende Vor- und Nachteile ermittelt:

- Vorteile
  - aufgrund weniger Keywords und Features einfach zu lernen

[21] `https://bellard.org/quickjs/`
[22] `https://github.com/patrickbucher/getting-to-philosophy`
[23] `https://github.com/patrickbucher/thumbnailer`
[24] `https://github.com/patrickbucher/go-scratch`

  - 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[25]

- Nachteile
  - keine Features wie Generics, Exceptions und `filter/map/reduce`
  - Binaries fallen relativ gross aus[26]
  - 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, p. 12). Nach selbständiger Beschäftigung mit dieser Programmiersprache im Sommer können (teils ergänzend) folgende Vor- und Nachteile genannt werden:

- Vorteile
  - viele moderne Features (Generics, `filter/map/reduce`)
  - hervorragendes Typsystem
  - gutes und ausgereiftes Tooling
  - weder manuelles Memory-Management noch Garbage Collector nötig

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

  - 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[27]

- Nachteile
  - hohe Einstiegshürde und lange Einarbeitungszeit
  - Cross-Compilation benötigt Zusatztools
  - noch keine praxisnahe Erfahrung damit gesammelt
  - aufgrund schlanker Standard Library auf viele Dependencies angewiesen

8.2.4 Entscheidung

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[28], 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.

[27] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/rust-go.html`

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

8.3 Libraries

TODO: JWT, Keystore, Password Input

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.

Backlog Das Backlog (`Anhang/Backlog.pdf`) enthält die einzelnen User Stories, die Sprint-Planung, Umsetzungsnotizen zu einzelnen User Stories und Testprotokolle dazu.

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

Meilensteinbericht 2 Der zweiten Meilensteinbericht (`Anhang/Meilensteinbericht-2.pdf`) umfasst die ersten beiden Sprints.

Meilensteinbericht 3 `Anhang/Meilensteinbericht-3.pdf`

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.

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.

Apple Support. (2019). *Use zsh as the default shell on your Mac*. https://support.apple.com/en-us/HT208050. (Abgerufen am 26.10.2019)

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)

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.

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

Tabellenverzeichnis

Verzeichnis der Codebeispiele