px: PEAX Command Line Client

Wirtschaftsprojekt, Herbstsemester 2019

Patrick Bucher

20. Oktober 2019

Abstract

Lorem ipsum dolor sit amet (Raymond, 2004, p. 99), consetetur sadipscing elitr (Martin, 2018, p. 101), sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua. At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet (Laboon, 2017, p. 99). Lorem ipsum dolor sit amet, consetetur sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut labore et dolore magna aliquyam erat, sed diam voluptua (Gancarz, 1995, p. 88). At vero eos et accusam et justo duo dolores et ea rebum. Stet clita kasd gubergren, no sea takimata sanctus est Lorem ipsum dolor sit amet (Donovan & Kernighan, 2015, p. 23).

Inhaltsverzeichnis

1 Problemstellung

1.1 Analyse des Projektauftrags

Die Umgebung der PEAX API einerseits

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.

1.1.2 HTTP-Methoden

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

- `GET`
- `HEAD`
- `POST`
- `PUT`
- `DELETE`
- `OPTIONS`
- `PATCH` (Dusseault & Snell, 2010)

1.1.3 HTTP Status-Codes

Eine Antwort auf eine HTTP-Anfrage enthält jeweils einen Status-Code (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.[I]
- `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.

[I] 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.

1.1.4 Umgebungen

1.1.5 Arten von Parametern

1.1.6 Benutzer

1.1.7 Betriebssysteme

1.1.8 Shells

1.2 Ausgangslage und Vorleistungen

Das Projekt `px` wurde bereits am 11. Juni 2019 auf dem GitLab von PEAX erstellt[2]. 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 Builds 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`[3]) 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.

`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

[2] `https://gitlab.peax.ch/peax3/px`
[3] `https://semver.org/`

das eingegebene Passwort nicht angezeigt, was mit einem externen SSH-Terminal-Modul[4] 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[5].

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.

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

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

2 Stand der Praxis

2.1 Ansprechen der PEAX API

2.2 Kommandozeilenprogramme

3 Ideen und Konzepte

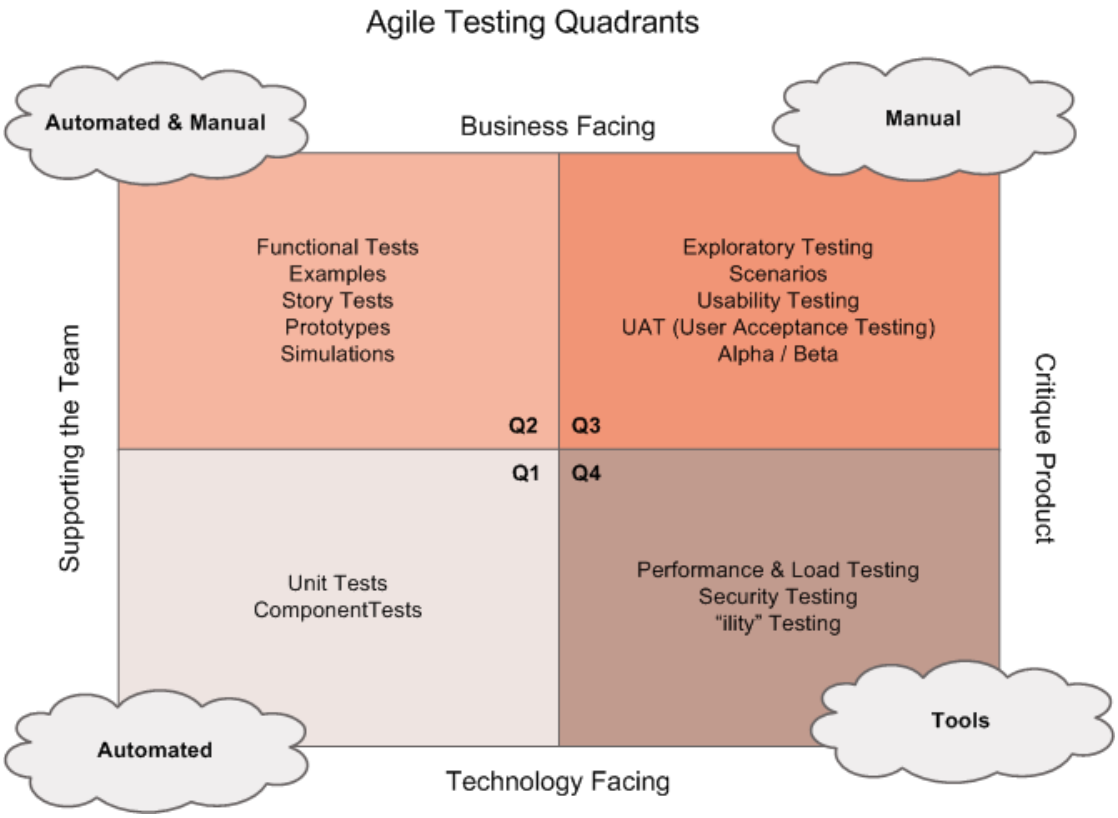

Abbildung 1: *Agile Testing Quadrants* nach Lisa Crispin (`https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/`)

4 Methoden

4.1 Teststrategie

Wie im Vorsemester im Modul *Software Testing* eingeübt, sollen die *Agile Test Quadrants* (Abbildung 1, Seite 10) als Grundlage zur Erarbeitung einer Teststrategie dienen (Crispin & Gregory, 2008, p. 242).

Für die einzelnen Quadranten bieten sich für das vorliegende Projekte folgende Arten von Tests an:

4.1.1 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.

4.1.2 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.[6] 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.

4.1.3 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.

[6] 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.

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, 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.

4.1.4 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[7], 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[8] 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,

[7] `go test -bench=[pattern]`
[8] `go test -cpuprofile/-memprofile/-blockprofile=[Ausgabedatei]`

muss dokumentiert sein. Hierzu gibt es einerseits das Tool `go vet`[9], das zum Lieferumfang von Go gehört, und potenzielle Fehler im Code meldet. Das Zusatztool `golint`[10] 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.[11]

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

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

[11] 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

6 Evaluation und Validierung

7 Ausblick

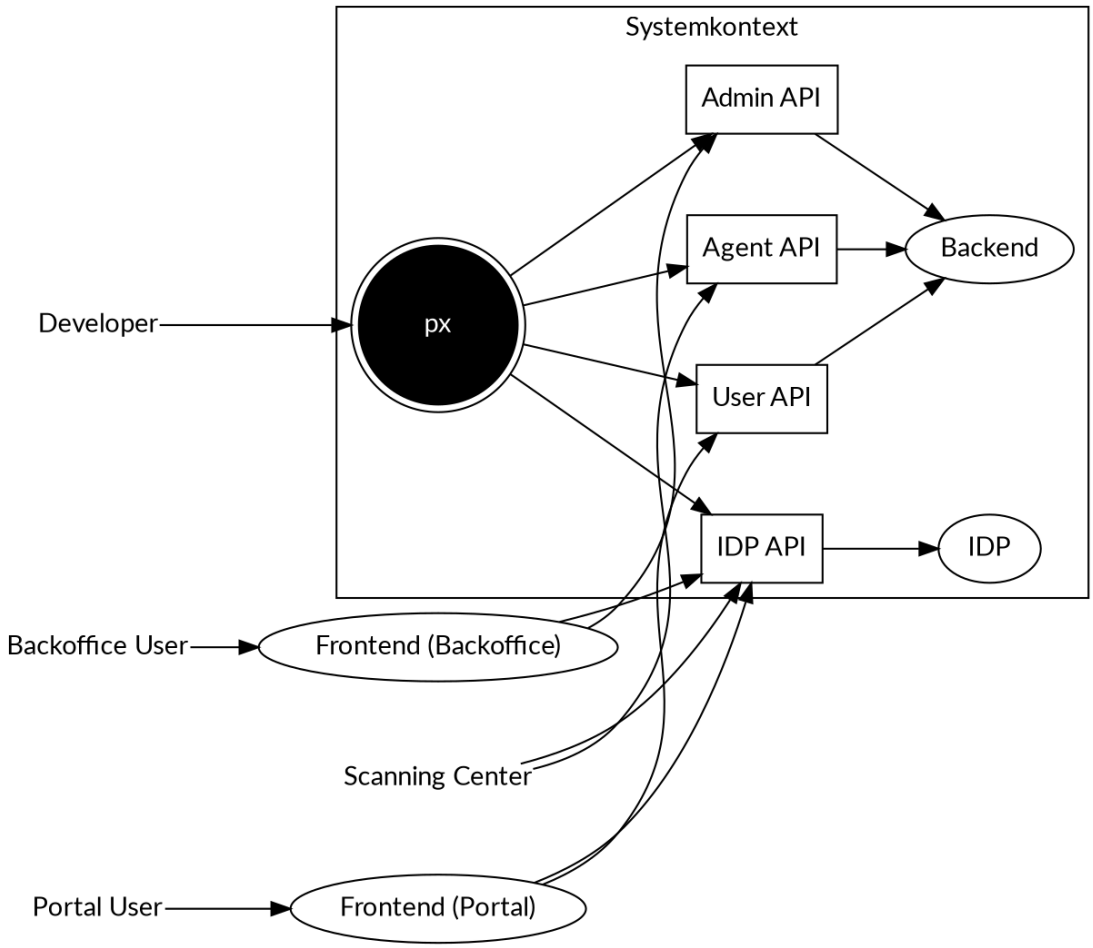

Abbildung 2: Kontextdiagramm: px als der Gegenstand der Arbeit innerhalb des Systemkontexts

8 Anhang

8.1 Systemspezifikation

8.1.1 Systemkontext

Im Kontextdiagramm (Abbildung 2, Seite 18) wird die zu entwickelnde Komponente px im Systemkontext von PEAX dargestellt. Andere Komponenten sind als Ellipsen, Schnittstellen als Rechtecke dargestellt. Ausserhalb vom Systemkonzexts 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.
- 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)

8.2 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, mac OS, 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[12] 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.

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

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[13], ein Thumbnailer[14] sowie zahlreiche Utilities[15] (viele darunter als HTTP-Clients) in Go entwickelt. Dabei wurden folgende Vor- und Nachteile ermittelt:

- Vorteile
  - 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[16]
- Nachteile
  - fehlende Features wie Generics und `filter`/`map`/`reduce`
  - Binaries fallen relativ gross aus[17]
  - Error-Handling aufwändig und teils repetitiv

[13] `https://github.com/patrickbucher/getting-to-philosophy`
[14] `https://github.com/patrickbucher/thumbnailer`
[15] `https://github.com/patrickbucher/go-scratch`
[16] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go.html`
[17] `https://golang.org/doc/faq#Why_is_my_trivial_program_such_a_large_binary`

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
  - 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[18]
- 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 fallen jedoch einige davon kaum ins Gewicht (etwa Performance und Zero-Cost Abstractions). Hier fallen die Vorzüge von Go (umfassende Standard Library, Cross-Compilation) wesentlich stärker ins Gewicht.

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

Gerade die absichtlich schlank gehaltene Standard Library von Rust, die etwa zur Generierung von Zufallszahlen bereits externe Abhängigkeiten erfordert[19], dürfte sich im vorliegenden Projektrahmen negativ auswirken, zumal die Evaluation verschiedener Libraries einen sehr hohen Zusatzaufwand zur Folge haben.

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

TODO: JWT, Keystore, Password Input

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

Literatur

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

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

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

Martin, R. C. (2018). *Clean Architecture*. Prentice Hall.

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

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Codebeispiele