px: PEAX Command Line Client

Wirtschaftsprojekt, Herbstsemester 2019

Patrick Bucher

22. September 2019

Abstract

Lorem ipsum dolor sit amet, consetetur sadipscing elitr, 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. 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, Mike, 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, Alan A. A. and Kernighan, Brian W., 2015, p. 23).

Inhaltsverzeichnis

1 Technologie-Evaluation

- Vorgabe: PEAX API (RESTful)

1.1 Programmiersprache

Anforderungen:

Installation Die Software soll sich einfach installieren lassen.

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

Plattformen Die Software soll auf allen gängigen 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 erfordert die lokale Installation einer JRE in der richtigen Version. 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.

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 gibt es mit QuickJS 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.

In die engere Auswahl kommen Go und Rust, da der Autor dieser Arbeit mit diesen Programmiersprachen bereits Erfahrungen im Studium machen konnte. (Mit C++ noch keine Erfahrungen gemacht. Mit C Erfahrungen gemacht, wodurch es als sehr aufwändig erscheint, einen HTTP-Client zu schreiben.)

1.1.1 Go

- einfach zu lernen (wenige Keywords und Features), dafür keine Features wie Generics und map/filter/reduce
- gutes und übersichtliches Tooling
- Cross-Compilation ohne Zusatztools möglich
- Kompilierung zu statischen Binaries, die jedoch recht gross ausfallen (Prototyp: ca. 4 MB)
- Kompilierung extrem schnell
- umfassende und qualitativ hochwertige Standard Library, inkl. HTTP-Library
- persönlich bereits viel damit gearbeitet, positive Erfahrungen damit gemacht
- Error Handling aufwändig, führt aber zu sehr solidem, wenn auch repetitivem Code
- vergleichbare Software (`oc`: OpenShift Command Line Client, `docker`: Docker Command Line Client) ist ebenfalls in Go geschrieben und bei uns täglich erfolgreich im Einsatz
- fügt sich sehr gut in UNIX-Philosophie ein (Tooling, Libraries)
- Einfaches Interface für nebenläufige Programmierung (Goroutines und Channels)
- Performance im Bereich von Java, jedoch tieferer Memory-Footprint

1.2 Rust

- viele Features, dafür aber schwer zu lernen (Lifetimes, Borrowing, siehe PCP)
- gutes und übersichtliches Tooling
- Kompilierung in statische Binaries, die relativ schlank ausfallen
- Kompilierung eher langsam
- Cross-Compilation benötigt Zusatztools
- wenig Erfahrung damit gesammelt
- Standard Library bewusst schlank gehalten, dafür viele externe Libraries benötigt
- extrem ausdrucksstarke Sprache mit starkem Typsystem
- diverse Rust-Utils erfolgreich persönlich im Einsatz (rg, bat, hexyl, battop)
- erstklassige Performance (im Bereich von C/C++)

1.3 Libraries

2 Einstiegsbeispiel

```
package main

import (
        "fmt"
        "log"
        "os"

        "gopl.io/ch05/links"
)

// breadthFirst calls f for each item in the worklist.
// Any items returned by f are added to the worklist.
// f is called at most once for each item.
func breadthFirst(f func(item string) []string, worklist []string) {
        seen := make(map[string]bool)
        for len(worklist) > 0 {
                items := worklist
                worklist = nil
                for _, item := range items {
                        if !seen[item] {
                                seen[item] = true
                                worklist = append(worklist, f(item)...)
                        }
                }
        }
}

func crawl(url string) []string {
        fmt.Println(url)
        list, err := links.Extract(url)
        if err != nil {
                log.Print(err)
        }
        return list
}

func main() {
```

```
        // Crawl the web breadth-first,
        // starting from the command-line arguments.
        breadthFirst(crawl, os.Args[1:])
}
```

Codebeispiel 1: Some Go Code

Literatur

Donovan, Alan A. A. and Kernighan, Brian W. (2015). *The Go Programming Language*. Addison-Wesley.

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

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Codebeispiele