Auswahl Programmiersprache und Framework für BFFH

Dieses Dokument basiert auf dem ADR-Commit https://gitlab.com/fabinfra/fabaccess/bffh/-/commit/a62a5678dc35cb32c899483cfab48ce218694d11 und wurde ins Deutsche übersetzt.

Der technische Hintergrund: Entscheidung über die zu verwendende Programmiersprache und das Framework für BFFH / den Backend-Code von FabAccess

Kontext und Problemstellung

Programmiersprachen-Diskussionen sind das perfekte Thema für einen Fahrradausflug. Unabhängig davon muss irgendwann eine Entscheidung getroffen werden, damit die Leute anfangen können, Code zu schreiben und allgemein nützlich zu sein. Da FabAccess als Projekt mit mehreren potenziellen Entwicklern und noch mehr Beteiligten, die am Erfolg der Software interessiert sind, gestartet wurde, war diese Diskussion besonders pikant. Die entsprechenden Diskussionen fanden im Zeitraum von etwa November 2019 bis März 2020 statt.

Entscheidungstreiber

- Verfügbare Entwickler, sowohl kurz- als auch mittelfristig, sollte das Projekt ein erfolgreiches Standbein für deutsche/europäische/irdische Makerspaces werden
- Einstiegshürde für neue/zusätzliche Entwickler.
- Tooling-Unterstützung, insbesondere für Nicht-Entwickler (Dokumentation, einfache Selbstkompilierung, ...)
- Bibliotheksunterstützung / Ökosystem in der Problemdomäne
- Bestehende Projekte, die erweitert werden könnten, um das Rad nicht noch einmal zu erfinden
- Sprachliche Merkmale

Geprüfte Optionen

- Python
- TypeScript

- Rust
- Erlang
- Ruby
- Haskell

Entscheidungsergebnis

Gewählte Option: „Rust“, weil das Projekt schließlich größtenteils von FabInfra entwickelt wird, dessen Kosten-Nutzen-Analyse der Typensicherheit Vorrang vor der Entwicklungsgeschwindigkeit einräumt.

Positive Konsequenzen

- Die Typsicherheit von Rust ist im Vergleich zu allen anderen Optionen mit Ausnahme von Haskell sehr hoch
- Eine einzige statisch gelinkte Binärdatei mit sehr wenigen Abhängigkeiten macht die Bereitstellung trivial
- Kompilierte Software nahe an Baremetal ermöglicht den Betrieb des Servers auf leistungsschwacher Hardware wie Raspberry Pis, selbst bei mittelgroßen Einsätzen.

Negative Konsequenzen

- Die Geschwindigkeit der Entwicklung leidet kurzfristig. Rust ist eine Sprache mit niedrigerem Niveau als alle anderen evaluierten Optionen und erfordert mehr Denkarbeit und Entwicklerzeit, um Funktionen auf hohem Niveau zu implementieren.
- Der verfügbare Entwicklerpool ist klein und teuer. Rust ist keine gute Einstiegssprache und für Entwickler, die keine Erfahrung mit C++ haben, vergleichsweise schwer zu erlernen. Es können zwar Entwickler eingestellt werden, aber die Gehaltsvorstellungen von Rust-Entwicklern sind im Vergleich zu Python/Java hoch.
- Kleines Ökosystem / wenige Bibliotheken. Rust ist eine sehr junge Programmiersprache und verfügt, obwohl sie seit mehreren Jahren als die beliebteste Sprache auf stackoverflow.com eingestuft wird und kommerziell genutzt wird, über weniger Bibliotheken als die meisten anderen Optionen. Die Unterstützung für z.B. LDAP ist nicht ausgereift. Aber die Problemdomäne von FabAccess liegt in dem Bereich, für den Rust die meiste Unterstützung hat.

Vor- und Nachteile der Optionen

Python

- Gut, weil großer verfügbarer Entwicklerpool
- Gut, weil ein großer Teil der an der Entwicklung interessierten Entwickler Erfahrung mit Python hat
- Gut, weil Python leicht zu erlernen ist und eine gute Einstiegssprache darstellt
- Gut, weil Python über umfangreiche Bibliotheksunterstützung und ein auf Automatisierung ausgerichtetes Ökosystem verfügt
- Schlecht, weil vergleichsweise langsam. Langsamste oder zweitlangsamste Option in Betracht.
- Schlecht, weil die hochdynamische Typisierung und die mittelmäßige statische Analyse dazu führen können, dass Edgecase-Fehler lange Zeit verborgen bleiben
- Schlecht, weil Python wenig Kontrolle über den Absturzprozess bietet. Das Abfangen von Exceptions ist sehr grob.

TypeScript

- Gut, weil stärkere Typisierung als bei Python und Ruby
- Schlecht, weil interpretierte Sprache und zwar schneller als Ruby/Python, aber immer noch viel langsamer als die kompilierten Optionen.
- Schlecht, weil das Node.js Ökosystem groß ist, aber nicht so sehr auf Automatisierung auf niedriger Ebene ausgerichtet ist.

Rust

- Gut, weil Rust ein gutes Typensystem hat, das es erlaubt, viele Fehler und Abstürze durch die in den Compiler eingebaute statische Analyse zu verhindern.
- Gut, weil Rust-Code sehr effizient ist.
- Gut, weil viele Leute sehr daran interessiert sind, Rust zu lernen und ihnen nur ein Grund dafür fehlt.
- Schlecht, weil die Geschwindigkeit der Entwicklung langsamer ist als bei den meisten anderen Optionen.
- Schlecht, weil nur ein Entwickler in der Gruppe Erfahrung mit Rust hat.
- Schlecht, weil der verfügbare Entwicklerpool klein ist.

Erlang

- Gut, weil speziell für hochverfügbare, absturzsichere Software entwickelt
- Gut, weil es ein extrem feinkörniges Absturzsystem mit gekapselten Prozessen bietet, die sich nicht gegenseitig ausschalten können

- Gut, weil das Ökosystem und die Bibliotheken von Erlang generell auf die Problemdomäne von FabAccess ausgerichtet sind
- Gut, weil Erlang im Vergleich zu TypeScript, Pyton und Ruby eine umfangreiche statische Analyse ermöglicht
- Schlecht, weil Erlang ein schlechteres Typsystem hat als Rust und Haskell.
- Schlecht, weil nur ein Entwickler in der Gruppe Erfahrung mit Erlang hat.
- Schlecht, weil der Entwicklerpool der zweitkleinste ist.

Haskell

- Gut, weil das stärkste Typensystem viele Fehler und Abstürze durch die im Compiler integrierte statische Analyse verhindern kann
- Schlecht, weil nur ein Entwickler Erfahrung mit Haskell hat.
- Schlecht, weil der Entwicklerpool am kleinsten ist.

Links

- Die Diskussionen sind dokumentiert in:
  - Protokoll Jit.si-Konferenz 09.12.19
  - Pad "Roseguarden"
  - Pad BF²H "Grundlegendes"
  - JitSi Call 18.02

Version #1
Erstellt: 2024-12-27 00:09:38 CET von Mario Voigt (Stadtfabrikanten e.V.)
Zuletzt aktualisiert: 2025-02-20 14:02:51 CET von Mario Voigt (Stadtfabrikanten e.V.)