SPENGERGASSE

Diploma Thesis

title

Joachim GRÜNEIS, Klaus UNGER

Version 1.0 - 2018-06-18

Table of content

Colophon

Spengergasse Press, Vienna

Schuljahr 2018/19

Datum:	**übernommen von:**

Table 1. Abgabevermerk

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche erkenntlich gemacht habe.

Wien, am 08.04.2019	**VerfasserInnen**
	Florian FREIMÜLLER

Themenstellung: Ein Vergleich von JVM Sprachen im Umgang mit modernen Programmierschnittstellen

Florian Freimüller <fre18149@spengergasse.at>

Abstract

In diesem Paper wird mithilfe von eigens ausgewählten Bewertungskriterien bewertet, welche JVM Sprache wie gut geeignet ist, um verschiedene Programmierschnittstellen anzusteuern.

Bewertungskriterien

Um die Schnittstellen so gut wie möglich bewerten zu können, wird eine Beurteilungstabelle erstellt. Pro Kriterium können maximal 5 Punkte und minimal 0 Punkte vergeben werden. Diese Tabelle setzt sich aus folgenden Kriterien zusammen:

Lesbarkeit des Codes

Ein wichtiger Aspekt bei der Beurteilung ist, wie lesbar der Code ist, wenn die Schnittstelle angesteuert wird. Hierbei wird vor ein Augenmerk darauf gelegt, ob der Code durch das Ansprechen der Schnittstelle unlesbar wird oder nicht.

Dokumentation

Bei der Dokumentation wird beurteilt, ob es eine Schnittstellendokumentation für die jeweilige Sprache gibt. Sollte es eine geben, wird bewertet, wie gut und übersichtlich die Dokumentation gestaltet ist.

Lines of Codes

Je weniger Codezeilen benötigt werden, um ein Beispiel in der jeweiligen Sprache zu programmieren desto mehr Punkte werden hier vergeben. Als Grundlage wird der Code von der Sprache genommen, die am wenigsten Zeilen für das jeweilige Beispiel benötigt.

Eine Zeile ist:

- Eine Annotation
- Ein Statement (ein Statement über mehrere Zeilen = eine Codezeile)
- Eine Deklaration (einer Klasse, eines Interfaces, einer Variable etc.)

Nicht zu Codezeilen zählt folgendes:

- Zeilen, die **nur** eine Klammer schließen/öffnen
- Leere Zeilen

Unterstüzte Paradigmen

Bei dem Kriterium "Unterstützte Paradigmen" wird darauf geachtet, dass alle Programmierparadigmen der jeweiligen Sprache (falls eine Sprache mehrere unterstützen sollte) unterstützt werden. Sollte dies nicht der Fall sein, gibt es hier einen Punkteabzug.

Beispielstabelle

Sprache	**Java**	**Kotlin**	**Groovy**	**Scala**	**Clojure**
Lesbarkeit	4	5	3	4	5
Dokumentation	5	5	5	4	2
Lines of Code	2	4	4	3	5
Unterstützte Paradigmen	5	5	5	5	5
Ergebnis	21	24	22	20	20

Figure 1. Beispielsbeurteilungstabelle

Auswahl der JVM Sprachen

Um möglichst viele Vergleichswerte zu haben, werden die Schnittstellen in sechs verschiedenen JVM Sprachen verglichen.

Java

Java ist eine objektorientierte Programmiersprache und wurde im Jahr 1995 von James Gosling veröffentlicht und wird bis heute in sehr vielen Bereichen verwendet. Da Java eine general purpose language ist und Java dank der JVM (Java virtual machine) plattformunabhängig ist, kann Java für sehr viele Anwendungsimplementierungen eingesetzt werden, angefangen von simplen Konsolenprogrammen bis hin zu Anwendungen auf Bordcomputern von Automobilen.

Kotlin

Die Programmiersprache Kotlin wurde von der Firma JetBrains entwickelt und im Jahre 2011 veröffentlicht. Wichtig bei der Erstellung von Kotlin war sowohl, dass Kotlin problemlos mit Java gemeinsam verwendet werden kann als auch, dass der in Kotlin geschrieben Code eleganter und effizienter ist als der equivalente Java Code. Hauptsächlich wird Kotlin für Android Applikationen verwendet, allerdings ist es ebenso möglich, die Sprache für Web-Applikationen oder auch native Applikationen zu verwenden, da Kotlin eine general purpose language ist.

Groovy

Scala

Clojure

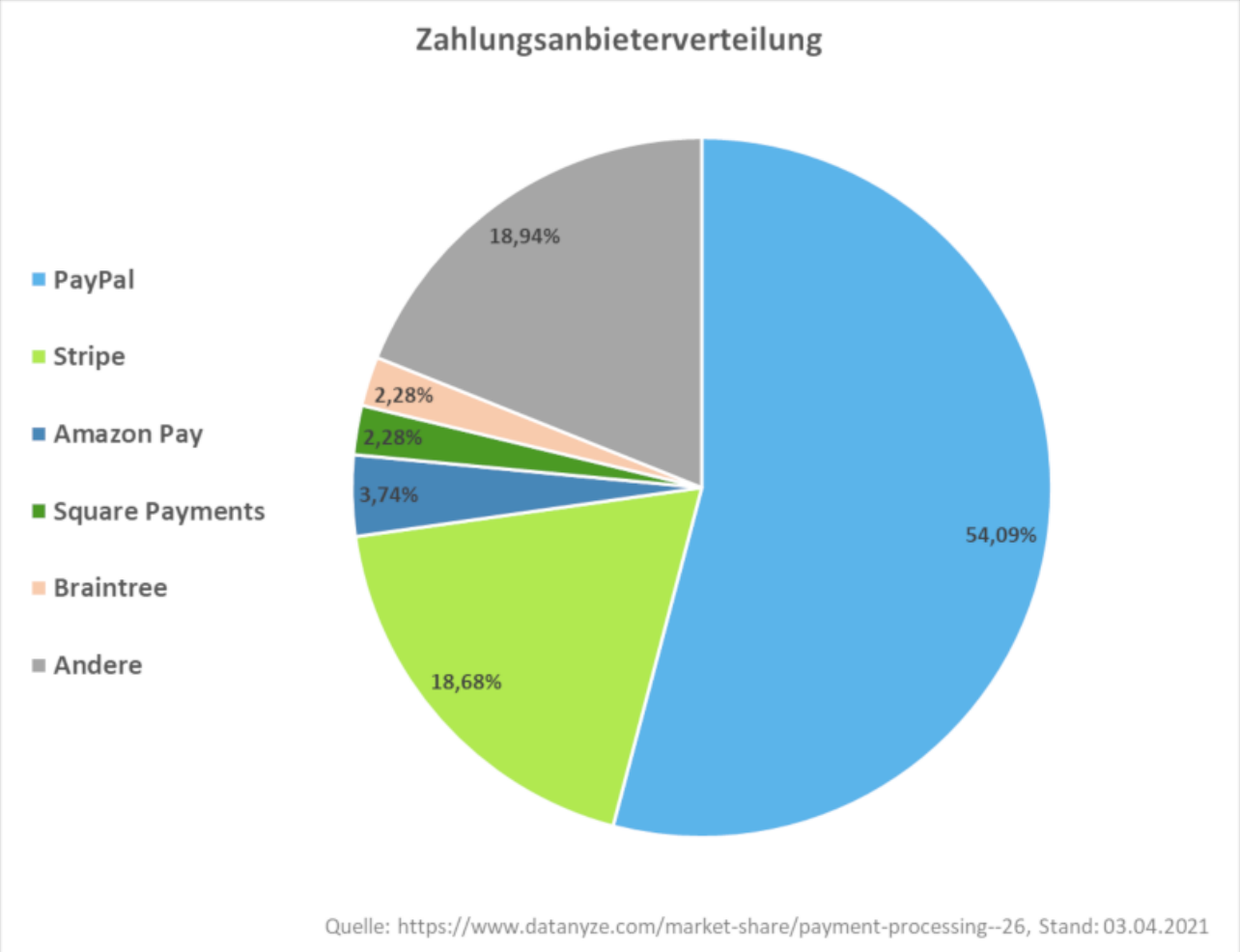
Auswahl der Schnittstellen

Bei den behandelten Schnittstellen wurde darauf geachtet, dass diese häufig Anwendung finden und es daher auch einen Grund für die Entwickler dieser Schnittstellen gibt, diese Schnittstellen so kompatibel wie möglich zu gestalten.

Stripe API (über Bibliotheken)

Stripe ist ein Zahlungsanbieter, der im Diplomprojekt verwendet wird.

Im nachfolgenden Diagramm ist der Marktanteil der größten Zahlungsanbieter zu sehen, in dem Stripe den zweiten Platz belegt:



REST APIs [Clients]

Da heutzutage sehr viele Services als REST-API zur Verfügung gestellt werden ist es oftmals notwendig, REST-APIs mithilfe von Clients anzusprechen. Dies kann sowohl in Mobilapplikationen der Fall sein als auch in serverseitigen Anwendungen.

Streaming API

In Java gibt es die Streaming-API, in diesem Kapitel wird verglichen, welche Alternativen oder nativen Sprachfeatures es in den anderen Sprachen gibt.

Persistence API

Wenn Daten in einer Datenbank gespeichert werden wird, kann man eine Persistence API verwenden. Hier wird der Fokus auf Lösungen für die jeweiligen Sprachen gelegt.

Android API

Die Android API wird verwendet, um auf Funktionen des Android Systems zuzugreifen, folglich kommt diese Schnittstelle hauptsächlich bei Applikationen für Mobilgeräte zum Einsatz.

E-Mail APIs

E-Mail APIs werden vor allem in Backend Applikationen benötigt, um Benutzer*innen Informationen per E-Mail zu senden.

Stripe API

Rest APIs

In allen Sprachen wird die Rest-API von https://reqres.in/ verwendet. Als Code sample wird jeweils ein GET-Request und ein POST-Request abgesendet und das Resultat soll als Objekt soll in einer Variable abgespeichert werden.

Die DTO Klassen werden nicht zur Bewertung herangezogen.

Java

In Java wird die Bibliothek OpenFeign verwendet.

Code Snippet

Um die Rest-API aufzurufen wird ein Client erstellt, der die Funktionen der API deklariert.

```
/* File: UserFeignClient.java */

public interface UserFeignClient {
    @RequestLine("GET /users/{id}")
    GetUser getUser(@Param("id") int id);

    @RequestLine("POST /users")
    @Headers("Content-Type: application/json")
    CreateUser.Response createUser(CreateUser.Request createUser);
}
// Lines: 6
```

Anschließend wird ein Client mithilfe des FeignBuilders erstellt und die Funktionen werden aufgerufen.

```
/* File: Main.java */

    public static void main( String[] args )
    {

        UserFeignClient client = Feign.builder()
                .client(new OkHttpClient())
                .encoder(new GsonEncoder())
                .decoder(new GsonDecoder())
                .target(UserFeignClient.class, "https://reqres.in/api"
);


        GetUser getUserResponse = getUser(client);
        CreateUser.Response createUserResponse =
                createUser(client, new CreateUser.Request("Testuser",
"Programmer")));
    }

    public static GetUser getUser(UserFeignClient client) {
        return client.getUser(2);
    }

    public static CreateUser.Response createUser(UserFeignClient
client, CreateUser.Request request) {
        return client.createUser(request);
    }
// Lines: 8
```

Bewertung

- Lines of Code: 14 Zeilen
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: Die Dokumentation [https://github.com/OpenFeign/feign] ist sehr umfangreich und bietet auch zahlreiche Beispiele zum Einsatz der Bibliothek, außerdem werden verschiedenste Encoder/Decoder vorgestellt, die von der Bibliothek unterstützt werden. → 5/5
- Unterstützte Paradigmen: Die OpenFeign Bibliothek unterstützt sowohl objektorientierte Programmierung als auch funktionale Programmierung (mit CompletableFuture Objekten). → 5/5

Kotlin

In Kotlin wird die OpenFeign Bibliothek verwendet.

Code Snippet

Zuerst wird ein interface mit den beiden Methoden, die anschließend aufgerufen werden, deklariert.

```
/* File: UserFeignClient.kt */

interface UserFeignClient {
    @RequestLine("GET /users/{id}")
    fun getUser(@Param("id") id: Int): GetUser

    @RequestLine("POST /users")
    @Headers("Content-Type: application/json")
    fun createUser(createUser: CreateUserRequest): CreateUserResponse
}

// Lines: 6
```

Nun wird eine Instanz des UserFeignClients mithilfe des FeignBuilders erstellt.

```
/* File: Main.kt */

fun main() {
    val userFeignClient = Feign::builder()
            .client(OkHttpClient())
            .encoder(GsonEncoder())
            .decoder(GsonDecoder())
            .target(UserFeignClient::class.java,
"https://reqres.in/api")

    val getUserResponse = getUser(userFeignClient)
    val createdUser = createUser(userFeignClient, CreateUserRequest(
        name = "Testuser",
        job = "Programmer"
    ))
}

fun getUser(client: UserFeignClient): GetUser {
    return client.getUser(2)
}

fun createUser(client: UserFeignClient, user: CreateUserRequest):
CreateUserResponse {
    return client.createUser(user)
}

// Lines: 8
```

Bewertung

- Lines of Code: 14 Zeilen
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: Die Dokumentation [https://github.com/OpenFeign/feign] ist zwar sehr umfangreich und enthält viele Beispiele, allerdings gibt es leider keine Beispiele für den Umgang mit Kotlin. → 3/5
- Unterstützte Paradigmen: Die OpenFeign Bibliothek unterstützt sowohl objektorientierte Programmierung als auch funktionale Programmierung (mit CompletableFuture Objekten). → 5/5

Groovy

Scala

Clojure

Stream API

Java persistence API (JPA)

Android API

Java Mail API

Fazit

References

https://practicalli.github.io/blog/posts/consuming-apis-with-clojure/

https://reqres.in/

https://www.baeldung.com/intro-to-feign

https://github.com/OpenFeign/feign

Glossary

Index