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.

Die Erwartungshaltung ist, dass die Ergebnisse in allen Sprachen ziemlich gleich sind, da die meisten JVM-Sprachen in der Auswahl syntaktisch ähnlich aufgebaut sind.

Um die Sprachen gut vergleichen zu können, wird für jede Programmierschnittstelle in jeder berücksichtigten JVM Sprache ein Code Snippet geschrieben. Die Aufgabe, die mit dem Snippet erfüllt werden soll, wird im Vorhinein definiert und dieses Snippet wird dann größtenteils zum Vergleich zwischen den Sprachen herangezogen.

# 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 benötigte Paradigmen unterstützt werden (zum Beispiel funktionale Programmierung für reactive Programming). Nur wenn das Paradigma nicht unterstützt wird, gibt es 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 | 16 | 19 | 17 | 16 | 17 |

*Figure 1. Beispielsbeurteilungstabelle*

# Auswahl der JVM Sprachen

Um möglichst viele Vergleichswerte zu haben, werden die Schnittstellen in fünf 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. [1]

**Tiobe Index**

Laut dem Tiobe Index ist Java die zweitpopulärste Sprache (nach C).[2]

## 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. Außerdem werden in Kotlin sowohl objektorientierte als auch funktionale Programmierung unterstüzt.[3][4]

**Tiobe Index**

Kotlin belegt auf dem Tiobe Index den 39ten Platz.

## Groovy

Apache Groovy ist eine OpenSource general purpose language, in der man sowohl Skripte schreiben kann als auch, wie in Java, objektorientierte Applikationen erstellen kann. Außerdem wird auch funktionale Programmierung in Groovy unterstützt. Groovy kann, wie Java und Kotlin auch, in etlichen Anwendungsfällen verwendet werden und man kann alle Java Bibliotheken auch in Groovy verwenden. [5][6]

**Tiobe Index**

Groovy ist auf dem 17ten Platz auf dem Tiobe Index und hat damit seit dem letzten Jahr 31 Plätze gut gemacht.

## Scala

Scala ist eine general purpose langauge mit voller Java Interoperabilität und wurde von Martin Odersky entwickelt. Die erste Version von Scala wurde am 20.01.2004 veröffentlicht und die Sprache wird immer noch weiterentwickelt. Wie bei den vorherigen Sprachen kann Scala für sehr viele Anwendungsfälle verwendet werden. [7] [8]

**Tiobe Index**

Auf dem Tiobe Index belegt Scala den 26ten Platz.

## Clojure

Clojure ist eine funktionale JVM Sprache, die im Jahre 2007 veröffentlicht wurde. Mit Clojure kann man sowohl Anwendungen schreiben als auch Skripte, da die Sprache, wie alle anderen berücksichtigten JVM Sprachen, eine general purpose

language ist, kann man auch mit Clojure Webapplikationen, mobile Applikationen und native Applikationen erstellen. [9]

### Tiobe Index

Clojure ist die in dieser Auswahl am wenigsten populäre Sprache und belegt nur den 60ten Platz im Tiobe Index.


[1] FreeJavaGuide: History of Java programming language, https://www.freejavaguide.com/history.html abgerufen am 25.03.2021

[2] Toibe: TIOBE Index for April 2021, https://www.tiobe.com/tiobe-index/ abgerufen am 05.04.2021

[3] Deshmane, Rohini: Introduction to Kotlin, https://medium.com/@rohinideshmane.21/introduction-to-kotlin-5f39b31610e0 abgerufen am 25.03.2021

[4] Kotlinlang: Calling Java from Kotlin, https://kotlinlang.org/docs/java-interop.html abgerufen am 25.03.2021

[5] Scand: Groovy vs Java: Detailed Comparison and Tips on the Language Choice, https://scand.com/company/blog/groovy-vs-java/ abgerufen am 05.04.2021

[6] Groovy-lang: A multi-faceted language for the Java platform, https://groovy-lang.org/ abgerufen am 05.04.2021

[7] Hicks, Matt: Scala vs. Java: Why Should I Learn Scala?, https://www.toptal.com/scala/why-should-i-learn-scala abgerufen am 05.04.2021

[8] javaTpoint: History of Scala, https://www.javatpoint.com/history-of-scala abgerufen am 05.04.2021

[9] Hickey, Rich: A history of Clojure, https://download.clojure.org/papers/clojure-hopl-iv-final.pdf abgerufen am 05.04.2021

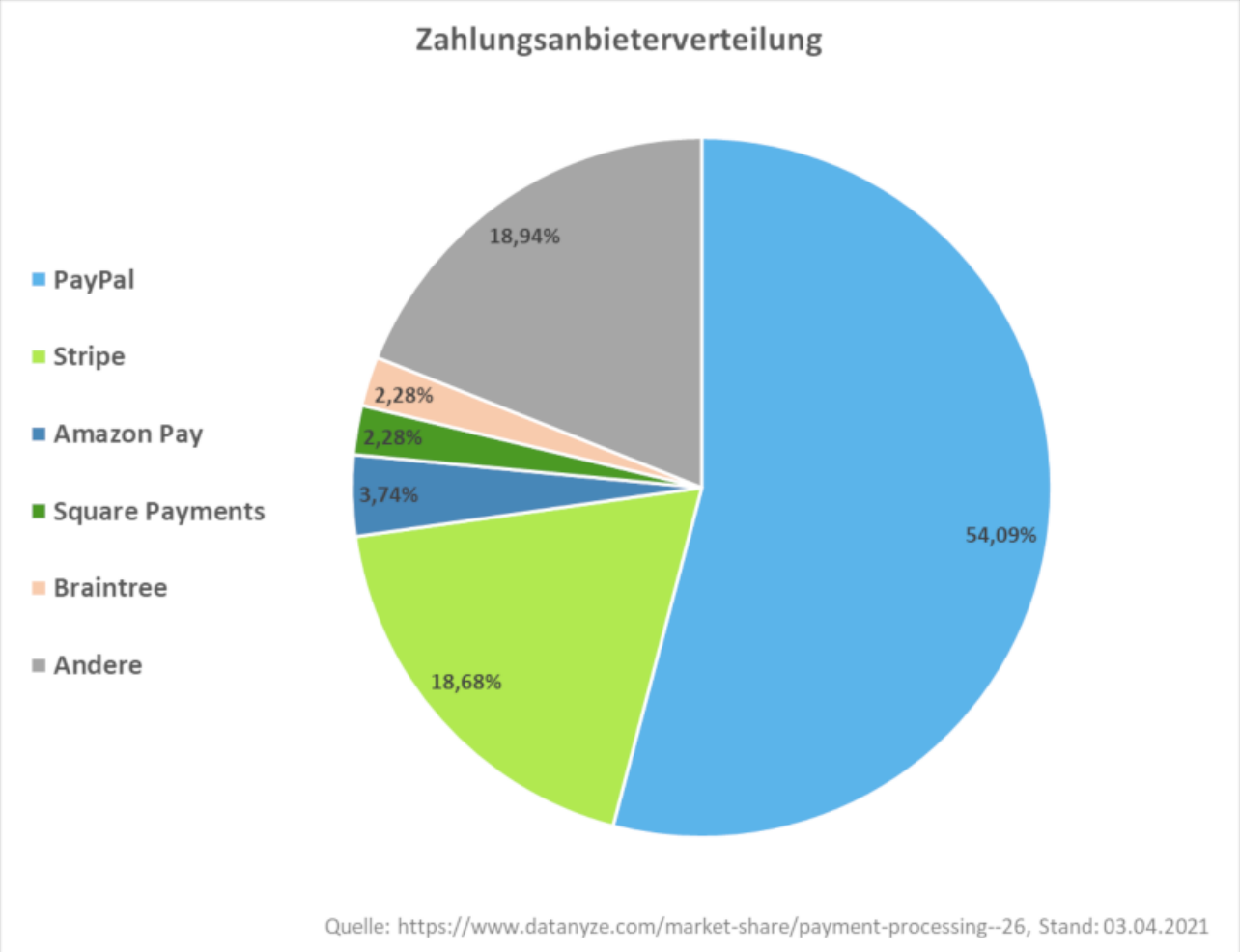
# Auswahl der Schnittstellen

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 und Server-side]

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.

## E-Mail APIs

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

# Stripe API

In allen Sprachen wird

- Eine Zahlung durchgeführt
- Die ID der Zahlung gespeichert
- Der Status der Zahlung mithilfe der ID abgefragt und auf die Konsole ausgegeben

Die verwendete Bibliothek in allen Sprachen ist "stripe-java".

# Java

### Code Snippet

Zuerst wird eine Klasse erstellt, mit der eine Zahlung getätigt werden kann und die auch den Status per Methode zurückgibt.

```
/* File: Payment.java */
public class Payment {
    public String makePayment(Long amount, String stripeToken,
RequestOptions options) throws StripeException {
        ChargeCreateParams params = ChargeCreateParams.builder()
                .setAmount(amount)
                .setCurrency("EUR")
                .setDescription("testpayment")
                .setSource(stripeToken)
                .build();

        Charge charge = Charge.create(params, options);
        return charge.getId();
    }

    public String getStatus(String chargeId, RequestOptions options)
throws StripeException {
        return Charge.retrieve(chargeId, options).getStatus();
    }
}
//Lines: 7
```

Anschließend werden die RequestOptions festgelegt und die Funktionen der Payment Klasse werden aufgerufen.

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

    public static void main(String[] args) {
        try {
            RequestOptions options = RequestOptions.builder()
                    .setApiKey(STRIPE_API_KEY)
                    .build();

            Payment payment = new Payment();
            String id = payment.makePayment(1000L, PAYMENT_TOKEN,
options);

            System.out.println(payment.getStatus(id, options));

        } catch(StripeException stripeException) {
            stripeException.printStackTrace();
        }
    }

// Lines: 7
```

**Bewertung**

- Lines of Code: 14 Zeilen. → 2
- Lesbarkeit: Der Code ist leicht verständlich, wird durch das in Java notwendige Exception-handling allerdings etwas unübersichtlich. → 4/5
- Dokumentation: In der Dokumentation [10] werden alle Endpunkte dokumentiert und es gibt auch Beispiele für verschiedene Sprachen, darunter auch Java. → 5/5
- Unterstützte Paradigmen: Die Bibliothek unterstützt objektorientierte Programmierung, allerdings wird keine funktionale Programmierung berücksichtigt, diese wäre in diesem Fall sinnvoll, da man dadurch zum Beispiel mithilfe eines Observers auf Änderungen des Status achten könnte. → 3/5

# Kotlin

**Code Snippet**

Zuerst wird eine Klasse erstellt, mit der eine Zahlung getätigt werden kann und die auch den Status per Methode zurückgibt.

```
/* File: Payment.kt */

class Payment {

    fun makePayment(amount: Long, stripeToken: String, options:
RequestOptions): String {
        val params = ChargeCreateParams.builder()
            .setAmount(amount)
            .setCurrency("EUR")
            .setDescription("testpayment")
            .setSource(stripeToken)
            .build()
        val charge = Charge.create(params, options)
        return charge.id
    }

    fun getStatus(chargeId: String, options: RequestOptions): String {
        return Charge.retrieve(chargeId, options).status
    }
}

//Lines: 7
```

Anschließend werden die RequestOptions festgelegt und die Funktionen der Payment Klasse werden aufgerufen.

```
/* File: main.kt */

fun main(args: Array<String>) {
    val options = RequestOptions.builder()
        .setApiKey(STRIPE_API_KEY)
        .build()
    val payment = Payment()
    val id = payment.makePayment(1000L, PAYMENT_TOKEN, options)
    println(payment.getStatus(id, options))
}

//Lines: 5
```

**Bewertung**

- Lines of Code: 12 Zeilen. → 4/5
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: In der Dokumentation werden alle Endpunkte dokumentiert und es gibt auch Beispiele für verschiedene Sprachen, darunter zwar Java aber leider nicht Kotlin. Da der Code in Kotlin allerdings fast derselbe ist wie der in

Java geschrieben Code gibt es hier keinen Punkteabzug.→ 5/5

- Unterstützte Paradigmen: Die Bibliothek unterstützt objektorientierte Programmierung, allerdings wird keine funktionale Programmierung berücksichtigt, diese wäre in diesem Fall sinnvoll, da man dadurch zum Beispiel mithilfe eines Observers auf Änderungen des Status achten könnte. → 3/5

## Groovy

### Code Snippet

Zuerst wird eine Klasse erstellt, mit der eine Zahlung getätigt werden kann und die auch den Status per Methode zurückgibt.

```
/* File: Payment.groovy */

class Payment {
    String makePayment(Long amount, String stripeToken, RequestOptions
options) {
        ChargeCreateParams params = ChargeCreateParams.builder()
                .setAmount(amount)
                .setCurrency("EUR")
                .setDescription("testpayment")
                .setSource(stripeToken)
                .build()
        Charge charge = Charge.create(params, options)
        charge.id
    }

    def getStatus(String chargeId, RequestOptions options) {
        Charge.retrieve(chargeId, options).status
    }
}

//Lines: 7
```

Anschließend werden die RequestOptions festgelegt und die Funktionen der Payment Klasse werden aufgerufen.

```groovy
/* File: Main.groovy */

    static main(args) {
        def options = RequestOptions.builder()
                .setApiKey(STRIPE_API_KEY)
                .build()
        Payment payment = new Payment()
        String id = payment.makePayment(1000L, PAYMENT_TOKEN, options)
        println(payment.getStatus(id, options))
    }

//Lines: 5
```

**Bewertung**

- Lines of Code: 12 Zeilen. → 4/5
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: In der Dokumentation werden alle Endpunkte dokumentiert und es gibt auch Beispiele für verschiedene Sprachen, darunter zwar Java aber leider nicht Groovy. Da der Code in Groovy allerdings fast derselbe ist wie der in Java geschrieben Code gibt es hier keinen Punkteabzug. → 5/5
- Unterstützte Paradigmen: Die Bibliothek unterstützt objektorientierte Programmierung, allerdings wird keine funktionale Programmierung berücksichtigt, diese wäre in diesem Fall sinnvoll, da man dadurch zum Beispiel mithilfe eines Observers auf Änderungen des Status achten könnte. → 3/5

# Scala

**Code Snippet**

Zuerst wird eine Klasse erstellt, mit der eine Zahlung getätigt werden kann und die auch den Status per Methode zurückgibt.

```
/* File: Payment.scala */

class Payment {
  def makePayment(amount: Long, stripeToken: String, options:
RequestOptions): String = {
    val params = ChargeCreateParams.builder()
      .setAmount(amount)
      .setCurrency("EUR")
      .setDescription("testpayment")
      .setSource(stripeToken)
      .build()
    val charge = Charge.create(params, options)
    charge.getId
  }1

  def getStatus(id: String, options: RequestOptions): String = {
    Charge.retrieve(id, options).getStatus
  }
}

//Lines: 7
```

Anschließend werden die RequestOptions festgelegt und die Funktionen der Payment Klasse werden aufgerufen.

```
/* File: Main.scala */

  def main(args: Array[String]): Unit = {
    def options = RequestOptions.builder()
      .setApiKey(STRIPE_API_KEY)
      .build()
    val payment = new Payment()
    val id = payment.makePayment(1000L, PAYMENT_TOKEN, options)
    println(payment.getStatus(id, options))
  }

//Lines: 5
```

**Bewertung**

- Lines of Code: 12 Zeilen. → 4/5
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: In der Dokumentation werden alle Endpunkte dokumentiert und es gibt auch Beispiele für verschiedene Sprachen, darunter zwar Java aber leider nicht Scala. Da der Code in Scala allerdings fast derselbe ist wie der in Java

geschrieben Code gibt es hier keinen Punkteabzug. → 5/5

- Unterstützte Paradigmen: Die Bibliothek unterstützt objektorientierte Programmierung, allerdings wird keine funktionale Programmierung berücksichtigt, diese wäre in diesem Fall sinnvoll, da man dadurch zum Beispiel mithilfe eines Observers auf Änderungen des Status achten könnte. → 3/5

## Clojure

### Code Snippet

Zuerst werden die Funktionen make-payment und get-status definiert, mit denen die benötigten Funktionen implementiert werden.

```
;;File: payment.clj

(defn make-payment
  [amount stripeToken ^RequestOptions options]
        (let [chargeParams (-> (ChargeCreateParams/builder)
                               (.setSource stripeToken)
                               (.setCurrency "EUR")
                               (.setDescription "testpayment")
                               (.setAmount amount)
                               (.build)
                               )]
        (.getId (Charge/create chargeParams options)))
)

(defn get-status
  [id requestOptions] (.getStatus (Charge/retrieve id requestOptions))
  )

;;Lines: 6
```

Als nächstes werden die RequestOptions definiert und die beiden Methoden werden aufgerufen und der Status der Zahlung wird auf die Konsole ausgegeben.

```
;;File: core.clj

(defn -main
  ([] (let [options (-> (RequestOptions/builder)
                        (.setApiKey STRIPE_API_KEY)
                        (.build))]
        (let [id (payment/make-payment 1000 PAYMENT_TOKEN options)]
          (println (payment/get-status id options)))
        )
   )
  )
;;Lines: 4
```

**Bewertung**

- Lines of Code: 10 Zeilen. → 5/5
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: In der Dokumentation werden alle Endpunkte dokumentiert und es gibt auch Beispiele für verschiedene Sprachen, darunter zwar Java aber leider nicht Clojure. Da der Code in Clojure trotz unterschiedlicher Syntax fast derselbe ist wie der in Java geschrieben Code gibt es hier keinen Punkteabzug. → 5/5
- Unterstützte Paradigmen: Die Bibliothek unterstützt objektorientierte Programmierung, allerdings wird funktionale Programmierung insofern nicht berücksichtigt dass man zum Beispiel mithilfe eines Observers auf Änderungen des Status achten könnte. Die Implementierung in Clojure (einer funktionalen Sprache) ist jedoch leicht möglich. → 3/5

## Beurteilungstabelle und Fazit

| Sprache | Java | Kotlin | Groovy | Scala | Clojure |
|---|---|---|---|---|---|
| Lines of Code | 3 | 4 | 4 | 4 | 5 |
| Lesbarkeit | 4 | 5 | 5 | 5 | 5 |

| Sprache | Java | Kotlin | Groovy | Scala | Clojure |
|---|---|---|---|---|---|
| Dokumentation | 5 | 5 | 5 | 5 | 5 |
| Unterstützte Paradigmen | 3 | 3 | 3 | 3 | 3 |
| Ergebnis | 15 | 17 | 17 | 17 | 18 |

*Figure 2. Beurteilungstabelle Stripe-API*

Da Java die meisten Zeilen Code benötigt und durch das notwendige Exceptionhandling auch unlesbarer wird, belegt Java in diesem Fall den letzten Platz.

Dass Clojure auf dem ersten Platz liegt, liegt daran, dass die Schreibweise in Clojure am kompaktesten ist. Allgemein führt der Einsatz der Stripe-API nicht dazu, dass der Code unlesbarer wird, allerdings wäre es besser, wenn man den Status mit einem Observer asynchron überwachen könnte.

[10] https://stripe.com/docs/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.

In allen Sprachen wird die Feign-Bibliothek verwendet, da diese in allen Sprachen verwendet werden kann.
Die DTO Klassen werden nicht zur Bewertung herangezogen.

## Java

### Code Snippet

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

```java
/* 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. → 4/5
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: Die Dokumentation [11] 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

### Code Snippet

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

```kotlin
/* 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. → 4/5
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: Die Dokumentation ist zwar sehr umfangreich und enthält viele Beispiele, allerdings gibt es leider keine Beispiele für den Umgang mit Kotlin. Da jedoch fast kein Unterschied bei der Umsetzung in Kotlin zu der Umsetzung in Java besteht, werden hierfür keine Punkte abgezogen→ 5/5
- Unterstützte Paradigmen: Die OpenFeign Bibliothek unterstützt sowohl objektorientierte Programmierung als auch funktionale Programmierung (mit CompletableFuture Objekten). → 5/5

# Groovy

**Code Snippet**

Um auf die Rest-API zuzugreifen wird ein Interface mit den Methoden, die später aufgerufen werden, deklariert.

```
/* File: UserFeignClient.groovy */

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
```

Mit dem FeignBuilder wird der Client instanziert und die Methoden werden aufgerufen.

```
/* File: Main.groovy */

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

        def user = getUser(client)
        def createdUser = client.createUser(new CreateUser.Request
("Testuser", "Programmer"))
    }

    static def getUser(UserFeignClient client) {
        client.getUser(2)
    }

    static def createUser(UserFeignClient client, CreateUser.Request
user) {
        client.createUser(user)
    }


//Lines: 8
```

**Bewertung**

- Lines of Code: 14 Zeilen. → 4/5
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: Die Dokumentation ist zwar sehr umfangreich und enthält viele Beispiele, allerdings gibt es leider keine Beispiele für den Umgang mit Groovy. Da jedoch fast kein Unterschied bei der Umsetzung in Groovy zu der Umsetzung in Java besteht, werden hierfür keine Punkte abgezogen→ 5/5
- Unterstützte Paradigmen: Die OpenFeign Bibliothek unterstützt sowohl objektorientierte Programmierung als auch funktionale Programmierung (mit CompletableFuture Objekten). → 5/5

## Scala

### Code Snippet

Zuerst wird ein trait erstellt, in dem die Routen und Parameter definiert werden.

```scala
/* File: UserFeignClient.scala */

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

  @RequestLine("POST /users")
  @Headers(Array[String]("Content-Type: application/json"))
  def createUser(createUser: CreateUserRequest): CreateUserResponse
}

// Lines: 6
```

Der Client wird mit dem FeignBuilder erstellt und anschließend werden die Methoden des Clients aufgerufen.

```
/* File: Main.scala */

  def main(args: Array[String]): Unit = {
    val userFeignClient = Feign.builder()
      .client(new OkHttpClient())
      .encoder(new GsonEncoder())
      .decoder(new GsonDecoder())
      .target(classOf[UserFeignClient], "https://reqres.in/api")

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

  def getUser(client: UserFeignClient) :GetUser = {
    client.getUser(2)
  }

  def createUser(client: UserFeignClient, createUserRequest:
CreateUserRequest): CreateUserResponse = {
    client.createUser(createUserRequest)
  }

// Lines: 8
```

**Bewertung**

- Lines of Code: 14 Zeilen → 4/5
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: Die Dokumentation ist zwar sehr umfangreich und enthält viele Beispiele, allerdings gibt es leider keine Beispiele für den Umgang mit Scala. Da jedoch fast kein Unterschied bei der Umsetzung in Scala zu der Umsetzung in Java besteht, werden hierfür keine Punkte abgezogen→ 5/5
- Unterstützte Paradigmen: Die OpenFeign Bibliothek unterstützt sowohl objektorientierte Programmierung als auch funktionale Programmierung (mit CompletableFuture Objekten). → 5/5

# Clojure

**Code Snippet**

Zuerst wird ein Interface definiert, in dem die REST-Methoden definiert werden, die aufgerufen werden sollen.

```
;; File: userFeignClient.clj

(definterface userFeignClient
  (^{RequestLine "GET /users/{id}"}  getUser [^{Param "id"} id])
  (^{RequestLine "POST /users"} ^{Headers ["Content-Type:
application/json"]} createUser [user] )
)

;; Lines: 3
```

Anschließend wird ein Client mit dem FeignBuilder instanziert und verwendet, um die Requests abzusenden.

```
;; File: core.clj

(defn getUser
  [client] (. client getUser 2)
)

(defn createUser
  [client createUserRequest] (. client createUser createUserRequest)
)

(defn -main
  ([] (let [client (-> (Feign/builder)
           (.client (new OkHttpClient))
           (.encoder (new GsonEncoder))
           (.decoder (new GsonDecoder))
           (.target userFeignClient/userFeignClient
"https://reqres.in/api"))]
           (let [getUserResponse (getUser client)])
           (let [createUserResponse (createUser client {:name "Testuser"
:job "Programmer"})])
         )
    )
)

;; Lines: 8
```

**Bewertung**

- Lines of Code: 11 Zeilen → 5/5
- Lesbarkeit: Der Code ist leicht verständlich, allerdings sorgen die Annotationen

beim Interface dafür, dass der Code etwas unübersichtlich wird. → 4/5

- Dokumentation: Die Dokumentation ist zwar sehr umfangreich und enthält viele Beispiele, allerdings gibt es leider keine Beispiele für den Umgang mit Clojure. Da Clojure sich syntaktisch stärker von Java unterscheidet als die anderen berücksichtigten JVM Sprachen werden hier Punkte abgezogen.→ 3/5
- Unterstützte Paradigmen: Die OpenFeign Bibliothek unterstützt sowohl objektorientierte Programmierung als auch funktionale Programmierung (mit CompletableFuture Objekten). → 5/5

## Beurteilungstabelle und Fazit

| Sprache | Java | Kotlin | Groovy | Scala | Clojure |
|---|---|---|---|---|---|
| Lines of Code | 4 | 4 | 4 | 4 | 5 |
| Lesbarkeit | 5 | 5 | 5 | 5 | 4 |
| Dokumentation | 5 | 5 | 5 | 5 | 3 |
| Unterstützte Paradigmen | 5 | 5 | 5 | 5 | 5 |
| Ergebnis | 19 | 19 | 19 | 19 | 17 |

*Figure 3. Beurteilungstabelle Rest-APIs*

Die Ergebnisse sind bei allen Sprachen sehr ähnlich, da in allen Sprachen dieselbe Bibliothek verwendet werden konnte und diese Biblothek auch sehr gut geeignet ist, um Rest-APIs anzusteuern.

Clojure belegt aufgrund der Dokumentation, die nur für Java verfasst wurde, den letzten Platz, außerdem wird der Code durch die Annotationen in Clojure etwas unübersichtlich.

[11] https://github.com/OpenFeign/feign

# Stream API

In allen Sprachen soll mit der Stream API (oder der Alternative in der jeweiligen Sprache) eine Liste von Lebensmitteln in insgesamt drei Methoden

- nach dem Namen sortiert werden
- nach der Kategorie gruppiert werden
- nach Lebensmitteln durchsucht werden, die weniger als einen Euro kosten.

Die Liste soll unverändert bleiben.

## Java

In Java wird die Stream API verwenden.

**Code Snippet**

```
public static List<Food> sortFoodByName(List<Food> food) {
    return food.stream().sorted(Comparator.comparing(Food::getName
)).collect(Collectors.toList());
}

public static Map<Category, List<Food>> groupFoodByCategory(List<Food>
food) {
    return food.stream().collect(Collectors.groupingBy(Food:
:getCategory));
}

public static List<Food> findFoodWorthLessThanAEuro(List<Food> food) {
    return food.stream().filter(f -> f.getPrice() < 1.0).collect
(Collectors.toList());
}

//Lines: 6
```

**Bewertung**

- Lines of Code: 6 Zeilen → 5/5
- Lesbarkeit: Der Code ist leicht verständlich, im Vergleich zu den anderen Sprachen fällt jedoch auf, dass die sorted(), groupingBy() und filter() Methoden

nicht direkt auf dem Listenobjekt sind sondern auf einem Stream Objekt sind. Außerdem muss der Stream in zwei von drei Fällen wieder mit einem eigenen Call zu einer Liste umgewandelt werden. → 3/5

- Dokumentation: Die Dokumentation [12] enthält zahlreiche Beispiele und dokumentiert die komplette Stream API. → 5/5
- Unterstützte Paradigmen: Die Stream API sorgt dafür, dass Collections funktional verarbeitet werden können, die gestreamte Liste wird nicht verändert. → 5/5

## Kotlin

In Kotlin wird keine Stream API verwendet, da die Listen in Kotlin bereits die Funktionalitäten der Stream API implementieren.

### Code Snippet

```
fun sortFoodByName(food: List<Food>): List<Food> = food.sortedBy { it
.category }

fun groupFoodByCategory(food: List<Food>): Map<Category, List<Food>> =
food.groupBy(Food::category)


fun findFoodWorthLessThanAEuro(food: List<Food>): List<Food> = food
.filter { f->f.price<1.0 }

//Lines: 6
```

### Bewertung

- Lines of Code: 6 Zeilen → 5/5
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: Da die Dokumentation von Collections in der Kotlinlang Dokumentation [13] vollständig ist, die Dokumentation eine Suchfunktion hat und bei den jeweiligen Funktionen auch Anwendungsbeispiele vorhanden sind erhält Kotlin alle Punkte für die Dokumentation. → 5/5

- Unterstützte Paradigmen: Die Methoden, die statt der Stream API verwendet werden, sind funktionale Konstrukte und die Listen, die verwendet werden, werden nicht verändert. → 5/5

## Groovy

In Groovy wird die Stream API nicht verwendet, da die Listen in Groovy bereits die Funktionen der Stream API implementieren.

**Code Snippet**

```
static def sortFoodByName(List<Food> food) {
    food.toSorted {f->f.name}
}

static def groupFoodByCategory(List<Food> food) {
    food.groupBy {f->f.category}
}

static def findFoodWorthLessThanAEuro(List<Food> food) {
    food.findAll {f->f.price < 1}
}

//Lines: 6
```

**Bewertung**

- Lines of Code: 6 Zeilen → 5/5
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: Die Dokumentation von Groovy für den Umgang mit Collections [14] enthält gute Beispiele und Beschreibungen für alle Methoden, die equivalent zu den Funktionen der Stream API in Java sind. → 5/5
- Unterstützte Paradigmen: Die Methoden, die statt der Stream API verwendet werden, sind funktionale Konstrukte und die Listen, die verwendet werden, werden nicht verändert. → 5/5

## Scala

In Scala wird die Stream API nicht verwendet, da die Listen in Scala bereits die Funktionen der Stream API implementieren.

### Code Snippet

```
def sortFoodByName(food: List[Food]): List[Food] = {
  food.sortBy(f => f.name)
}

  def groupFoodByCategory(food: List[Food]): Map[String, List[Food]] =
{
  food.groupBy(f => f.category)
}

def findFoodWorthLessThanAEuro(food: List[Food]): List[Food] = {
  food.filter(f => f.price<1)
}

//Lines: 6
```

### Bewertung

- Lines of Code: 6 Zeilen → 5/5
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: In der Dokumentation von Scala [15] werden die Funktionen, die statt der Stream API verwendet werden können, dokumentiert und manche (nicht alle, zum Beispiel die "filter" Funktion) haben auch Beispiele dabei. Da nicht überall Beispiele sind gibt es hier einen Punkt Abzug, da alle anderen berücksichtigten JVM Sprachen mehr Beispiele gebracht haben. → 4/5
- Unterstützte Paradigmen: Die Methoden, die statt der Stream API verwendet werden, sind funktionale Konstrukte und die Listen, die verwendet werden, werden nicht verändert. → 5/5

## Clojure

In clojure.core gibt es bereits die Funktionen der Stream API, deshalb wird die Stream API nicht verwendet.

**Code Snippet**

```
(defn sort-food-by-name
  [food] (sort-by :name food))

(defn group-food-by-category
   [food] (group-by :category food))

(defn find-food-worth-less-than-a-euro
  [food] (filter #(< (:price %) 1) food))

;;Lines: 6
```

**Bewertung**

- Lines of Code: 6 Zeilen → 5/5
- Lesbarkeit: Der Code ist leicht verständlich. → 5/5
- Dokumentation: Die Dokumentation, die in dem Cursive Plugin von dem IntelliJ Marktplatz [16] inkludiert ist verfügt pro Funktion, die statt einer Stream API Funktion verwendet wird, sehr viele Beispiele mit verschiedenen Anwendungsfällen und erklärt auch die Funktion sehr detailliert, aus diesem Grund bekommt Clojure alle Punkte. → 5/5
- Unterstützte Paradigmen: Clojure ist eine funktionale Programmiersprache und die Funktionen, die anstelle der Stream-API verwendet werden, sind auch funktional konzipiert. → 5/5

## Beurteilungstabelle und Fazit

| Sprache | Java | Kotlin | Groovy | Scala | Clojure |
|---|---|---|---|---|---|
| Lines of Code | 5 | 5 | 5 | 5 | 5 |
| Lesbarkeit | 3 | 5 | 5 | 5 | 5 |
| Dokumentation | 5 | 5 | 5 | 4 | 5 |
| Unterstützte Paradigmen | 5 | 5 | 5 | 5 | 5 |
| Ergebnis | 18 | 20 | 20 | 19 | 20 |

*Figure 4. Beurteilungstabelle Stream-API und Alternativen*

Mit 18 Punkten belegt Java den letzten Platz, das liegt daran, dass der Code in Java durch den Einsatz der Stream-API etwas unleserlicher wird als der Code der anderen Sprachen.

Ansonsten haben alle Sprachen 19-20 Punkte, das liegt daran, dass die Alternativen zur Stream-API in den anderen Sprachen direkt auf den Collections sind (beziehungsweise core Funktionen in Clojure).

[12] https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html

[13] https://kotlinlang.org/docs/

[14] http://docs.groovy-lang.org/next/html/documentation/working-with-collections.html

[15] https://www.scala-lang.org/api/2.12.5/scala/collection/immutable/List.html

[16] https://plugins.jetbrains.com/plugin/8090-cursive

# Java Mail API

## Java

**Code Snippet Bewertung**

## Kotlin

**Code Snippet Bewertung**

## Groovy

**Code Snippet Bewertung**

## Scala

**Code Snippet Bewertung**

## Clojure

**Code Snippet Bewertung**

# Fazit

Alle Sprachen

# References

https://reqres.in/

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

https://github.com/OpenFeign/feign

https://stripe.com/docs/api/

https://github.com/stripe/stripe-java

https://blog.frankel.ch/learning-clojure/5/

https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html

# Glossary

# Index