Entwurfsphase

18. Januar 2017

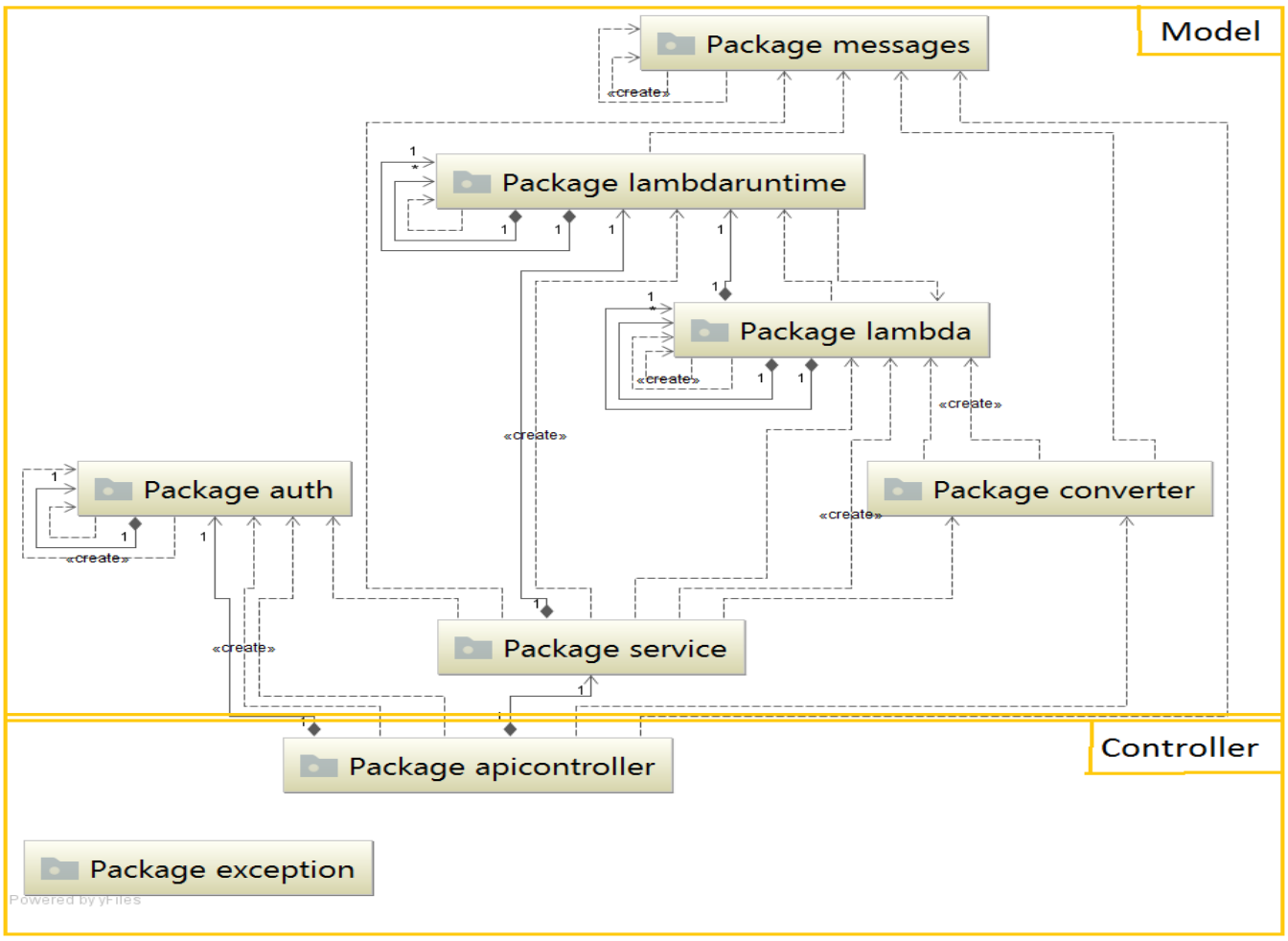
Inhaltsverzeichnis

Kapitel 1

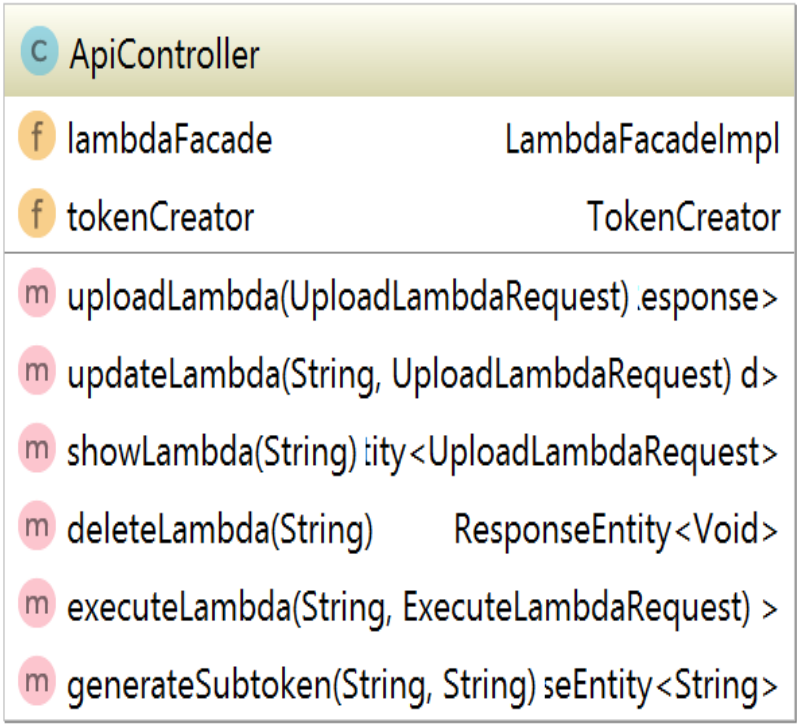
Klassen

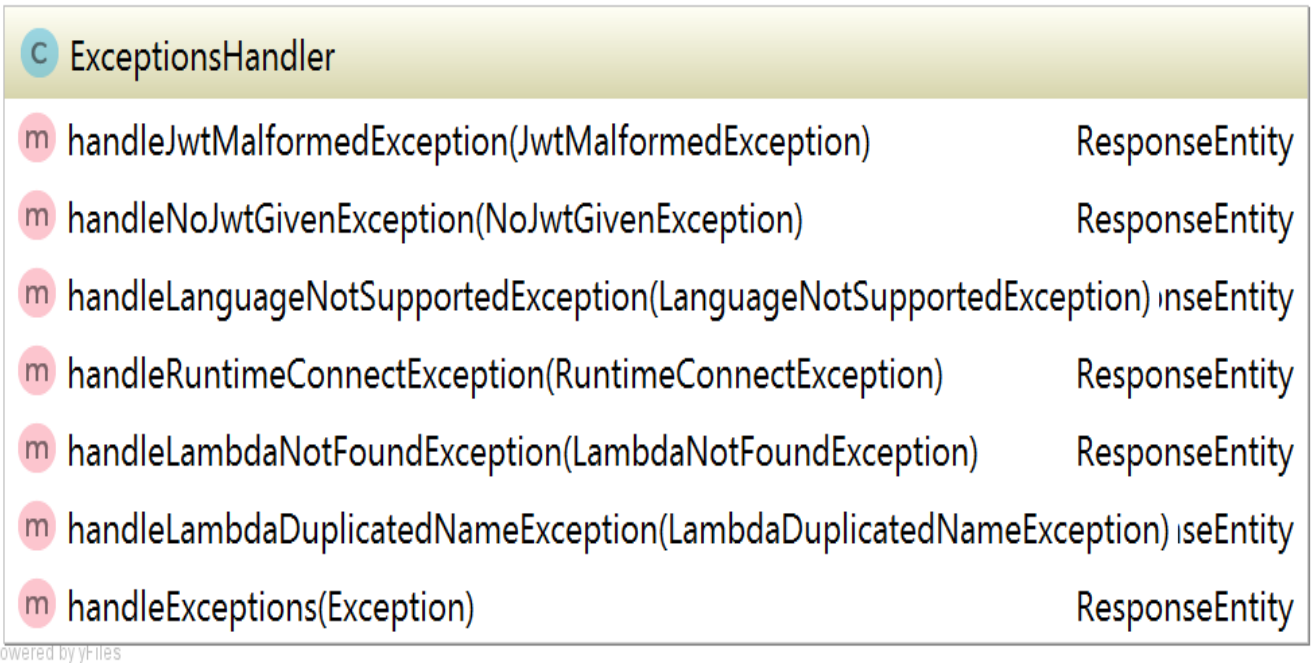
1.1 Klassendiagramm

1.1.1 Grobstruktur

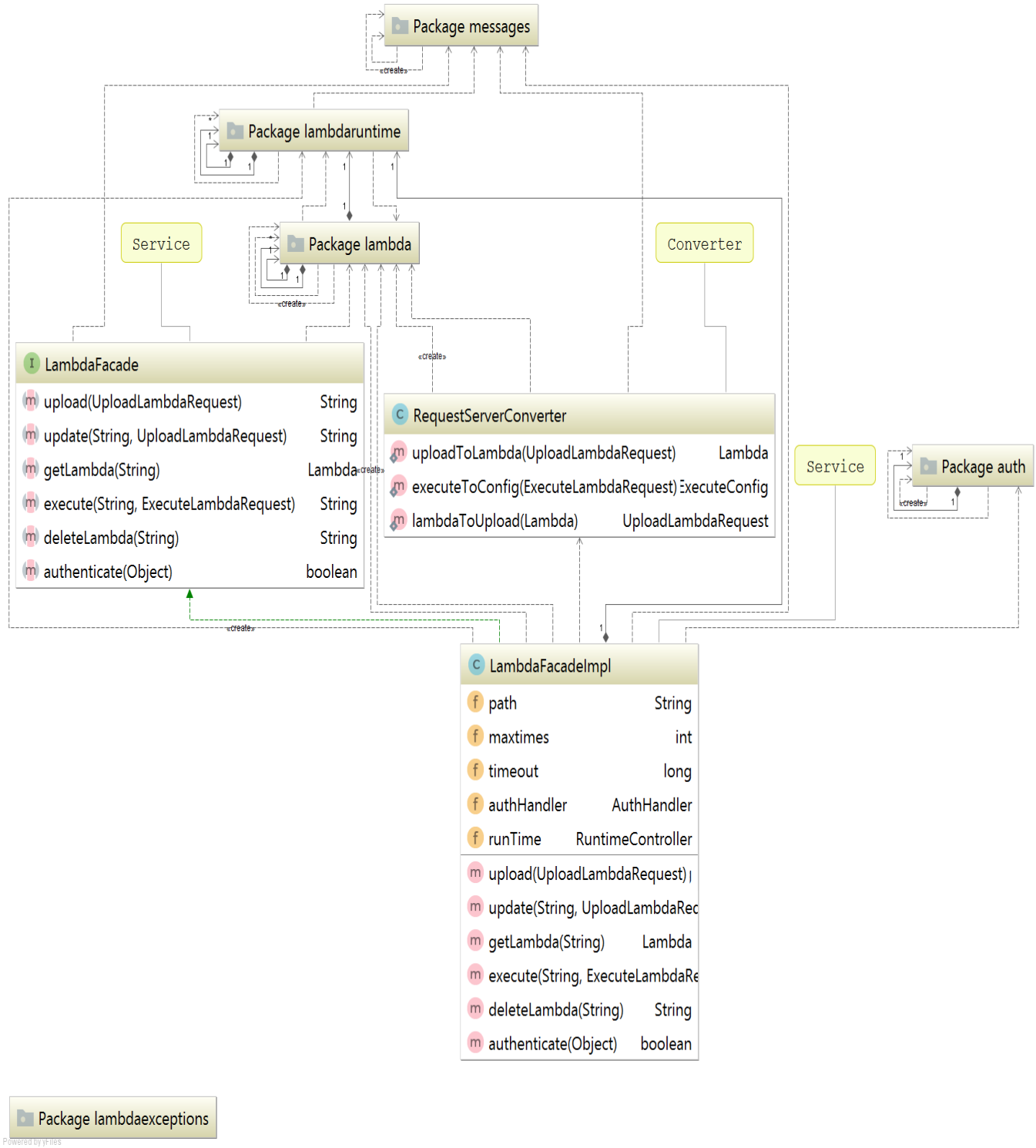


1.1.2 ApiController

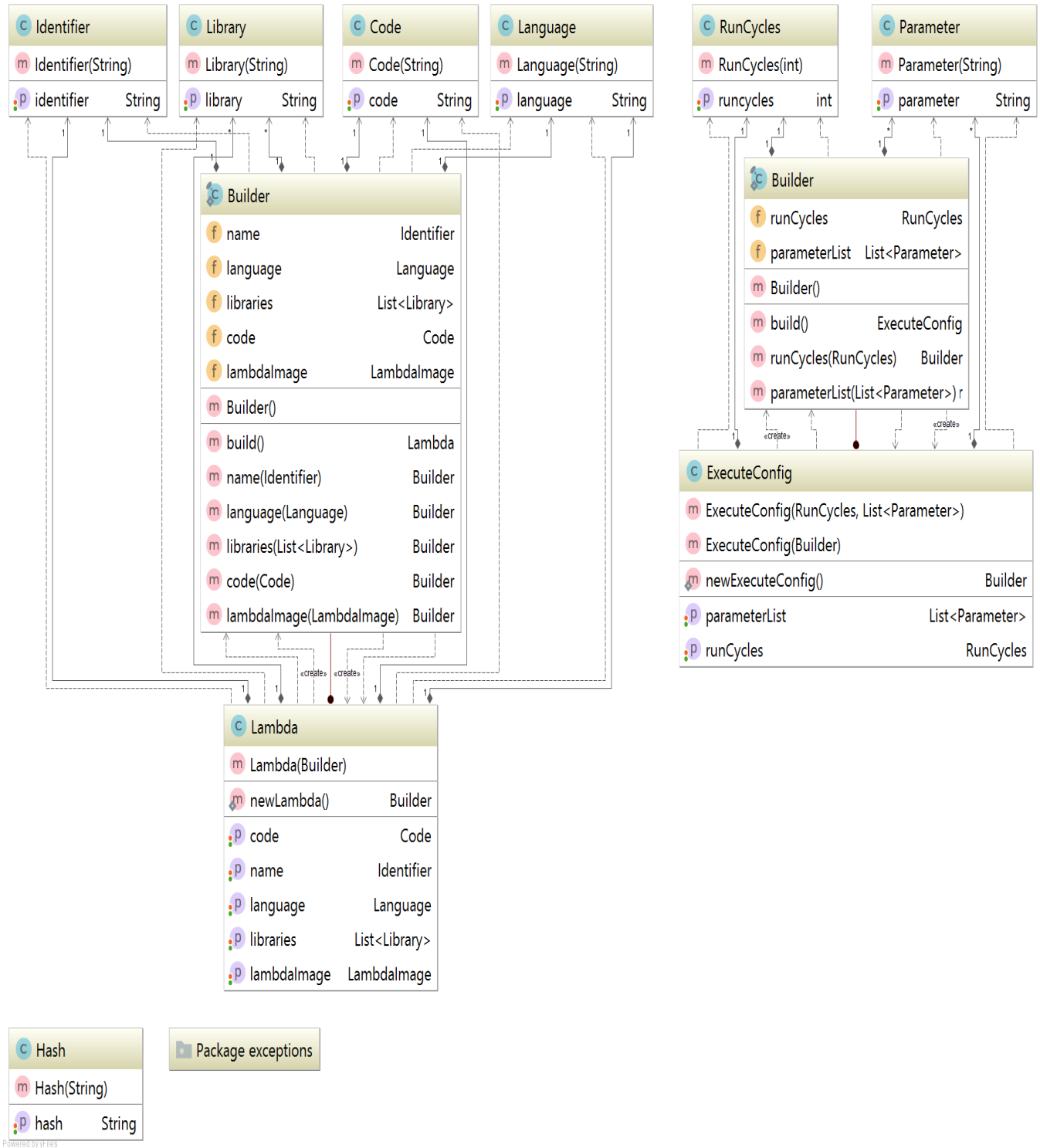




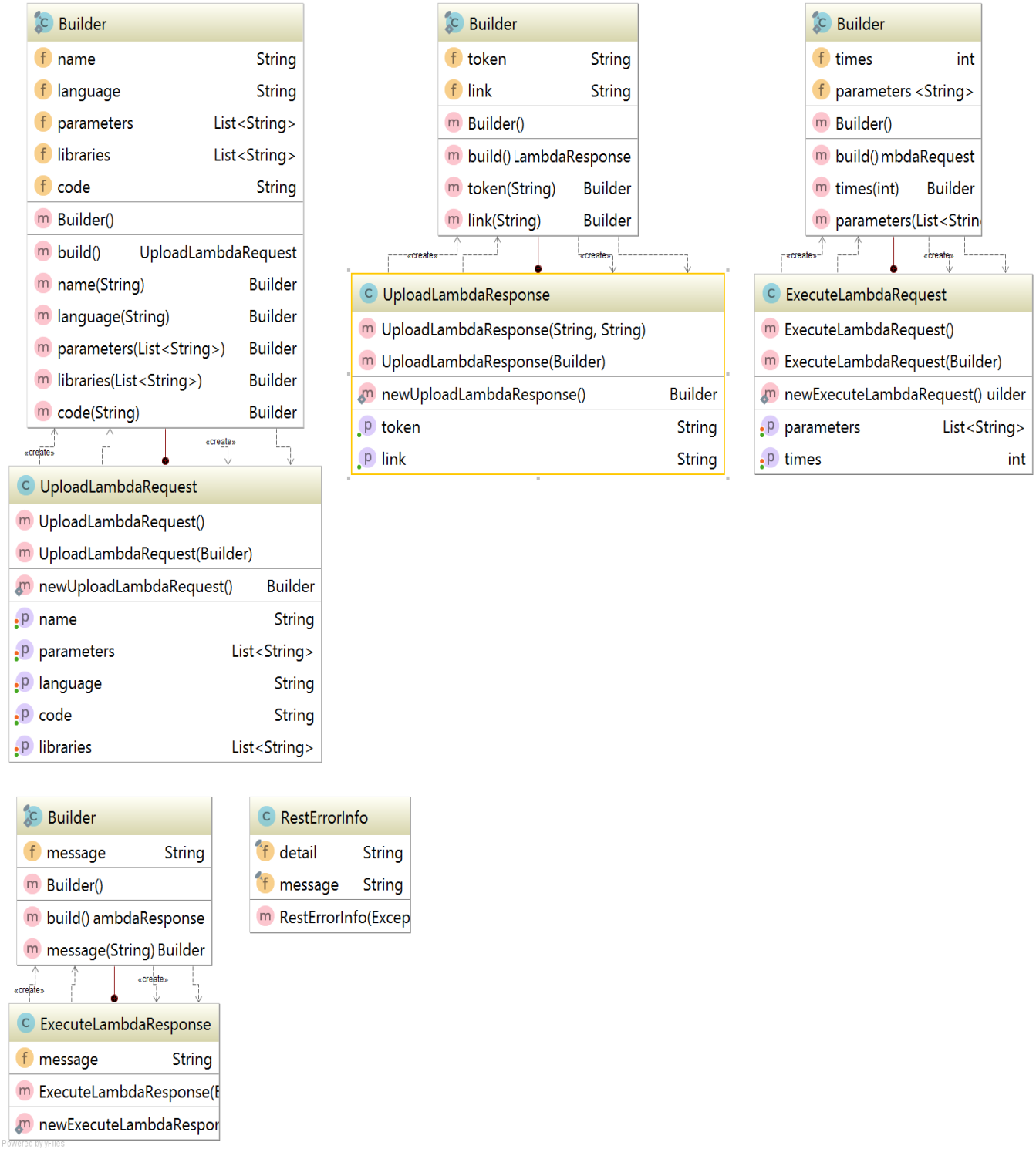
1.1.3 Model


Package messages
«create»
Package lambdaruntime
Service
Package lambda
Converter
«create»
«create»
LambdaFacade
upload(UploadLambdaRequest) String
update(String, UploadLambdaRequest) String
getLambda(String) Lambda
execute(String, ExecuteLambdaRequest) String
deleteLambda(String) String
authenticate(Object) boolean
RequestServerConverter
uploadToLambda(UploadLambdaRequest) Lambda
executeToConfig(ExecuteLambdaRequest) ExecuteConfig
lambdaToUpload(Lambda) UploadLambdaRequest
«create»
Service
Package auth
«create»
«create»
LambdaFacadeImpl
path String
maxtimes int
timeout long
authHandler AuthHandler
runTime RuntimeController
upload(UploadLambdaRequest)
update(String, UploadLambdaRec
getLambda(String) Lambda
execute(String, ExecuteLambdaRe
deleteLambda(String) String
authenticate(Object) boolean
Package lambdaexceptions

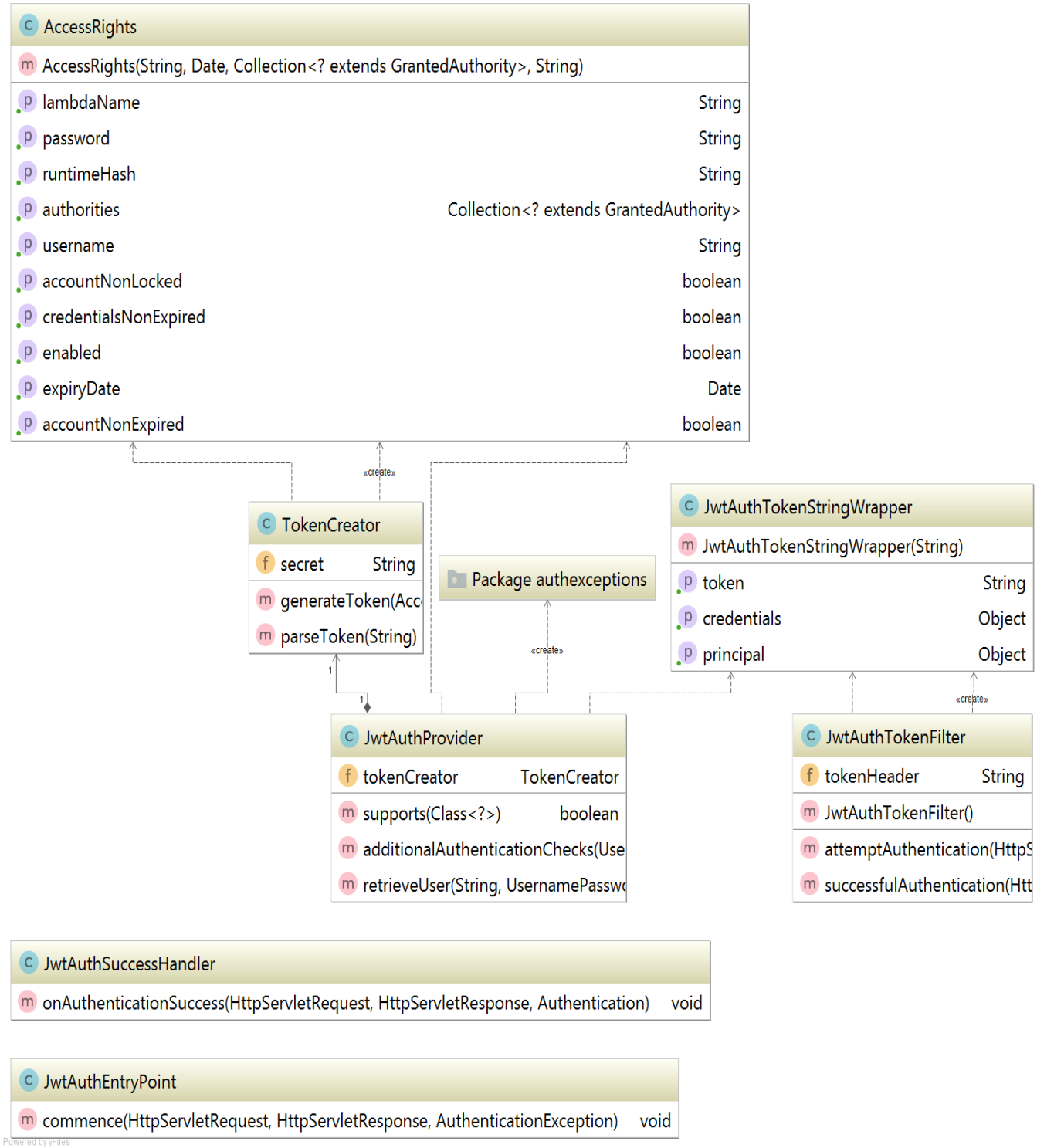
1.1.4 Lambda


Identifier
Identifier(String)
identifier String
Library
Library(String)
library String
Code
Code(String)
code String
Language
Language(String)
language String
RunCycles
RunCycles(int)
runcycles int
Parameter
Parameter(String)
parameter String
Builder
name Identifier
language Language
libraries List<Library>
code Code
lambdaImage LambdaImage
Builder()
build() Lambda
name(Identifier) Builder
language(Language) Builder
libraries(List<Library>) Builder
code(Code) Builder
lambdaImage(LambdaImage) Builder
Builder
runCycles RunCycles
parameterList List<Parameter>
Builder()
build() ExecuteConfig
runCycles(RunCycles) Builder
parameterList(List<Parameter>) r
ExecuteConfig
ExecuteConfig(RunCycles, List<Parameter>)
ExecuteConfig(Builder)
newExecuteConfig() Builder
parameterList List<Parameter>
runCycles RunCycles
«create»
«create»
«create»
«create»
Lambda
Lambda(Builder)
newLambda() Builder
code Code
name Identifier
language Language
libraries List<Library>
lambdaImage LambdaImage
Hash
Hash(String)
hash String
Package exceptions

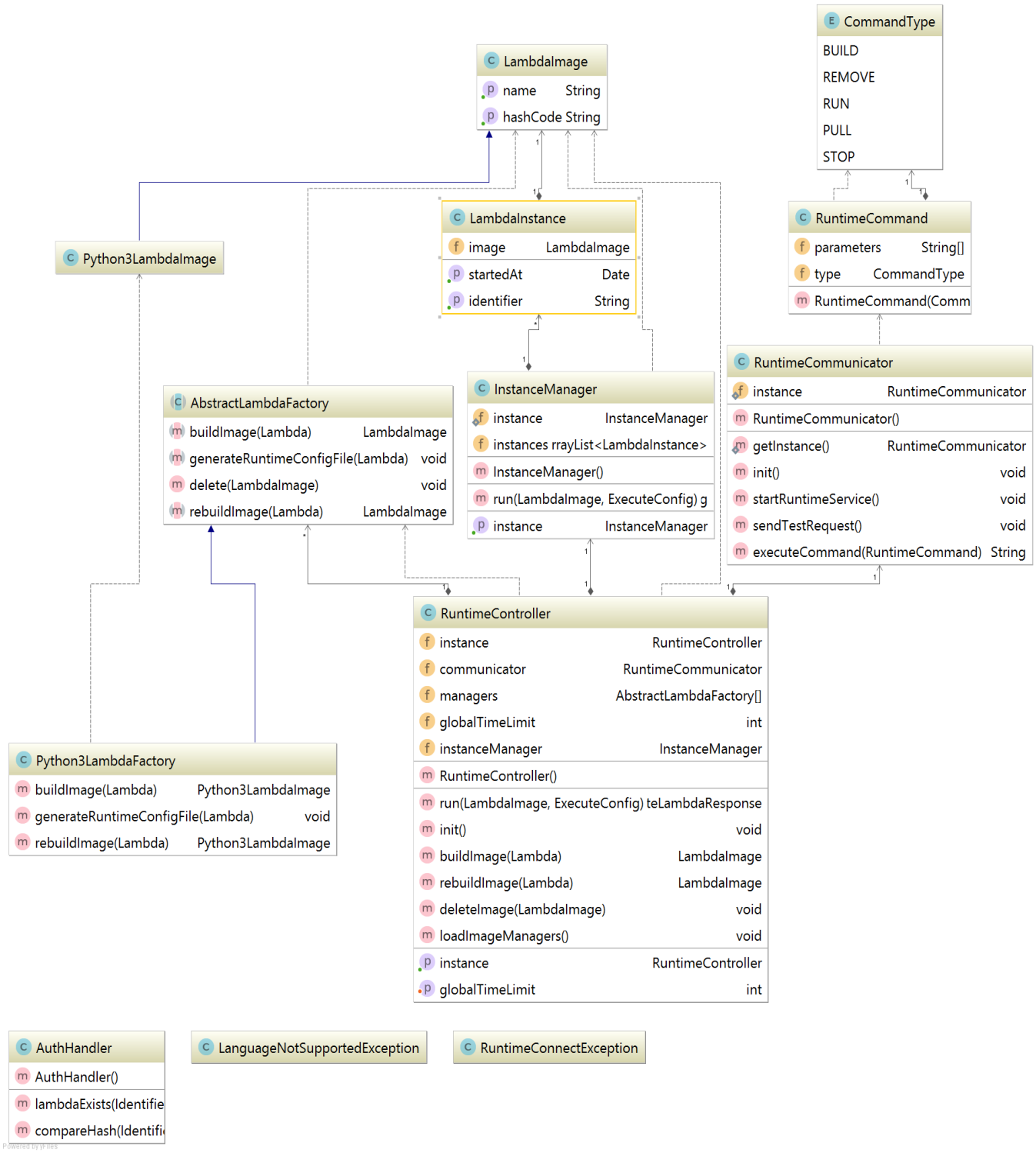
1.1.5 Messages



1.1.6 Auth

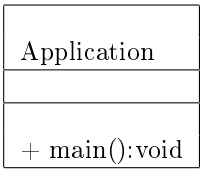

AccessRights
AccessRights(String, Date, Collection<? extends GrantedAuthority>, String)
lambdaName String
password String
runtimeHash String
authorities Collection<? extends GrantedAuthority>
username String
accountNonLocked boolean
credentialsNonExpired boolean
enabled boolean
expiryDate Date
accountNonExpired boolean
«create»
TokenCreator
secret String
generateToken(Acc
parseToken(String)
Package authexceptions
«create»
JwtAuthTokenStringWrapper
JwtAuthTokenStringWrapper(String)
token String
credentials Object
principal Object
«create»
1
1
JwtAuthProvider
tokenCreator TokenCreator
supports(Class<?>) boolean
additionalAuthenticationChecks(Use
retrieveUser(String, UsernamePassw
JwtAuthTokenFilter
tokenHeader String
JwtAuthTokenFilter()
attemptAuthentication(HttpS
successfulAuthentication(Htt
JwtAuthSuccessHandler
onAuthenticationSuccess(HttpServletRequest, HttpServletResponse, Authentication) void
JwtAuthEntryPoint
commence(HttpServletRequest, HttpServletResponse, AuthenticationException) void

1.1.7 Runtime


CommandType
BUILD
REMOVE
RUN
PULL
STOP
LambdaImage
name String
hashCode String
LambdaInstance
image LambdaImage
startedAt Date
identifier String
RuntimeCommand
parameters String[]
type CommandType
RuntimeCommand(Comm
Python3LambdaImage
RuntimeCommunicator
instance RuntimeCommunicator
RuntimeCommunicator()
getInstance() RuntimeCommunicator
init() void
startRuntimeService() void
sendTestRequest() void
executeCommand(RuntimeCommand) String
AbstractLambdaFactory
buildImage(Lambda) LambdaImage
generateRuntimeConfigFile(Lambda) void
delete(LambdaImage) void
rebuildImage(Lambda) LambdaImage
InstanceManager
instance InstanceManager
instances rrayList<LambdaInstance>
InstanceManager()
run(LambdaImage, ExecuteConfig) g
instance InstanceManager
RuntimeController
instance RuntimeController
communicator RuntimeCommunicator
managers AbstractLambdaFactory[]
globalTimeLimit int
instanceManager InstanceManager
RuntimeController()
run(LambdaImage, ExecuteConfig) teLambdaResponse
init() void
buildImage(Lambda) LambdaImage
rebuildImage(Lambda) LambdaImage
deleteImage(LambdaImage) void
loadImageManagers() void
instance RuntimeController
globalTimeLimit int
Python3LambdaFactory
buildImage(Lambda) Python3LambdaImage
generateRuntimeConfigFile(Lambda) void
rebuildImage(Lambda) Python3LambdaImage
AuthHandler
AuthHandler()
lambdaExists(Identifie
compareHash(Identifi
LanguageNotSupportedException
RuntimeConnectException

1.2 Detaillierte Klassenbeschreibung: Start-und Konfigurationsklassen

Application



Beschreibung:
Main-Klasse zum Starten des Programms.

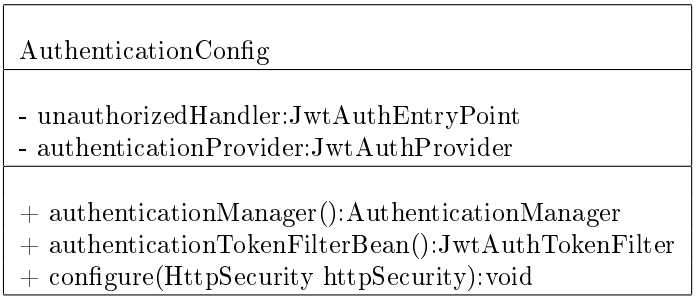
Attribute:

- keine

Methoden:

- main():void
  Startet Spring und die SpringApplication.

AuthenticationConfig



Beschreibung:
Konfigurationsklasse für die Authentifizierung. Hier wird festgelegt, welche Pfade auf eine Authentifizierung gemappt werden und welche Rollen die Authentifizierungsmerkmale besitzen müssen, um eine Anfrage bearbeiten zu lassen.

Attribute:

- unauthorizedHandler:JwtAuthEntryPoint
  JwtAuthEntryPoint Objekt zur Behandlung von nicht-authorisierten Anfragen
- authenticationProvider:JwtAuthProvider
  JwtAuthProvider Objekt zur Prüfung der Authentifizierungsmerkmale

Methoden:

- authenticationManager():AuthenticationManager
  Gibt einen spezifizierten AuthenticationManager von Spring Security zurück.
- authenticationTokenFilterBean():JwtAuthTokenFilter
  Setzt die Klassen, mit denen Spring Security arbeiten soll.

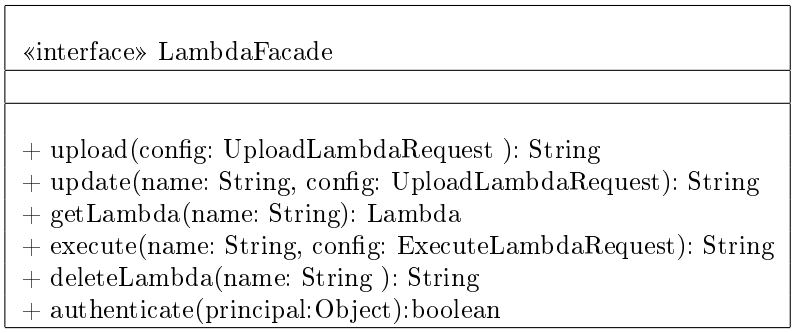
- configure(HttpSecurity httpSecurity):void
  Konfiguiert die Pfade für die Authentifizierung und die Rollen der Authentifizierungsmerkmale für die Pfade.

1.3 Detaillierte Klassenbeschreibung: Model

1.3.1 ServiceLayer: Service

Für Service wird die Fassade als Entwurfsmuster benutzt. Die Fassade ist dafür gemacht, um die Funktionalität an andere Klassen des Subsystems zu delegieren und dadurch den Umgang mit dem Subsystem zu vereinfachen. In unserem Fall wird die Fassade mit solchen Pakete wie lambda, messages, lambdaruntime, auth, converter und apicontroller arbeiten.

LambdaFacade



Beschreibung:
Es ist eine Fassadeschnittstelle. Sie wird als Schnittstelle implementiert, um die Fassade flexibeler zu machen.

Attribute:

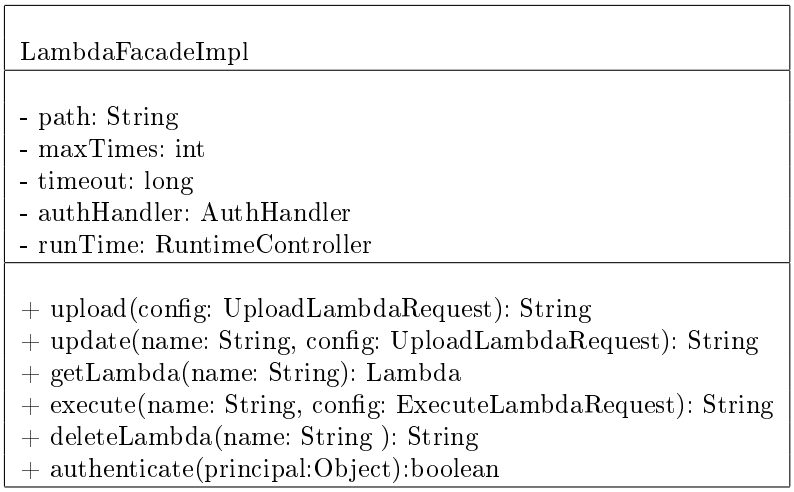
- keine

Methoden:

- upload(config: UploadLambdaRequest): String
  Die Schnittstelle für das Hochladen von Lambdas. Muss implementiert werden, damit der Nutzer das Lambda an Serverless übergeben könnte.
- update(name: String, config: UploadLambdaRequest): String
  Die Schnittstelle für die Lambda Aktualisierung. Muss implementiert werden, damit der Nutzer das Lambda korrigieren oder ändern kann.
- getLambda(name: String): Lambda
  Die Schnittstelle, damit der Nutzer die Lambda-Konfigurationen anschauen kann.
- execute(name: String, config: ExecuteLambdaRequest): String
  Die Schnittstelle für die Lambda Ausführung. Muss implementiert werden, damit der Nutzer mit eingegebenen Parameter und Anzahl der Ausführungen das Lambda in Serverless ausführen kann.
- deleteLambda(name: String): String
  Die Schnittstelle für die Entfernung von Lambdas. Muss implementiert werden, damit der Nutzer oder Serverless selbst nach einer definierten Zeit das Lambda löschen kann.

- authenticate(principal:Object):boolean
  Interface für die Authentifizierung. Muss implementiert werden, um den Hash aus dem Authentifizierungsmerkmal mit dem realen Image Hash zu checken. Garantiert das Vorhandensein eines zugehörigen Images.

LambdaFacadeImpl



Beschreibung:
Es ist eine Fassadeklasse. Sie bekommt die Aufgaben vom ApiController und verbindet sich mit den anderen Klassen, die in Fassade sind.

Attribute:

- path: String
  Die Directory, wo die LambdaImages gespeichert werden.
- maxTimes: int
- timeout: long
- authHandler: AuthHandler
- runTime: RuntimeController

Methoden:

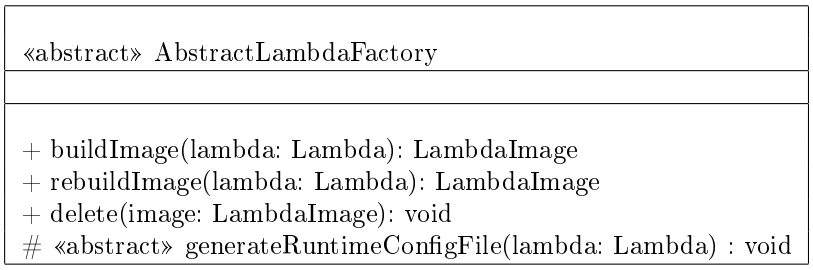
- upload(config: UploadLambdaRequest): String
  Bekommt ein UploadLambdaRequest Objekt. Mit Hilfe der RequestServerConverter Methode wird ein Lambda Objekt aus einem UploadLambdaRequest Objekt erzeugt. Dann wird das Lambda Objekt an RuntimeController übergeben, um ein LambdaImage zu bauen.
- update(name: String, config: UploadLambdaRequest): String
  Bekommt den Name eines Lambda, das aktualisiert werden soll, und ein UploadLambdaRequest Objekt. Mit Hilfe von RequestServerConverter Methode wird ein Lambda Objekt von ein UploadLambdaRequest Objekt erzeugt. Danach wird nach dem Lambda mit dem übergebenen

Namen gesucht. Falls gefunden, wird das Lambda gelöscht und das neue erzeugte Lambda an RuntimeController übergegeben.

- getLambda(name: String): Lambda
  Es wird nach dem Lambda mit dem übergebenen Namen gesucht. Falls gefunden, wird die getLambda Methode das gefundene Lambda zurückgeben.
- execute(name: String, config: ExecuteLambdaRequest): String
  Bekommt den Namen eines Lambda, das ausgeführt sein soll, und ein ExecuteLambdaRequest Objekt. Mit Hilfe der RequestServerConverter Methode wird ein ExecuteConfig Objekt aus einem ExecuteLambdaRequest Objekt erzeugt. Danach wird nach dem Lambda mit dem übergebenen Namen gesucht. Falls gefunden, wird das Lambda mit ExecuteConfig ausgeführt.
- deleteLambda(name: String): String
  Bekommt den Namen eines Lambda, das gelöscht werden soll. Es wird nach dem Lambda mit dem übergebenen Namen gesucht. Falls gefunden, wird es gelöscht.
- authenticate(principal:Object):boolean
  Parst ein AccessRights Objekt, ruft in der Runtime den AuthHandler auf, um den Hashcode und Identifier zu bekommen und prüft die Werte mit dem übergebenen Objekt.

1.3.2 ServiceLayer: LambdaRuntime

AbstractLambdaFactory



Beschreibung: Die AbstrakteFabrik Klasse eines Abstrakte-Fabrik-Entwurfsmusters zur Erzeugung von LambdaImages.

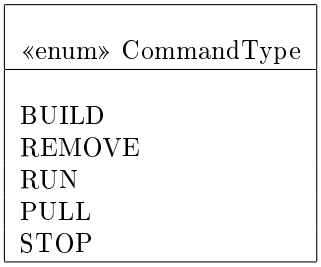
Attribute:

- keine

Methoden:

- buildImage(lambda: Lambda): LambdaImage
  Gibt ein Befehl an RuntimeCommunikator um ein Image zu erzeugen.
- rebuildImage(lambda: Lambda): LambdaImage
  Gibt ein Befehl an RuntimeCommunikator um alte Image zu löschen und ein neues zu bauen.
- delete(image: LambdaImage): void
  Gibt ein Befehl an RuntimeCommunikator um ein Image zu löschen.
- generateRuntimeConfigFile(lambda: Lambda): void
  Erzeugt Configaration für ein neues Image.

CommandType

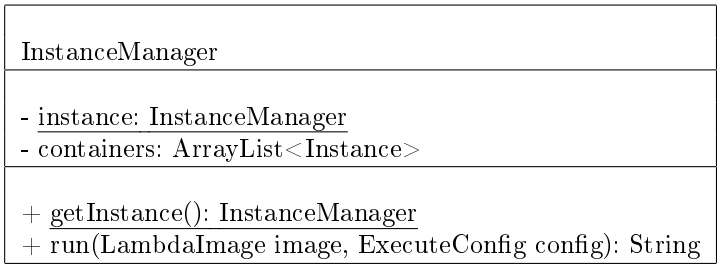


Beschreibung: Enumeration für die verschiedenen Befehlstypen, die innerhalb der Runtime-Umgebung benötigt werden.

Konstanten:

- BUILD
  bauen eines Images
- REMOVE
  löschen eines Images
- RUN
  starten einer Instanz eines Images
- PULL
  download der für ein Image benötigten Komponenten
- STOP
  stoppen einer Instanz

InstanceManager



Beschreibung:
Klasse zur Steuerung von Instanzen (starten, stoppen, ...). Die Klasse wird als Singleton implementiert.

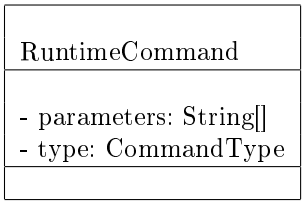
Attribute:

- instance: InstanceManager
  Eine Referenz auf das einzige Objekt der Klasse.
- containers: ArrayList<Instance>
  Liste aller laufenden Instanzen.

Methoden:

- getInstance(): InstanceManager
  Gibt die Referenz auf das einzige Objekt von InstanceManager zurück.
- run(LambdaImage image, ExecuteConfig config): String
  Erzeugt ein neues Objekt der Klasse Instanz, fügt es zu instances hinzu und führt die Instanz aus.

RuntimeCommand

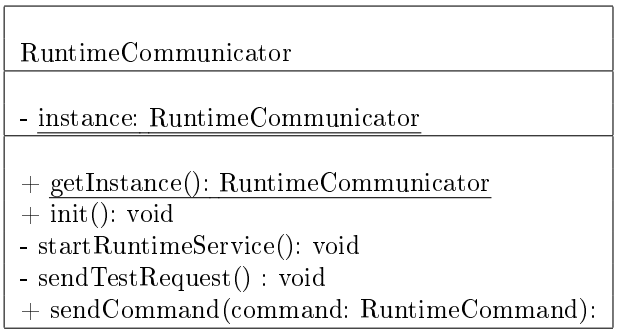


Beschreibung:
Kapselung der Daten, die benötigt werden, um ein Kommando an die Runtime-Umgebung zu senden.

Attribute:

- type: CommandType
  Art des Befehls
- parameters: String[]
  Eine Liste von Parameter zum Befehl.

RuntimeCommunicator



Beschreibung:
Die RuntimeCommunicator-Klasse übernimmt die Kommunikation mit der verwendeten Runtime-Umgebung. Die Klasse wird als Singleton implementiert.

Attribute:

- instance: RuntimeCommunicator
  Eine Referenz auf das einzige Objekt der Klasse.

Methoden:

- getInstance(): RuntimeCommunicator
  Gibt die Referenz auf das einzige Objekt von RuntimeCommunicator zurück.
- init(): void
  Initialisiert die Runtime-Umgebung mithilfe startRuntimeService() und sendTestRequest().

- startRuntimeService(): void
  Startet die Runtime-Umgebung
- sendTestRequest(): void
  Prüft, ob die Runtime-Umgebung korrekt funktioniert, indem testweise eine Verbindung hergestellt wird.
- sendCommand(command: RuntimeCommand): String
  Sendet einen Befehl an die Runtime-Umgebung.

RuntimeController

RuntimeController
- instance: RuntimeController - communicator: RuntimeCommunicator - factories: AbstractLambdaFactory[] - globalTimeLimit: int - instanceManager: InstanceManager
+ getInstance(): RuntimeController + init(): void + buildImage(lambda: Lambda): LambdaImage + rebuildImage(lambda: Lambda): LambdaImage + deleteImage(image: LambdaImage): void + setGlobalTimeLimit(limit: int): void + run(image: LambdaImage, config: ExecuteConfig): ExecuteLambdaResponce + getInstance(): RuntimeController - loadImageFactories(): void

Beschreibung:
Die Klasse RuntimeController ist eine Fassade, die alle Details des Runtime-Systems versteckt. Alle Methoden rufen nur die Methoden in den jeweils zuständigen Klassen auf. Die Klasse wird als Singleton implementiert.

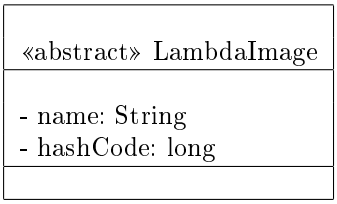
Attribute:

- instance: RuntimeController
  Eine Referenz auf das einzige Objekt der Klasse.
- communicator: RuntimeCommunicator
  Eine Referenz auf den RuntimeCommunicator.
- factories: AbstractLambdaFactory[]
  Die Fabriken zur Erzeugung von Lambdas.
- globalTimeLimit: int
  Die maximale Laufzeit einer LambdaFunktion.
- instanceManager: InstanceManager
  Die Referenz auf den InstanceManager.

Methoden:

- getInstance(): RuntimeController
  Gibt die Referenz auf das einzige Objekt von RuntimeController zurück.
- init(): void
  Initialisiert das Runtime-System.
- buildImage(lambda: Lambda): LambdaImage
  leitet den Aufruf weiter an die entsprechende ImageFactory.
- rebuildImage(lambda: Lambda): LambdaImage
  leitet den Aufruf weiter an die entsprechende ImageFactory.
- deleteImage(image: LambdaImage): void
  leitet den Aufruf weiter an die entsprechende ImageFactory.
- setGlobalTimeLimit(limit: int)
  setzt eine maximal Ausführungszeit für ein Lambda.
- run(image: LambdaImage, config: ExecuteConfig): ExcuteLambdaResponse
  leitet den Aufruf weiter an den InstanceManager.
- loadImageFactories(): void
  lädt und initialisiert die ImageFactory für jede unterstützte Sprache.

LambdaImage



Beschreibung:
Abstrakte Klasse für die Speicherung von Informationen über die angelegten Lambdas. Enthält die Informationen, die zum Starten einer Instanz notwendig sind.

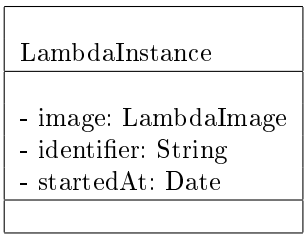
Attribute:

- identifier: String
  Der Identifikator für ein Image innerhalb der Runtime-Umgebung.
- hashCode: long
  Der HashCode, der dem Image innerhalb der Runtime-Umgebung zugeordnet wurde.

Methoden:

- keine

LambdaInstance



Beschreibung: Eine laufende Instanz einer Lambda-Funktion.

Attribute:

- image: LambdaImage
  Eine Referenz auf das zugehörige Image der laufenden Instanz
- identifier: String
  Der Identifikator für eine laufende Instanz innerhalb der Runtime-Umgebung.
- startetAt: Date
  Der Zeitpunkt, an dem die Instanz gestartet wurde. Wird benötigt, um die Instanz bei Überschreiten des Zeitlimits abbrechen zu können.

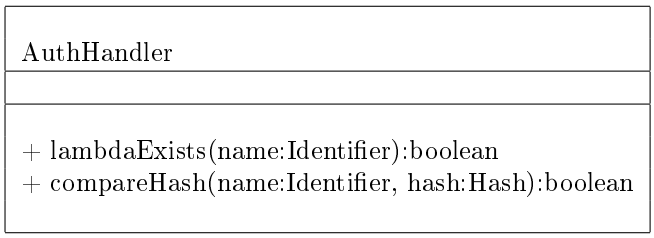
Methoden:

- keine

Python3LambdaImage und Python3LambdaFactory

Beschreibung:
Diese beiden Klassen sind Teil des AbstrakteFabrik-Entwurfsmusters zur Produktion von LambdaImages. Python3LambdaImage und Python3LambdaFactory sind die nicht-abstrakte Fabrik und das nicht-abstrakte LambdaImage für Lambdas in der Programmiersprache Python (Version 3).
Die Verwendung von Python3 wird unterstützt. Die Software kann um weitere Sprachen erweitert werden. Dafür müssen für jede Sprache eine Unterklasse von AbstractLambdaFactory und von LambdaImage implementiert werden (z. B. JavaScriptLambdaFactory und JavaScriptLambdaImage).

AuthHandler



Beschreibung:
Klasse zur Bereitstellung von Information über die Runtime für die finale Authentifizierung.

Attribute:

- keine

Methoden:

- compareHash(name:Identifier, hash:Hash):boolean
  Vergleicht bekommenen Hash mit dem des RuntimeImages unter bestimmten Namen.
- lambdaExists(name:Identifier):boolean
  Prüft, ob ein Lambda mit dem Namen vorhanden ist.

Exceptions

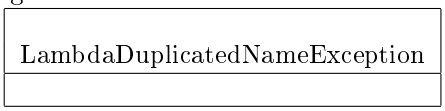
Beschreibung:
Exception-Klasse für die Angabe ungültiger Parameter für die Sprache eines Lambdas.

Beschreibung:
Exception-Klasse für Fehler, die mit der Runtime-Umgebung zusammenhängen (v.a. Verbindungsfehler)

1.3.3 Exception

LambdaExceptions

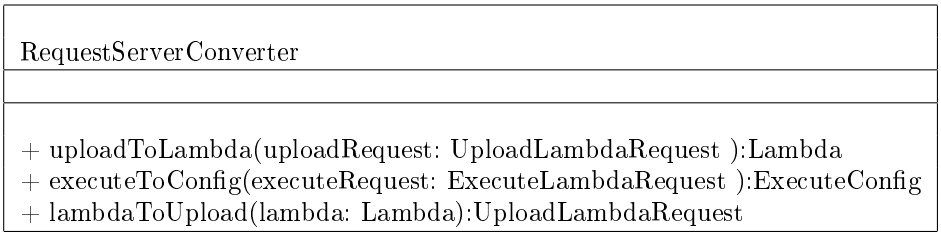
Beschreibung:
Exception, falls das Lambda nicht gefunden wird.



Beschreibung:
Exception, falls ein Lambda mit gleichem Namen schon existiert.

1.3.4 Converter

RequestServerConverter



Beschreibung:
Die Klasse macht die Syntax Überprüfung. Sie verbindet die lambda Pakete und die messages Pakete.

Attribute:

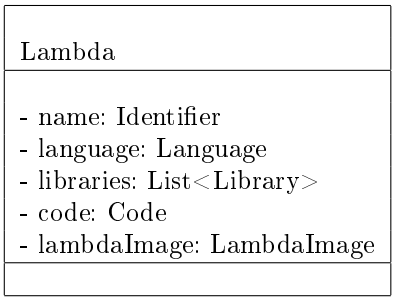
- keine

Methoden:

- uploadToLambda(uploadRequest: UploadLambdaRequest):Lambda
  Erzeugt ein Lambda Objekt aus einem UploadLambdaRequest Objekt.
- executeToConfig(executeRequest: ExecuteLambdaRequest):ExecuteConfig
  Erzeugt ein ExecuteConfig Objekt aus einem ExecuteLambdaRequest Objekt.
- lambdaToUpload(lambda: Lambda):UploadLambdaRequest
  Erzeugt ein UploadLambdaRequest Objekt aus einem Lambda Objekt.

1.3.5 Lambda

Lambda



Beschreibung:
Klasse zur objektorientierten Modellierung der Lambda-Funktion.

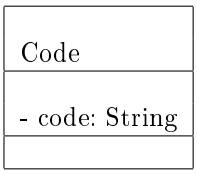
Attribute:

- name: Identifier
  Name der Funktion.
- language: Language
  Sprache der Funktion.
- libraries: List<Library>
  Libraries, die die Funktion zur Ausführung benötigt.
- code: Code
  Die Funktion selbst.
- lambdaImage: LambdaImage
  Eine Referenz auf das zugehörige Image der Lambda.

Methoden:

- keine

Code



Beschreibung:
Klasse zur Verpackung der Code von der Lambda-Funktion.

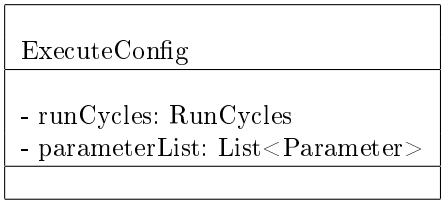
Attribute:

- code: String
  Der Code der Funktion als String.

Methoden:

- keine

ExecuteConfig



Beschreibung:
Klasse zur Verpackung der Ausführungskonfigurationen.

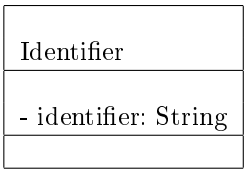
Attribute:

- runCycles: RunCycles
  Wie oft die Funktion ausgeführt werden muss.
- parameterList: List<Parameter>
  Parameter, die die Funktion zur Ausführung benötigt.

Methoden:

- keine

Identifier



Beschreibung:
Klasse zur Verpackung des Funktionsnamens.

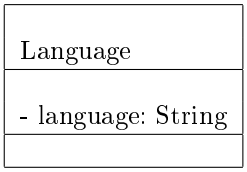
Attribute:

- identifier: String
  Name der Funktion.

Methoden:

- keine

Language



Beschreibung:
Klasse zur Verpackung einer Programmiersprache.

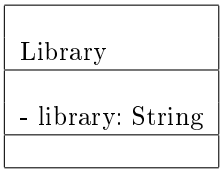
Attribute:

- language: String
  Sprache der Funktion.

Methoden:

- keine

Library



Beschreibung:
Klasse zur Verpackung einer Bibliothek.

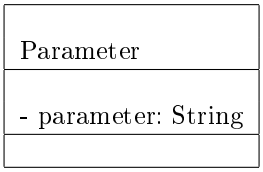
Attribute:

- library: String
  Name der Bibliothek als String.

Methoden:

- keine

Parameter



Beschreibung:
Klasse zur Verpackung eines Parameters der Lambda-Funktion.

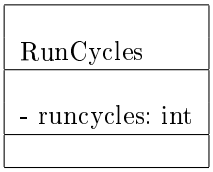
Attribute:

- parameter: String
  Ein Parameter als String.

Methoden:

- keine

RunCycles



Beschreibung:
Klasse zur Verpackung der Anzahl der Ausführungszyklen

Attribute:

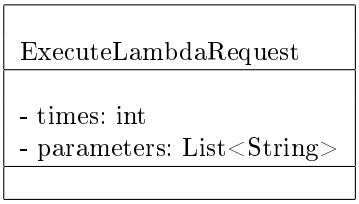
- runCycles: int
  Wie oft die Funktion ausgeführt wird.

Methoden:

- keine

1.3.6 Messages

ExecuteLambdaRequest



Beschreibung:
Enthält die in Java Attribute geparste JSON Datei, die die Anfrage zum Ausführen enthält.

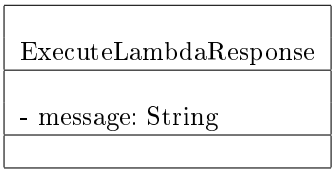
Attribute:

- time: int
  Die Anzahl von Ausführungen.
- parameters: List<String>
  Die Liste von Typen, welche die Parameter bei der Ausführung haben.

Methoden:

- keine

ExecuteLambdaResponse



Beschreibung:
Enthält die in Java Attribute geparste JSON Datei, die die Antwort nach dem Ausführen enthält.

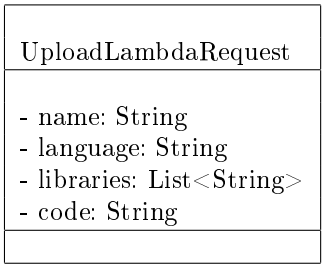
Attribute:

- message: String
  Die Antwort, die der Nutzer nach der Ausführung bekommt.

Methoden:

- keine

UploadLambdaRequest



Beschreibung:
Enthält die in Java Attribute geparste JSON Datei, die die Anfrage zum Hochladen enthält.

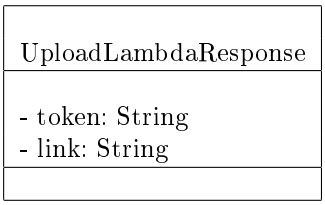
Attribute:

- name: String
  Der Name des Lambda.
- language: String
  Die Programmiersprache des Lambda.
- libraries: List<String>
  Die Liste von Biblioteken.
- code: String
  Der Code des Lambda als String.

Methoden:

- keine

UploadLambdaResponse



Beschreibung:
Enthält die in Java Attribute geparste JSON Datei, die die Antwort nach dem Hochladen enthält.

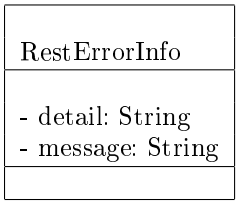
Attribute:

- token: String
  Der Token, der für die weitere Authentifizierung benutzt wird.
- link: String
  Der Link, unter dem das Lambda gespeichert wird.

Methoden:

- keine

RestErrorInfo

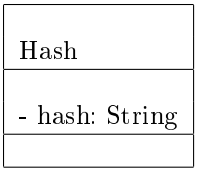


Beschreibung:
Die Klasse verallgemeinert die Ausgabe von Exceptions.

Attribute:

- detail: String
  Die Nachricht von Exception.
- message: String
  Die Nachricht für Entwickler.

Hash



Beschreibung:
Wrapperklasse für den Hashcode. Nur benötigt für Authentifizierung.

Attribute:

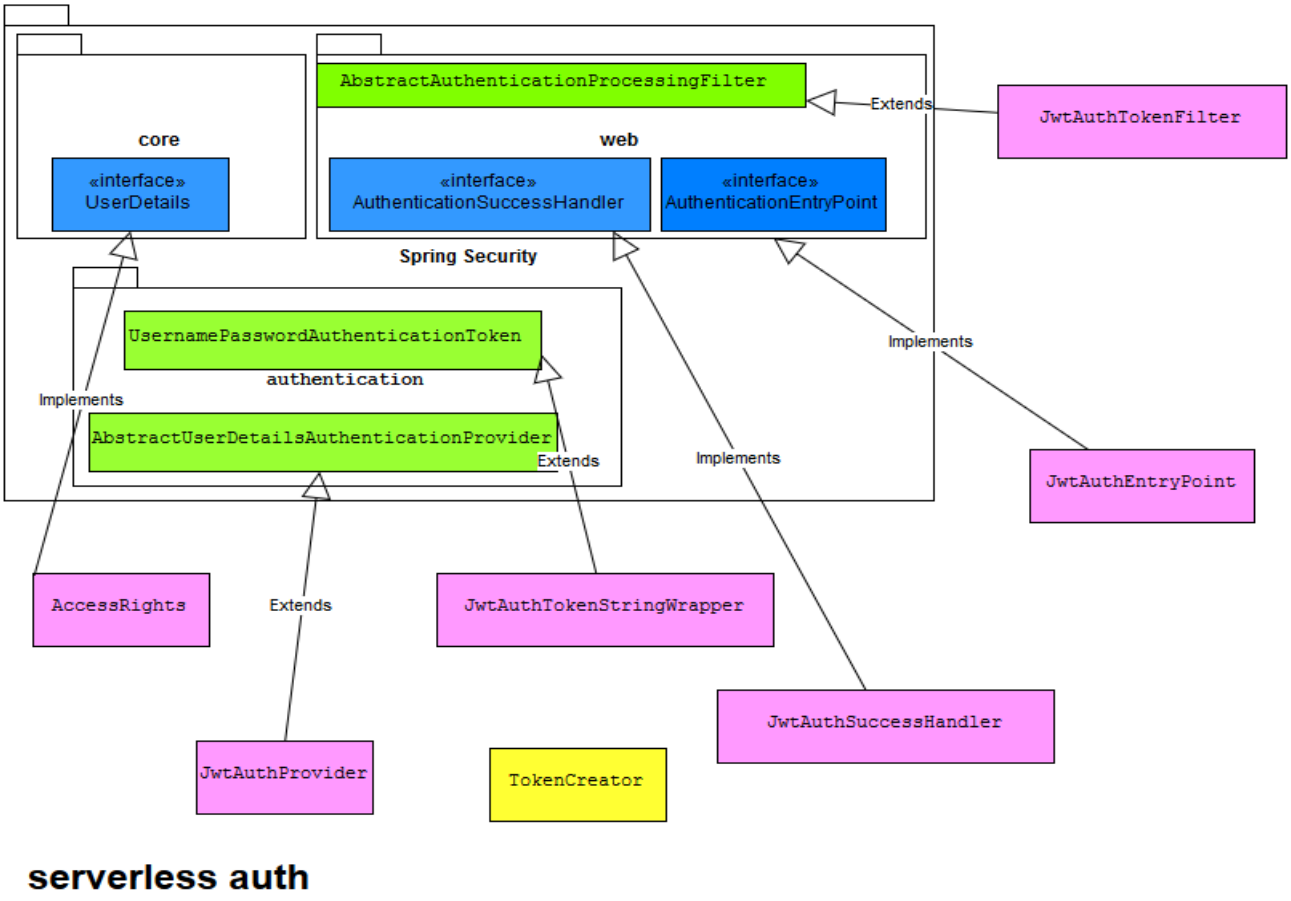
- hash: String
  Enthält den Hash

Methoden:

- Getter und Setter

1.4 Detaillierte Klassenbeschreibung: Auth

Die Authentifizierung besteht aus zwei Teilen. Dieser Auth Teil beschreibt das Zusammenspiel mit Spring Security, welches vor dem Eintritt in die Fassade passiert. Hier wird das übergebene JSON Webtoken auf Syntax und zeitliche Gültigkeit geprüft.
Die Konfiguration der REST Pfade, auf der die Authentifizierung gelten soll findet außerhalb des Pakets in AuthenticationConfig statt.
Der zweite Teil hinter der Fassade geschieht mit Zusammenspiel der Runtime und der Fassadenklasse. Diese prüfen die Werte aus dem Token (z.B. Hashcode) mit den zugehörigen Images und erlauben Zutritt bei korrekten Werten. Hier wird nur der erste Teil beschrieben.



JwtAuthTokenFilter

JwtAuthTokenFilter
- tokenHeader: String
+ attemptAuthentification(request:HttpServletRequest, response:HttpServletResponse): Authentification # successfulAuthentification(request:HttpServletRequest, response:HttpServletResponse, chain:FilterChain, authResult:Authentication): void

Beschreibung:
Kümmert sich um den Beginn der Authentifizierung und um die Weiterleitung der Anfrage bei erfolgreicher Authentifizierung. Im Fall einer nicht erfolgreichen Authentifizierung werden in der Oberklasse AbstractAuthenticaionProcessingFilter von Spring Security behandelt. Für Näheres dazu siehe die Spring Security Dokumentation.

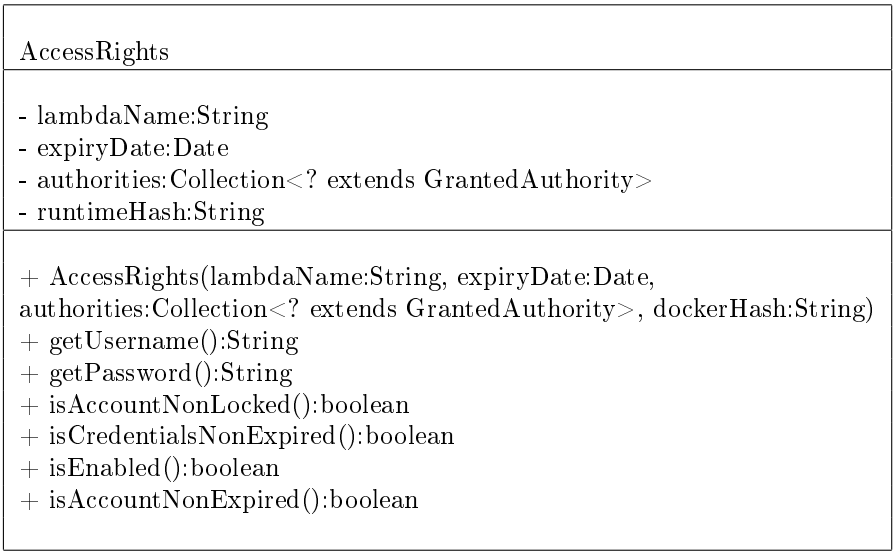
Attribute:

- tokenHeader: String
  Der Wert unter dem Eintrag jwt.header aus der Datei application.yml im Ordner resources wird gelesen und mithilfe von Spring in das Attribut geschrieben. Es beschreibt die Header in der REST-Anfrage, die als Authorisierungsmerkmal akzeptiert werden. Empfohlen ist " Authorization" unter diesem Eintrag.

Methoden:

- attemptAuthentification(request:HttpServletRequest, response:HttpServletResponse): Authentification
  Beginn des Authentifizierungsablaufs. Spring füttert die Methode mit den Parametern und ruft diese auf. Erstellt wird in der Methode ein Objekt JwtTokenStringWrapper, welches den übergebenen Token als Objekt darstellt, ruft mithilfe des AbstractAuthenticationProcessingFilters den AuthenticationManager von Spring auf und authentifiziert mithilfe dessen und dem JwtTokenStringWrapper Objekt die Anfrage. Als Rückgabe wird ein Authentification-Objekt übergeben, welches genauer spezifiziert ein Objekt der Klasse AccessRights ist. Da diese Methode eine Methode der Oberklasse überschreibt, kann dieses jedoch nicht so genau spezifiziert werden.
- successfulAuthentification(request:HttpServletRequest, response:HttpServletResponse, chain:FilterChain, authResult:Authentication): void
  Falls die Authentifizierung erfolgreich war, wird diese Methode von Spring aufgerufen. Die Methode führt den Anfragenfluss fort, als wäre keine Authentifizierung da gewesen.

AccessRights



Beschreibung:
Implementierung des Spring Security Interfaces UnserDetails. Wird erstellt nach der Prüfung des Authentifizierungsmerkmals in TokenCreator, um ein Objekt zu haben, das den Status eines Merkmals angibt und mit dem in der Abfolge weitergearbeitet werden kann.

Attribute:

- lambdaName:String
  Name der Lambdafunktion (Identifier).
- authorities:Collection<? extends GrantedAuthority>
  Objekte der Klasse GrantedAuthority, die in dem Subtoken (SUB) oder Mastertoken (MASTER) aus den Werten des Authentifizierungsmerkmals generiert werden
- runtimeHash:String
  Hash des Runtime Images, übergeben vom Authentifizierungsmerkmal. Wird später mit dem realen Hash des Images unter dem gegebenen Namen entgegengeprüft.
- expiryDate:Date
  Datum, an dem das Authentifizierungsmerkmal ungültig wird.

Methoden:

- AccessRights(lambdaName:String, expiryDate:Date, authorities:Collection<? extends GrantedAuthority>, dockerHash:String)
  Konstruktor
- getUsername():String
  Stummelmethode. Gibt immer null zurück.
- getPassword():String
  Stummelmethode. Gibt immer null zurück.
- isAccountNonLocked():boolean
  Stummelmethode. Gibt immer true zurück.
- isCredentialsNonExpired():boolean
  Prüft, ob der Wert von expiryDate vor dem aktuellem Datum liegt. Gibt true zurück, falls dies der Fall ist, sonst false.
- isEnabled():boolean
  Stummelmethode. Gibt immer true zurück.
- isAccountNonExpired():boolean
  Stummelmethode. Gibt immer true zurück.
- Getter und Setter

JwtAuthEntryPoint

JwtAuthEntryPoint
+ commence(request:HttpServletRequest, response:HttpServletResponse, authException:AuthentificationException): void

Beschreibung:
Klasse zur Behandlung von Unauthorisierten Anfragen. Darunter fallen der Aufruf von Pfaden, die nicht freigegeben worden sind und Aufrufe ohne Authentifizierungsmerkmale, welche benötigt werden.

Attribute:

- keine

Methoden:

- commence(request:HttpServletRequest, response:HttpServletResponse, authException:AuthentificationException): void
  Sendet eine Fehlermeldung mit 401 Unauthorized.

JwtAuthProvider

JwtAuthProvider
- tokenCreator: TokenCreator
+ supports(authentification:Class<?>):boolean # additionalAuthentificationChecks(userDetails:UserDetails, authentification:UsernamePasswordAuthentificationToken):void # retrieveUser(username:String, authentication:UsernamePasswordAuthentificationToken): UserDetails

Beschreibung:
Unterklasse der AbstractUserDetailsAuthenticationProvider von Spring Security. Leitet die Anfrage des AuthenticationManagers weiter, um die Gültigkeit des Authentifizierungsmerkmals zu prüfen.

Attribute:

- tokenCreator: TokenCreator
  Objekt der Klasse TokenCreator, welche für das Prüfen und Erstellen von JWT verantwortlich ist

Methoden:

- supports(authentification:Class<?>):boolean
  Gibt die Klasse zurück, die benutzt wird, um das Authentifizierungsmerkmal weiterzuleiten.
- additionalAuthentificationChecks(userDetails:UserDetails, authentification:UsernamePasswordAuthentificationToken):void
  Tut nichts und überschreibt damit die Implementierung von Spring Security.
- retrieveUser(username:String, authentication:UsernamePasswordAuthentificationToken):UserDetails
  Ruft tokenCreator auf und prüft damit das Authentifizierungsmerkmal entgegen. Damit ist es allerdings noch nicht vollständig geprüft und muss noch weiter gegen den Hashcode des Runtime Images geprüft werden.

JwtAuthSuccessHandler

JwtAuthSuccessHandler
+ onAuthenticationSuccess(request:HttpServletRequest, response:HttpServletResponse, authentication:Authentication): void

Beschreibung:

Implementiert den von Spring Security benötigten AuthenticationSuccessHandler .

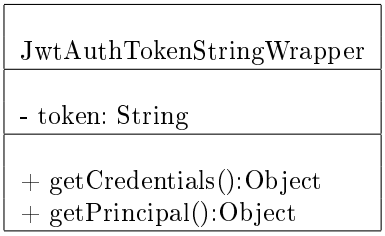
Attribute:

- keine

Methoden:

- onAuthenticationSuccess(request:HttpServletRequest, response:HttpServletResponse, authentication:Authentication): void
  Tut nichts und überschreibt damit die Methode von Spring Security.

JwtAuthTokenStringWrapper



Beschreibung:
Unterklasse des UsernamePasswordAuthenticationToken von Spring Security. Zur Erstellung von Objekten, die das Authentifizierungsmerkmal tragen.

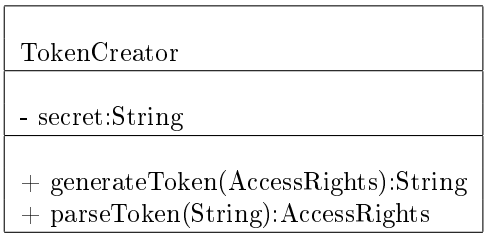
Attribute:

- token: String
  Authentifizierungsmerkmal

Methoden:

- getCredentials():Object
  Überschreibt die Methode in der Oberklasse. Gibt immer null zurück.
- getPrincipal():Object
  Überschreibt die Methode in der Oberklasse. Gibt immer null zurück.
- Getter

TokenCreator



Beschreibung:
Behandelt JWT mithilfe des vom Administrator gegebenen Geheimnis. Erstellt und parsed JWT.

Attribute:

- secret:String
  In applications.yml im Ordner resources gesetztes Geeheimnis um JWTs zu verschlüsseln. Wird automatisch von Spring geladen.

Methoden:

- generateToken(AccessRights):String
  Generiert ein JWT aus dem AccessRights Objekt.
- parseToken(String):AccessRights Parsed Information aus JWT in ein AccessRights Objekt. Das sagt alleine nicht genug über die Gültigkeit aus, diese muss mithilfe anderer Klassen noch geprüft werden.

Application

1.5 Detaillierte Klassenbeschreibung: ApiController

ApiController

ApiController
-lambdaFacade: LambdaFacadeImpl -tokenCreator: TokenCreator
+ uploadLambda(config: UploadLambdaRequest): ResponseEntity<UploadLambdaResponse> + updateLambda(name: String, config: UploadLambdaRequest): ResponseEntity<Void> + showLambda(name: String): ResponseEntity<UploadLambdaRequest> + deleteLambda(name: String): ResponseEntity<Void> + executeLambda(name: String, config: ExecuteLambdaRequest): ResponseEntity<ExecuteLambdaResponse> + generateSubtoken(name: String, expiryDate: String): ResponseEntity<String>

Beschreibung:
Bearbeitet Web-Anfragen, die der Nutzer schickt. Mit Hilfe von Spring Annotations werden die Methoden mit REST Anfragen verbindet. In manche Methoden wird vom Nutzer JSON Datei erwartet um die Anfrage zu bestimmen. Die Methoden geben entweder nur HTTP Status zurück oder noch JSON Datei dazu. ApiController ist was ähnliches zum View Teil, weil es die Anfragen vom Nutzer nimmt und sie weiter zur Fassade übergibt.

Attribute:

- lambdaFacade: LambdaFacadeImpl
  Objekt der Klasse LambdaFacadeImpl, welche für den Zugriff zur Klasse LambdaFacadeImpl ist.
- tokenCreator: TokenCreator
  Objekt der Klasse TokenCreator, welche für das Prüfen und erstellen von JWT verantwortlich ist.

Methoden:

- uploadLambda(config: UploadLambdaRequest): ResponseEntity<UploadLambdaResponse>
  Wird aufgeruft, wenn der Nutzer die Lambda hochladen will und dafür die POST Anfrage schickt. Nutzer gibt noch JSON Datei mit Konfigurationen für Hochladung dazu. Spring wird die Datei mit Konfigurationen parsen und als ein UploadLambdaRequest Objekt hier übergeben. Dann werden die weiteren Aufgaben an der LambdaFacade geleitet. Nach der Bearbeitung gibt die LambdaFacade ein String mit dem Token zurück. Dann wird ein Response gemacht. Am Ende wird HTTP Antwort mit JSON File zurückgegeben.
- updateLambda(name: String, config: UploadLambdaRequest): ResponseEntity<Void>
  Wird aufgeruft, wenn der Nutzer die Lambda aktualisieren will und dafür die PUT Anfrage schickt. Nutzer gibt noch JSON Datei mit Konfigurationen für Hochladung der neuen Lambda und die Name der Lambda, die aktualisiert sein soll. Spring wird die Datei mit Konfigurationen parsen und als ein UploadLambdaRequest Objekt hier übergeben. Dann werden die weiteren Aufgaben an der LambdaFacade geleitet. Nach der Bearbeitung gibt die LambdaFacade ein String mit dem Status zurück. Am Ende wird HTTP Antwort zurückgegeben.
- showLambda(name: String): ResponseEntity<UploadLambdaRequest>
  Wird aufgeruft, wenn der Nutzer die Lambda Konfigurationen anschauen will und dafür die GET

Anfrage schickt. Er gibt noch die Name der Lambda dazu. Es werden die weiteren Aufgaben an der LambdaFacade geleitet. Nach der Bearbeitung gibt die LambdaFacade ein Lambda Objekt zurück. Am Ende wird mit Hilfe von der lambdaToUpload Methode ein UploadLambdaRequest von Lambda gemacht und mit HTTP Antwort zurückgegeben.

- deleteLambda(name: String): ResponseEntity<Void>
  Wird aufgeruft, wenn der Nutzer die Lambda löschen will und dafür die DELETE Anfrage schickt. Nutzer gibt noch die Name der Lambda dazu. Es werden die weiteren Aufgaben an der LambdaFacade geleitet. Nach der Bearbeitung gibt die LambdaFacade ein String mit dem Status zurück. Am Ende wird HTTP Antwort zurückgegeben.

- executeLambda(name: String, config: ExecuteLambdaRequest):
  ResponseEntity<ExecuteLambdaResponse>
  Wird aufgeruft, wenn der Nutzer die Lambda ausführen will und dafür die POST Anfrage schickt. Nutzer gibt noch die Name der Lambda und JSON Datei mit Konfigurationen für Ausführung dazu. Spring wird die Datei mit Konfigurationen parsen und als ein ExecuteLambdaRequest Objekt hier übergeben. Es werden die weiteren Aufgaben an der LambdaFacade geleitet. Nach der Bearbeitung gibt die LambdaFacade ein String mit dem Status zurück. Dann wird ein Response gemacht und mit HTTP Antwort zurückgegeben.

- generateSubtoken(name: String, expiryDate: String): ResponseEntity<String>
  Wird aufgeruft, wenn der Nutzer die Tokens generieren will und dafür die GET Anfrage schickt. Nutzer gibt noch die Name der Lambda und expiryDate dazu. Es wird hier ein AccessRights Objekt erzeugt. Dann wird mit Hilfe von JWT Bibliothek die Werte vom erzeugten Objekt genommen und in einen Token gespeichert. Am Ende wird HTTP Antwort mit Token zurückgegeben.

1.5.1 Exception

ExceptionsHandler

ExceptionsHandler
+ handleJwtMalformedException(ex:JwtMalformedException): ResponseEntity + handleNoJwtGivenException(ex:NoJwtGivenException): ResponseEntity + handleLanguageNotSupportedException(ex:LanguageNotSupportedException): ResponseEntity + handleRuntimeConnectException(ex:RuntimeConnectException): ResponseEntity + handleLambdaNotFoundException(ex:LambdaNotFoundException): ResponseEntity + handleLambdaDuplicatedNameException(ex:LambdaDuplicatedNameException): ResponseEntity + handleExceptions(Exception ex): ResponseEntity

Beschreibung:
Die Klasse erzeugt HTTP Status und verständliche Nachrichten anhand des Exception Typs. Für jede Exception soll eine eigene Methode sein.

Attribute:

- keine

Methoden:

- handleJwtMalformedException(ex:JwtMalformedException): ResponseEntity

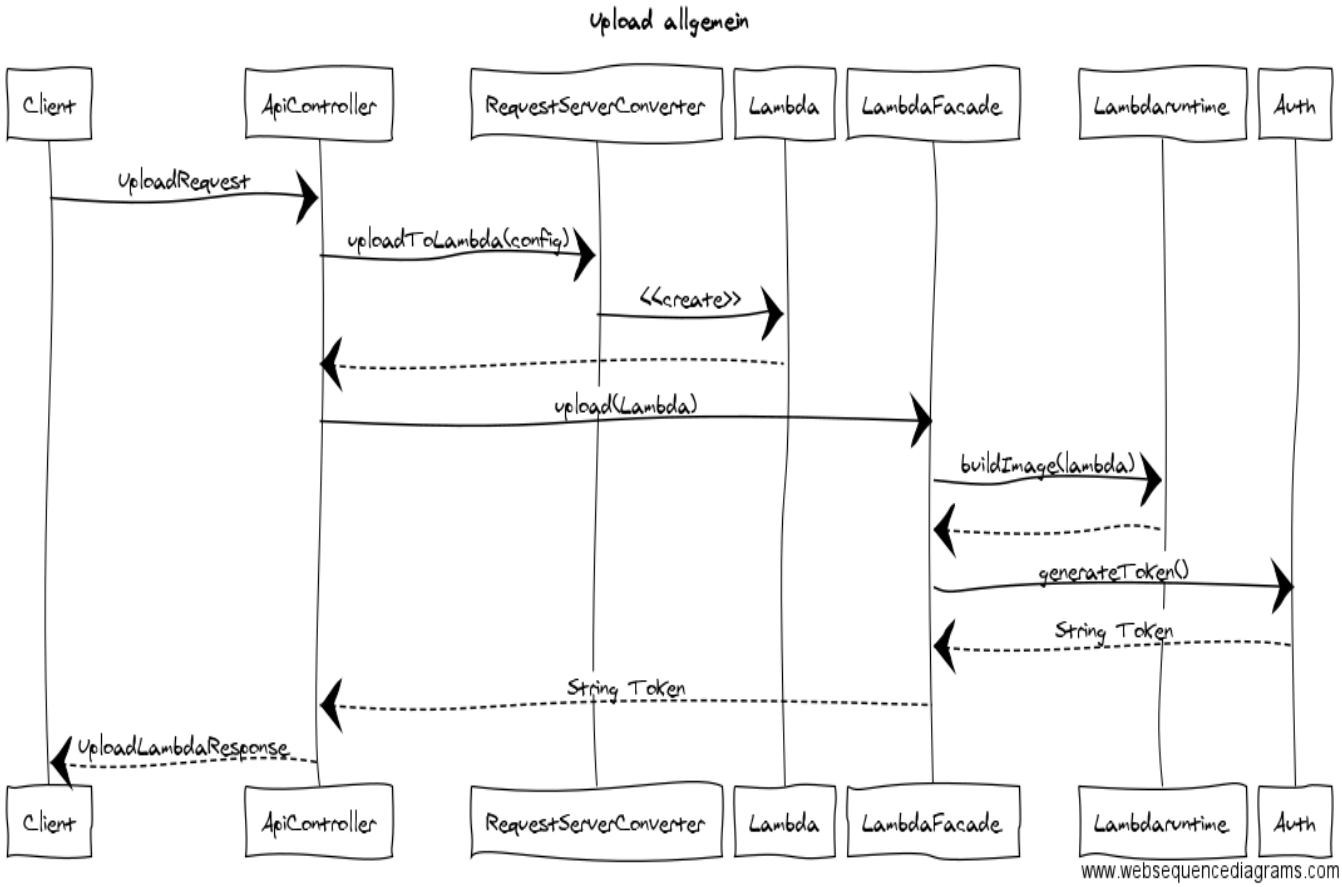
- handleNoJwtGivenException(ex:NoJwtGivenException): ResponseEntity
- handleLanguageNotSupportedException(ex:LanguageNotSupportedException): ResponseEntity
- handleRuntimeConnectException(ex:RuntimeConnectException): ResponseEntity
- handleLambdaNotFoundException(ex:LambdaNotFoundException): ResponseEntity
- handleLambdaDuplicatedNameException(ex:LambdaDuplicatedNameException): ResponseEntity
- handleExceptions(Exception ex): ResponseEntity

Kapitel 2

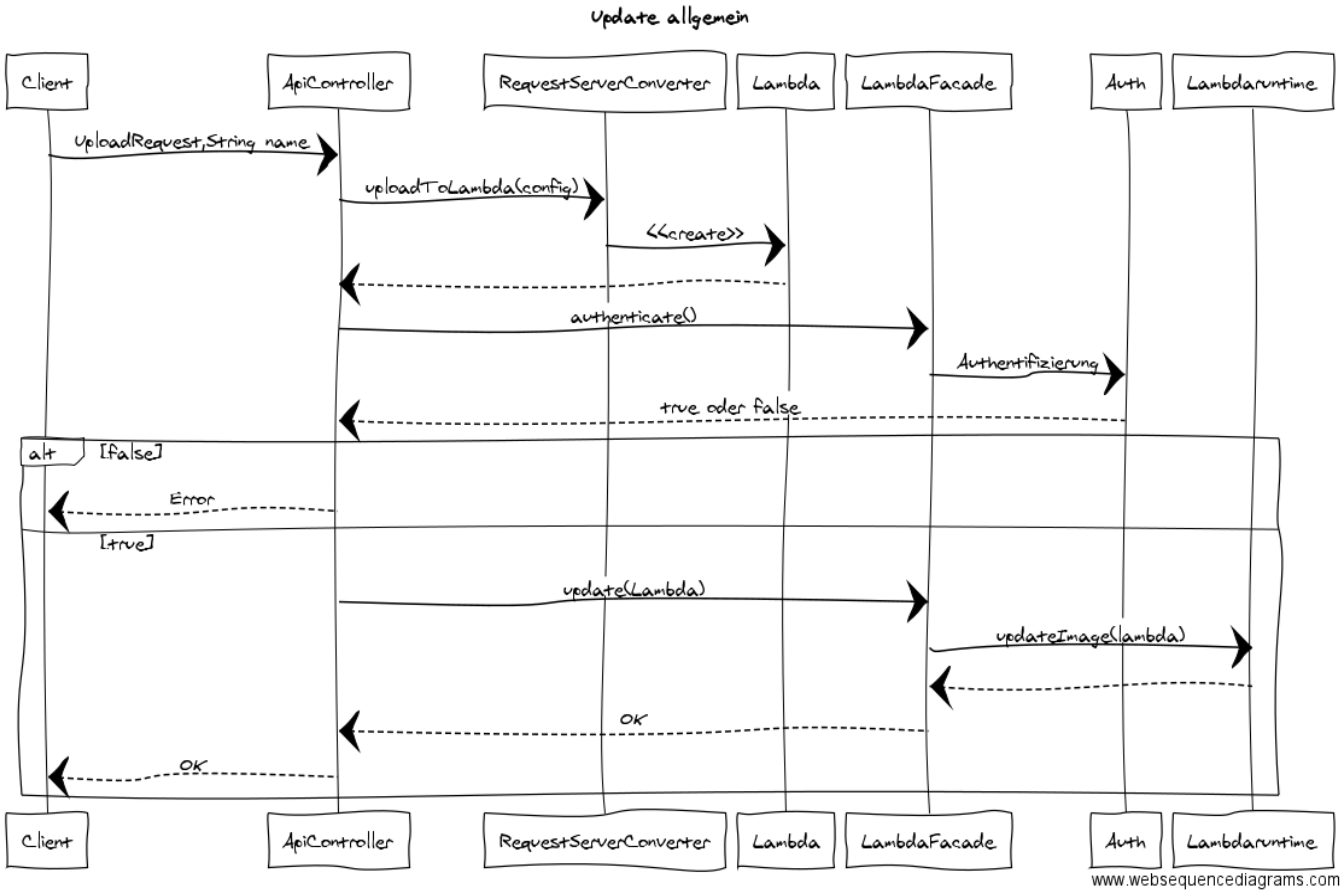
Sequenzdiagramme

2.1 Allgemeiner Ablauf

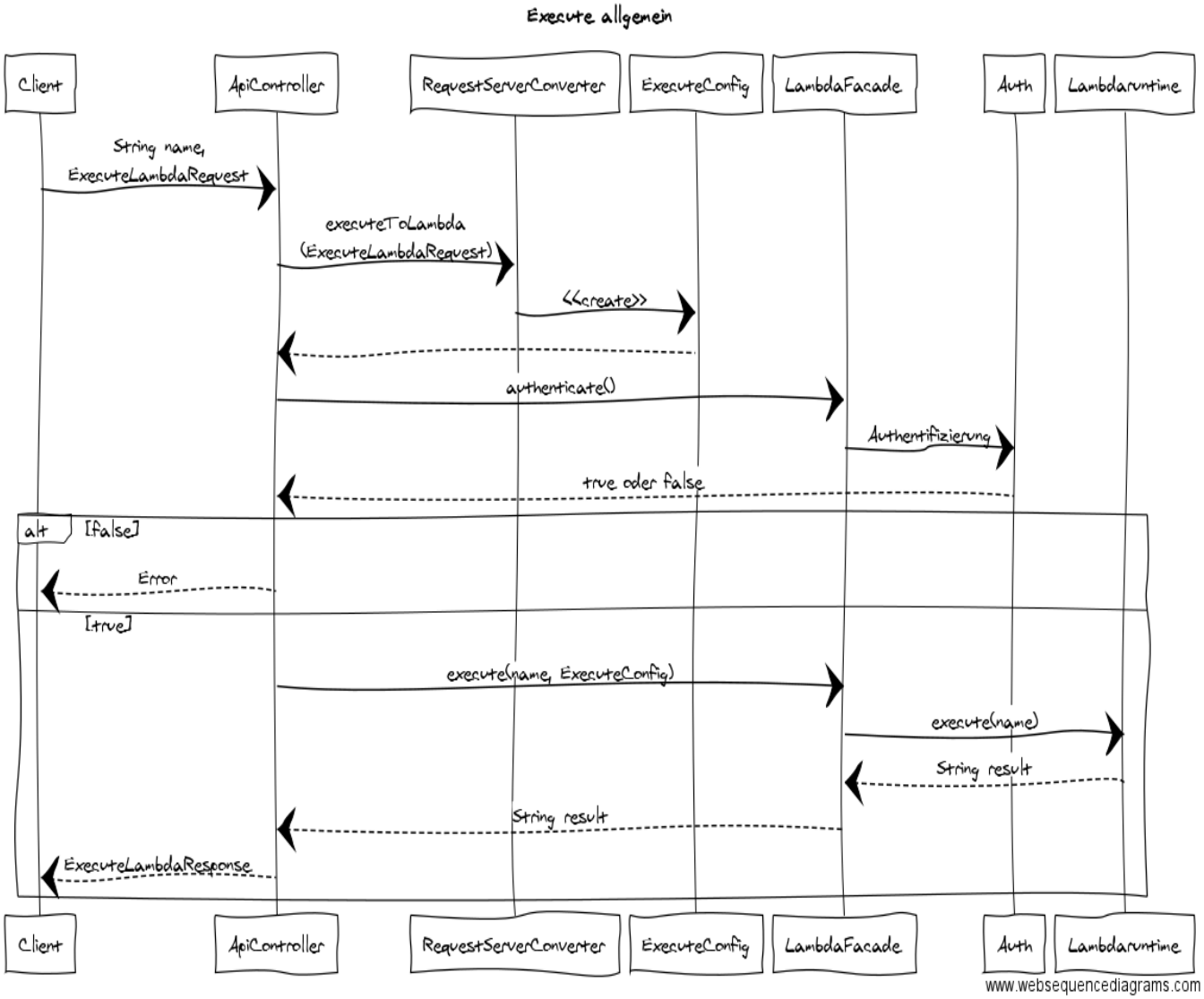
2.1.1 Hochladen



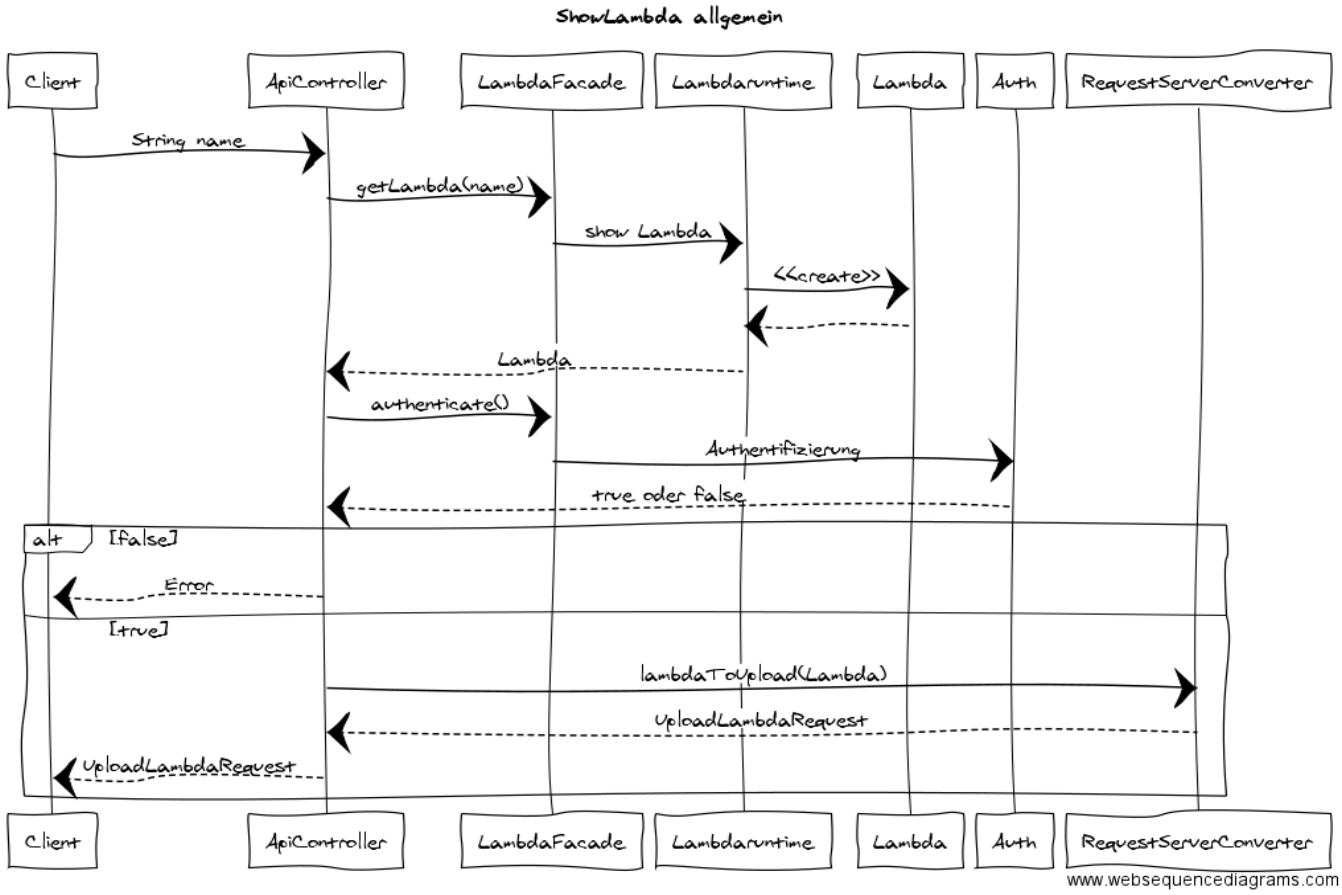
2.1.2 Aktualisieren



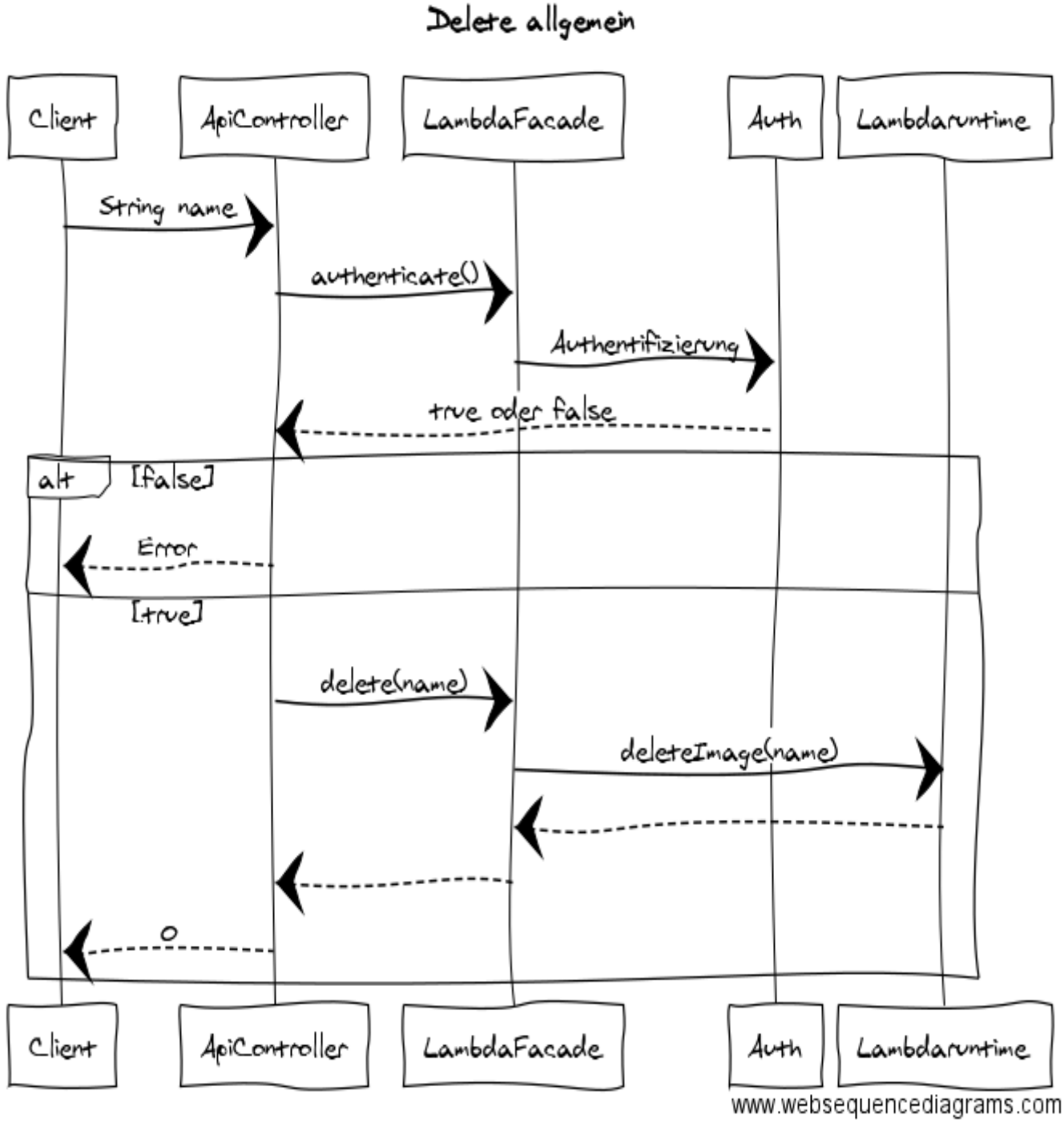
2.1.3 Ausführen



2.1.4 Lambda anzeigen

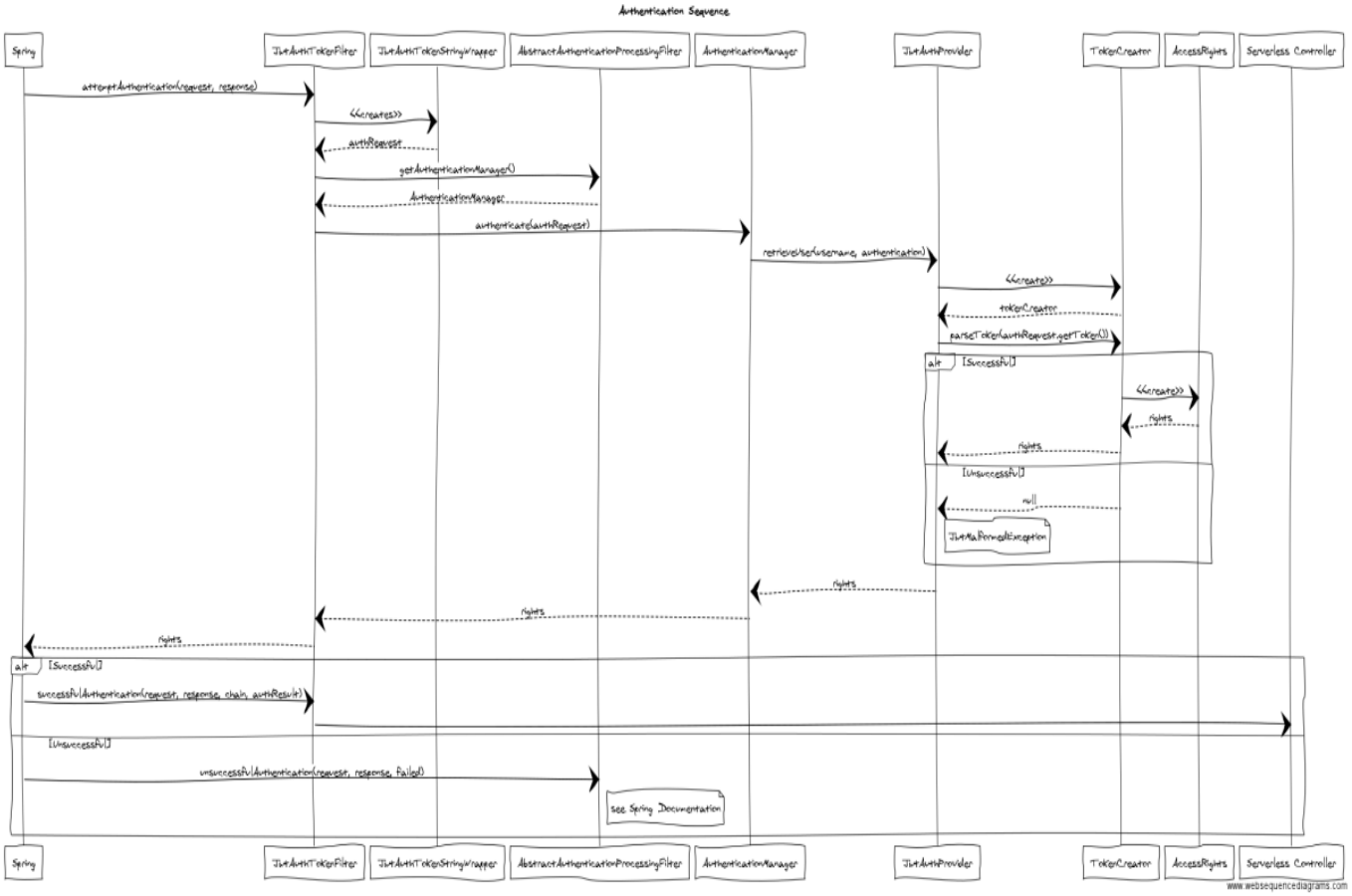


2.1.5 Löschen

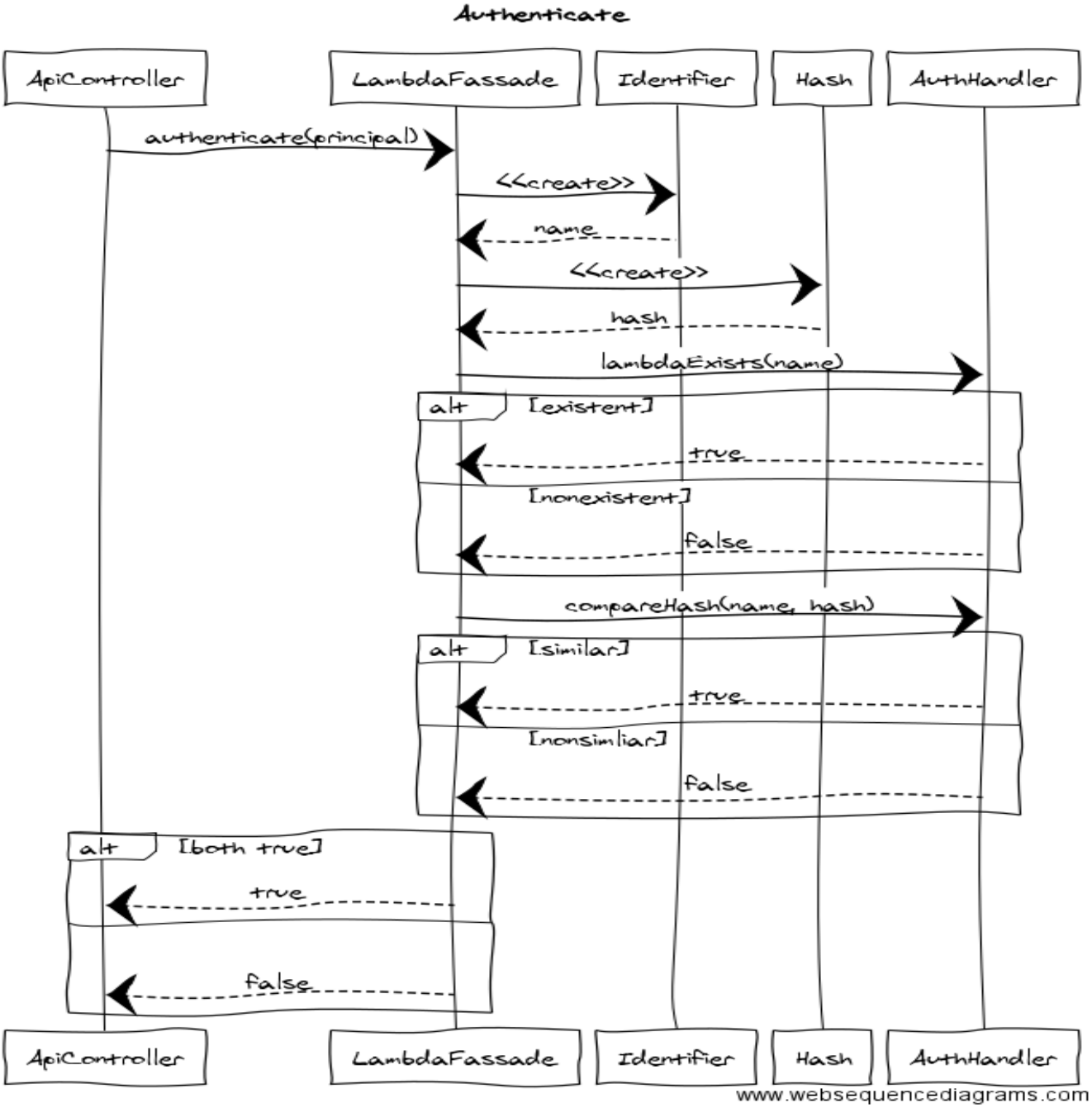

Delete allgemein
Client
ApiController
LambdaFacade
Auth
Lambdaruntime
String name
authenticate()
Authentifizierung
true oder false
alt
[false]
Error
[true]
delete(name)
deleteImage(name)
0
Client
ApiController
LambdaFacade
Auth
Lambdaruntime
www.websequencediagrams.com

2.2 Authentifizierung

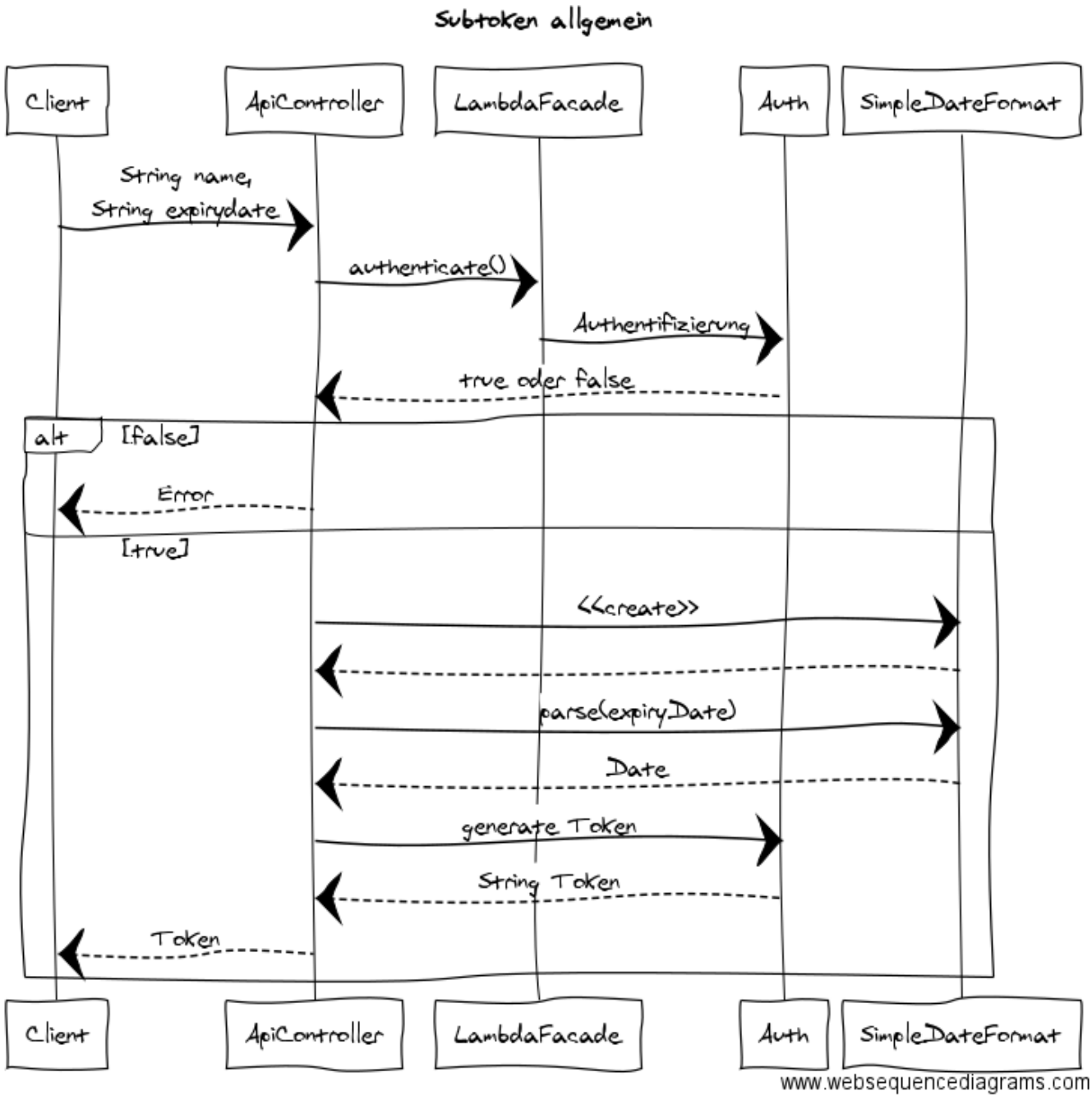
2.2.1 Spring Security Anbindung



2.2.2 Runtime Anbindung

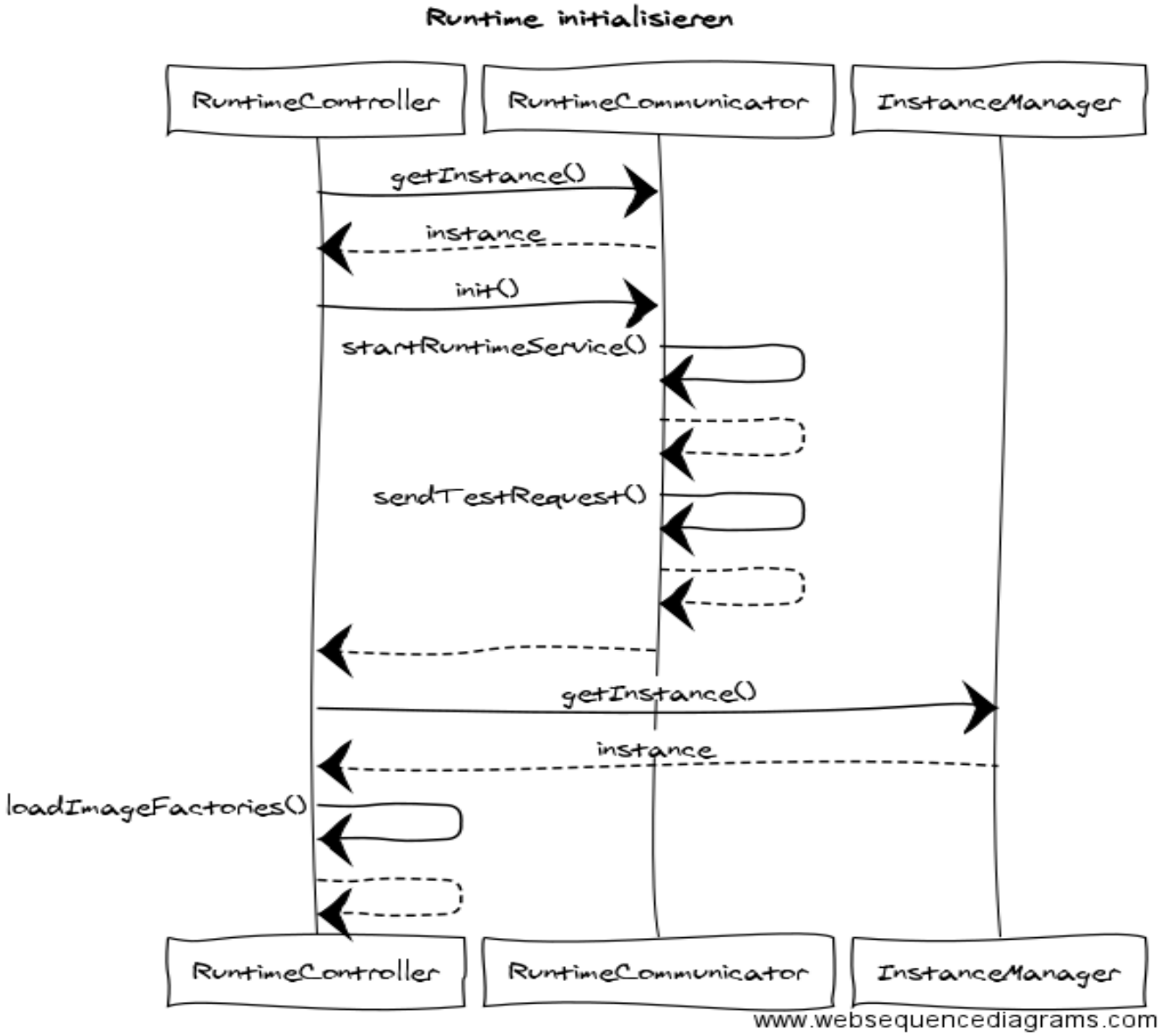


2.2.3 Subtoken erstellen

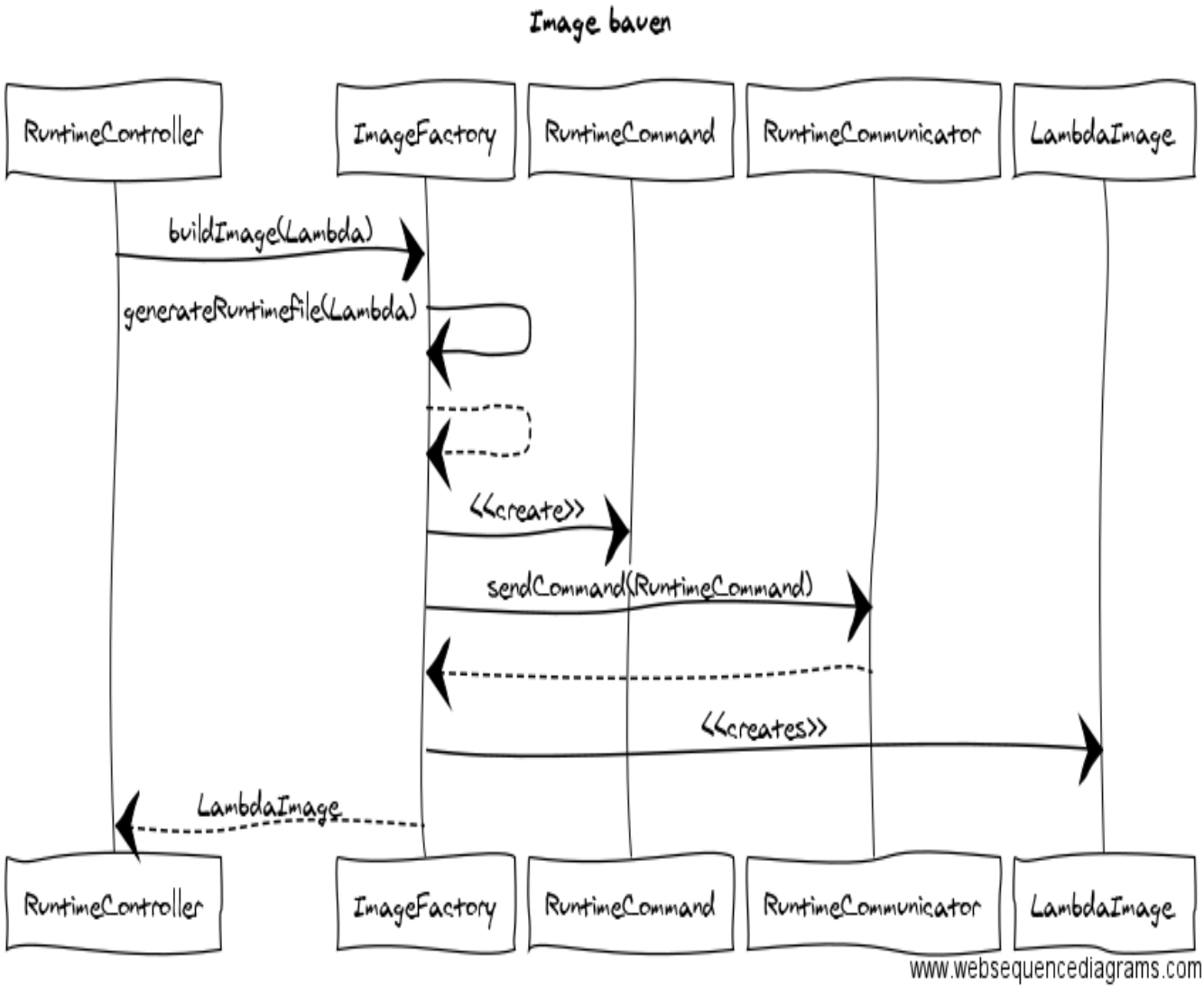


2.3 Runtime

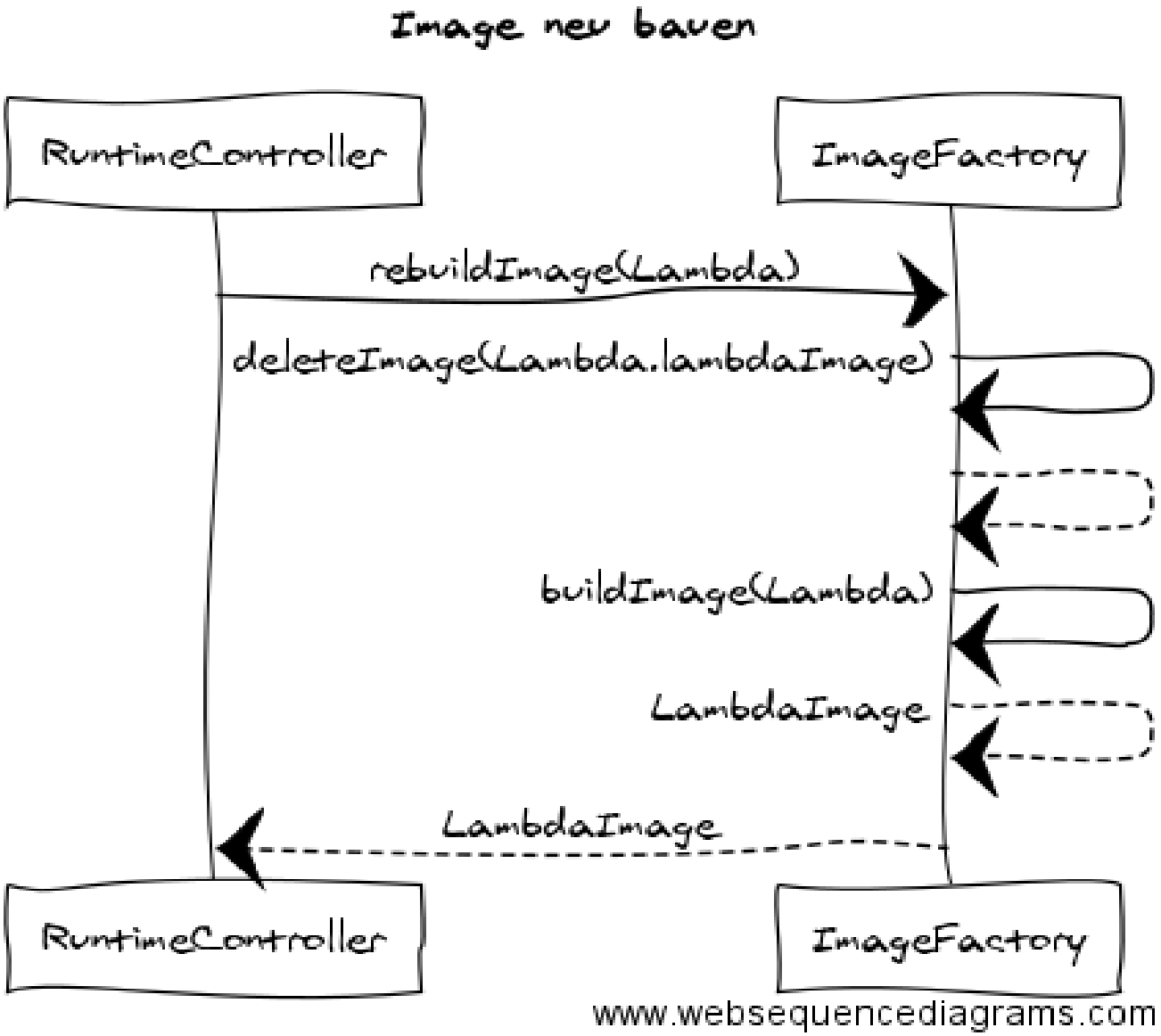
2.3.1 Initialisierung



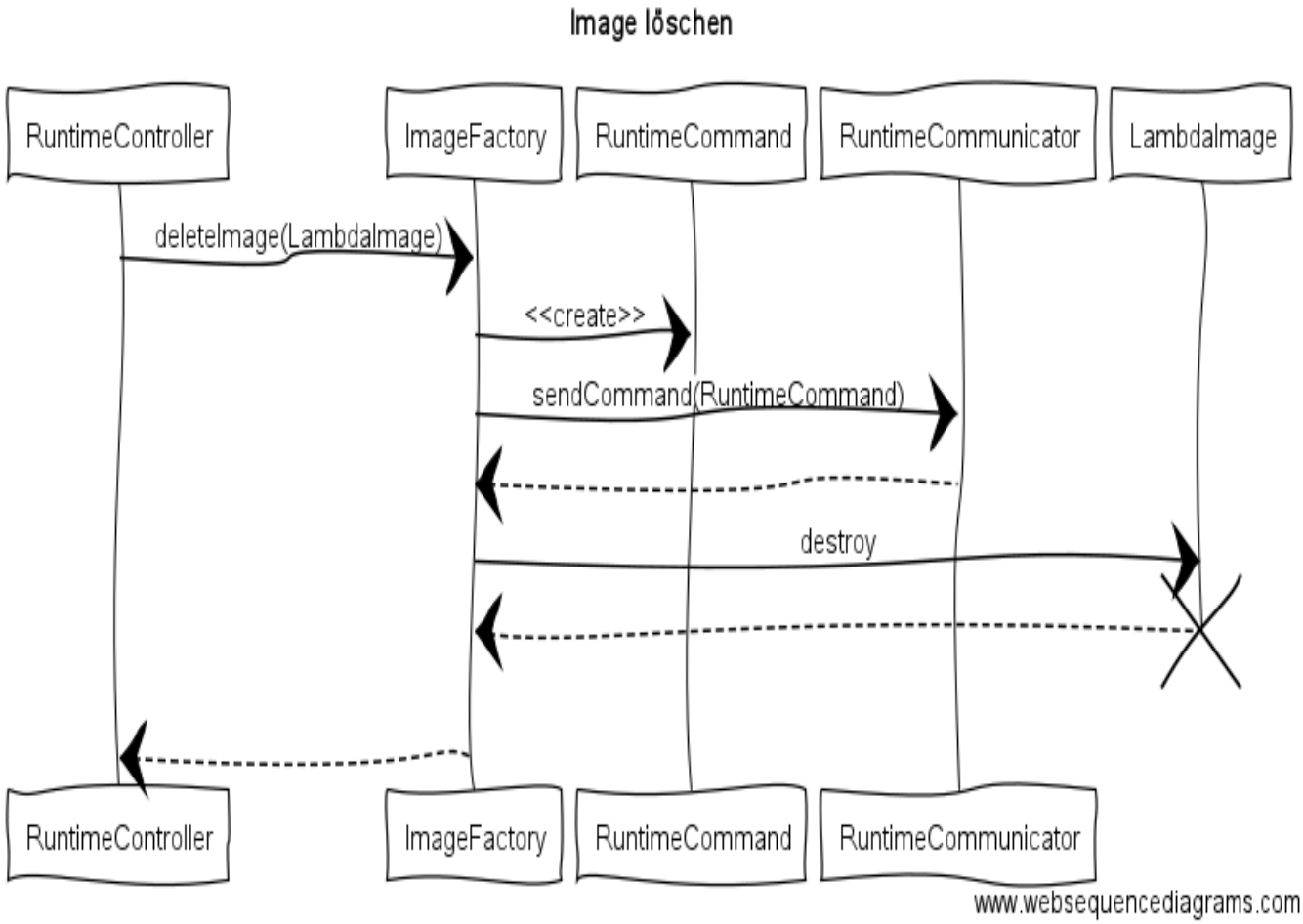
2.3.2 Image bauen



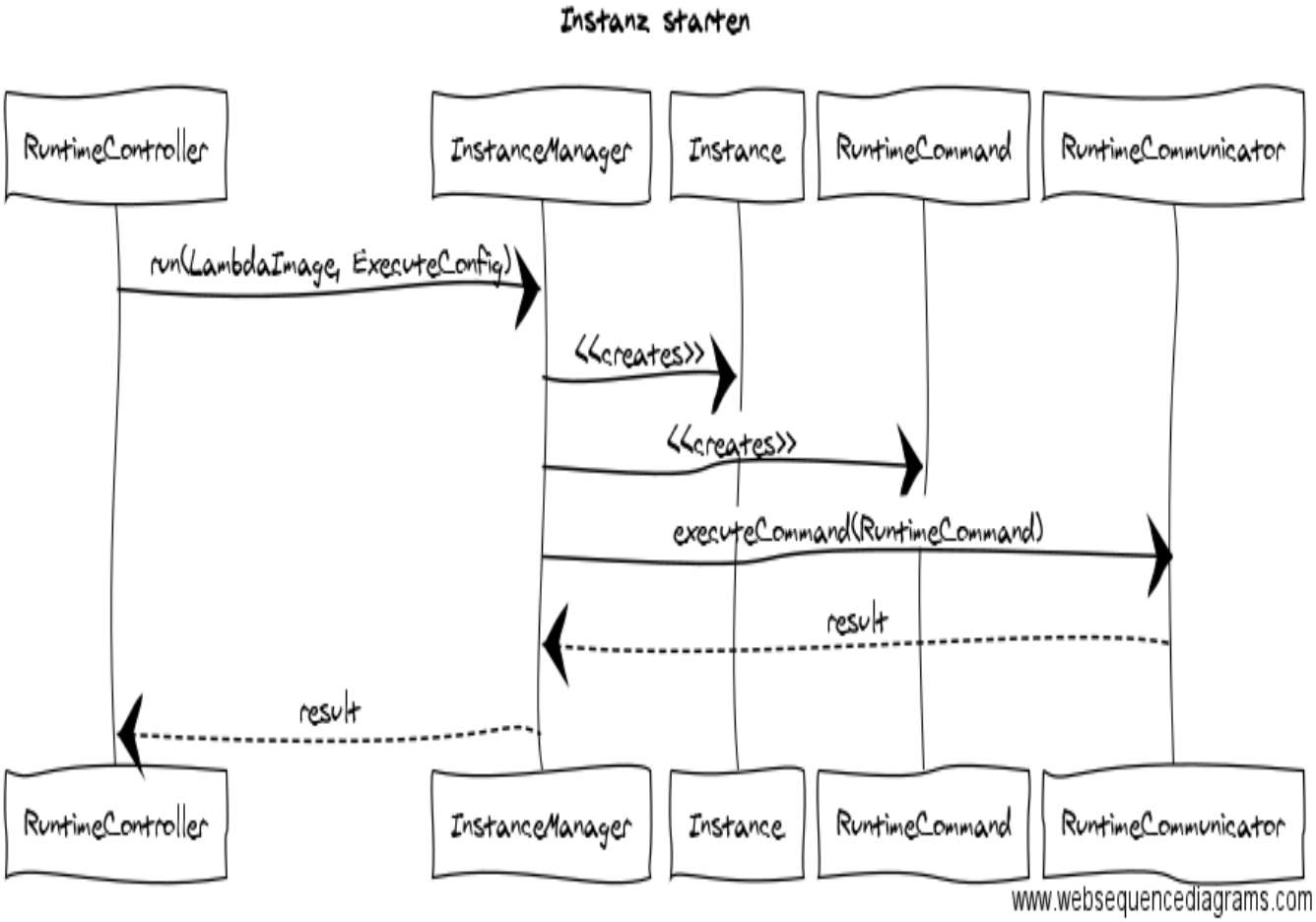
2.3.3 Image umbauen



2.3.4 Image löschen



2.3.5 Image starten



Kapitel 3

Dateiformate

In diesem Kapitel werden die Bedingungen für JSON-Dateien beschrieben, die der Benutzer für API-Befehle eingeben muss. Hier wird das Format des Parameters "config"beschrieben. Falls irgendeine Bedingung nicht erfüllt wird, bekommt der Nutzer sofort eine Fehlermeldung mit einer Beschreibung drüber, was falsch eingegeben wurde.

3.1 Hochladen/Anpassen von Lambda-Funktionen: Konfigurationsdatei

Diese **JSON**-Datei muss beim Nutzer eingegeben werden, um eine Funktion hochzuladen/zu ändern. Diese Datei beschreibt die Einstellungen einer Funktion, die der Nutzer hochladen will.

```
{
"name": <Name_of_Lambda>,
"language": <Language_of_Lambda>,
"parameters_input": [<type0>, <type1>],
"libraries":[library0, library1],
"code": <Code>
}
```

- **name** - Name der Lambda-Funktion. Dieser Parameter muss dem folgenden regulären Ausdruck entsprechen: [A-Za-z_]+[0-9]*, z.B. *foo21, _foo, Func12_ 12*
- **language** - Programmiersprache der Lambda-Funktion, z.B. *Java, Python2, C#*
- **parameters_input** - Parametertypen der Eingaben der Lambda-Funktion, z.B. *String, int, double[]* .
- **libraries** - Bibliotheken, die die Lambda-Funktion benutzt, z.B. *math.h, swing, JavaUtil*
- **code** - Text der Lambda-Funktion, z.B. *print("Hello Serverless");*

3.2 Ausführung von Lambda-Funktionen

Diese **JSON**-Datei muss beim Nutzer eingegeben werden, um eine Funktion auszuführen. Diese Datei beschreibt die Einstellungen für die Ausführung einer Funktion.

```
{
"times": <number_of_times_running>,
"parameters_input":[<input0>,<input1>]
}
```

- **times** - Anzahl von Ausführungen der Funktion, z.B. *1, 3, 100*
- **parameters_input** - Eingabe, die eine Funktion bei der Ausführung bearbeiten muss, z.B. *1,2, true.*

3.3 Generierung von Tokens

Diese **JSON**-Datei muss beim Nutzer eingegeben werden, um einen Subtoken zu erstellen. Diese Datei beschreibt die Einstellungen für Tokengenerierung.

```
{
  "expires": time
}
```

- **expires** - Zeit in Minuten, die beschreibt wie lange ein Subtoken gültig sein wird, z.B. *23.*