



UNIVERSITÀ DI PARMA

DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE E INFORMATICHE
Corso di Laurea in Informatica

TESI DI LAUREA

AUTENTICAZIONE TRAMITE TOKEN IN UN'ARCHITETTURA A MICROSERVIZI: REALIZZAZIONE DI UN PLUGIN CUSTOM PER L'API GATEWAY KONG

Candidato:
Valerio Desiati

Relatore:
Prof. Roberto Alfieri

Correlatore:
Gregorio Palamà

Anno Accademico 2022/2023

Dedica

Indice

Introduzione	1
1 Descrizione dello stage	3
1.1 Azienda	3
1.1.1 Servizi offerti	3
1.1.2 Organizzazione	3
1.1.3 Metodologie aziendali	4
1.2 Progetto di stage	4
1.2.1 Ripartizione del lavoro svolto	5
1.3 Obiettivi	7
2 Tecnologie utilizzate	9
2.1 Git	9
2.2 Java	9
2.3 Eclipse	10
2.4 Framework Spring	10
2.4.1 Spring Boot	11
2.4.2 Spring Data	11
2.5 Azure DevOps	11
2.6 Docker	12
2.7 Kong Gateway	12
2.8 PostgreSQL	13
2.9 Lua	14
2.10 JWT	15
2.11 Postman	16
3 Analisi dei requisiti	17
3.1 Casi d'uso	17
4 Progettazione e sviluppo	19
4.1 Introduzione ai microservizi	19

4.2	Il microservizio <code>CheckEmail</code>	21
4.2.1	Sviluppo	22
4.3	Introduzione a Kong Gateway	27
4.3.1	Service	27
4.3.2	Route	27
4.3.3	Consumer	28
4.3.4	Plugin	28
4.4	Architettura del plugin <code>checkemail</code>	28
4.4.1	Sviluppo	29
4.5	Configurazione Kong Gateway	31
5	Testing	33
	Conclusione	37
A	Appendice	41

Elenco delle figure

1.1	Logo di AESYS S.r.l.	4
2.1	Logo di Git	9
2.2	Logo di Java	10
2.3	Logo di Eclipse	10
2.4	Logo di Spring	11
2.5	Logo di Azure	12
2.6	Logo di Docker	12
2.7	Logo di Kong Gateway	13
2.8	Logo di PostgreSQL	14
2.9	Logo di Lua	14
2.10	Logo di JWT	15
2.11	Logo di Postman	16
4.1	Architettura monolitica e architettura a microservizi a confronto	20
4.2	Schema di comunicazione Utente - microservizio	22
4.3	Modello Entità Relazione del Database	23
4.4	Diagramma UML delle classi principali del progetto	24
4.5	Diagramma UML delle classi <i>controller</i> del progetto	25
4.6	Diagramma UML delle <i>custom exception</i> del progetto	26

Elenco degli algoritmi

1	Utilizzo di un <i>model</i> di Spring Boot JPA	25
2	Accesso al Database	27
3	Suddivisione token JWT	29
4	Parse token JWT	30
5	Inoltro richiesta HTTP dal plugin	30
6	Inoltro risposta dal plugin a Kong Gateway	31
7	Script per la configurazione di Kong Gateway	32

Elenco delle tabelle

1.1	Ripartizione settimanale del lavoro svolto	6
1.2	Ripartizione oraria del lavoro svolto	6
4.1	Codici HTTP restituiti dal microservizio	22

Introduzione

L'introduzione deve contenere un riassunto del lavoro di Tesi. In particolare bisogna esprimere chiaramente e molto sinteticamente: contesto dello studio, motivazioni, contributo e conclusioni. Bisogna quindi fare un sommario dello studio ad alto livello, fornendo le intuizioni senza ricadere in dettagli tecnici. Anche lo stile dovrebbe essere più discorsivo rispetto alle parti tecniche della tesi.

Capitolo 1

Descrizione dello stage

In questo capitolo saranno esposte brevemente tutte le caratteristiche e le dinamiche dello stage svolto, che saranno approfondite nei capitoli successivi.

1.1 Azienda

AESYS S.r.l. è un'azienda fondata nel 2013 a Pescara (PE) che si occupa di sviluppo software e consulting con sedi a Pescara e Torino. Ad oggi AESYS conta più di 240 dipendenti.

Nel corso degli anni l'azienda ha vissuto una grande crescita e ora dispone di figure specializzate in molteplici campi.

1.1.1 Servizi offerti

AESYS è attiva in molti campi nell'ambito dell'*Information Technology* fornendo servizi per piattaforme Web e Mobile, in ambito DevOps, Cloud, Machine Learning e sviluppo UX/UI.

1.1.2 Organizzazione

AESYS è suddivisa in *Business Unit*, ognuna per campo di sviluppo.

In particolare, nella realizzazione di questo progetto sono state coinvolte due Business Unit: la RED, che si focalizza su tutte le tecnologie legate al linguaggio Java e alla JVM e la ORANGE, unità in ambito DevOps che fornisce supporto per manutenzione e monitoraggio dei sistemi su grandi infrastrutture.

1.1.3 Metodologie aziendali

AESYS adotta la metodologia di *sviluppo Agile* da diverso tempo, applicandone i principi nel quotidiano.

La metodologia di sviluppo Agile si basa su dei valori fondamentali:

- Il personale e le interazioni sono più importanti dei processi e degli strumenti.
- È meglio avere un software funzionante che una documentazione esaustiva.
- La collaborazione con il cliente è più importante della stipula di un contratto.
- Essere pronti al cambiamento.

Strumenti di supporto

In AESYS lo strumento più utilizzato per comunicare è Microsoft Teams.

Il software in questione permette di creare al proprio interno una vera e propria gerarchia aziendale, programmare meeting, avere agende condivise, instant messaging e tante altre risorse utili per lavorare in team. All'interno dell'azienda viene utilizzato Git come sistema di versionamento del software (argomento approfondito nel *Capitolo 2 - Tecnologie utilizzate*).



Figura 1.1: Logo di AESYS S.r.l.

1.2 Progetto di stage

Il progetto di stage prevede la realizzazione di un plugin per Kong Gateway scritto con il linguaggio Lua.

Il plugin prevede l'integrazione del gateway con un microservizio scritto in Java per verificare l'avvenuto acquisto di un modulo applicativo da parte di un utente identificato tramite token.

Il progetto comprende la realizzazione del microservizio e l'ottimizzazione del plugin tramite l'utilizzo di una cache locale a Kong.

Ovviamente è stato necessario acquisire delle competenze di base prima della realizzazione del progetto, come ad esempio acquisire una conoscenza sufficiente del sistema di Version Control Git.

Le competenze necessarie per lo svolgimento del progetto sono:

- Conoscenza della metodologia di sviluppo Agile (acquisita durante lo stage).
- Conoscenza del sistema di Version Control Git (acquisita durante lo stage).
- Conoscenza del linguaggio Java e della JVM (Java Virtual Machine) per la realizzazione del microservizio (acquisita durante gli studi Accademici).
- Conoscenza del framework Spring da utilizzare nello sviluppo del microservizio.
- Conoscenza della struttura e del funzionamento di Kong Gateway (acquisita durante lo stage).
- Conoscenza del linguaggio Lua per lo sviluppo del plugin (acquisita durante lo stage).
- Conoscenza degli strumenti per effettuare testing sul prodotto finito (acquisita durante lo stage).

1.2.1 Ripartizione del lavoro svolto

Lo stage aziendale ha avuto una durata di 225 ore, come previsto dal piano di studio Universitario per la Facoltà di Informatica. All'interno dell'azienda le ore sono state suddivise in cinque settimane lavorative full time, dal lunedì al venerdì dalle 09:00 alle 18:00.

Di seguito due tabelle che riassumono la ripartizione settimanale e oraria dello sviluppo del progetto.

CAPITOLO 1. DESCRIZIONE DELLO STAGE

Settimana	Attività svolte
Prima settimana	Esposizione delle specifiche del progetto. Acquisizione delle competenze preliminari necessarie quali Git e metodologia di sviluppo Agile.
Seconda settimana	Studio del framework Spring. Sviluppo del microservizio.
Terza settimana	Sviluppo del microservizio.
Quarta settimana	Testing del microservizio. Studio del funzionamento di Kong Gateway. Configurazione Kong Gateway.
Quinta settimana	Studio delle specifiche del plugin. Studio della sintassi e della semantica del linguaggio Lua. Sviluppo del plugin per Kong Gateway. Testing del prodotto finito.

Tabella 1.1: Ripartizione settimanale del lavoro svolto

Attività svolte	Ore impiegate
Esposizione delle specifiche del progetto	2
Acquisizione delle competenze preliminari necessarie quali Git e metodologia di sviluppo Agile	38
Studio del framework Spring	20
Sviluppo del microservizio	60
Testing del microservizio	5
Studio del funzionamento di Kong Gateway	30
Configurazione Kong Gateway	5
Studio delle specifiche del plugin	2
Studio della sintassi e della semantica del linguaggio Lua	12
Sviluppo plugin per Kong Gateway	20
Testing del prodotto finito	8

Tabella 1.2: Ripartizione oraria del lavoro svolto

1.3 Obiettivi

Gli obiettivi fissati dal Tutor Aziendale per lo sviluppo di questo progetto riguardano l'acquisizione di una conoscenza teorica e pratica di tutte le tecnologie utilizzate:

- Conoscenza dei sistemi di versionamento, nello specifico Git.
- Conoscenza del mondo dei microservizi, API Composition, Access token, Service Discovery.
- Conoscenza dei linguaggi Java e Lua.
- Conoscenza delle librerie e dei framework utilizzati durante lo sviluppo.
- Conoscenza di Kong Gateway.
- Effettuare code review su codice sorgente prodotto.
- Test End – to – End sul microservizio e sul plugin.

Capitolo 2

Tecnologie utilizzate

In questo capitolo saranno descritte le tecnologie utilizzate nella realizzazione del progetto.

2.1 Git

Git è un DVCS (Distributed Version Control Systems) gratuito, open source e distribuito, utilizzabile da riga di comando, che consente di effettuare il controllo versione per un progetto.

Data la sua natura “distribuita” *Git* è basato su flussi di lavoro simultanei con i quali diversi sviluppatori possono collaborare ad un progetto, ognuno con il proprio workflow.



Figura 2.1: Logo di Git

2.2 Java

Java è un linguaggio di programmazione ad alto livello orientato agli oggetti. Il suo scopo è quello di essere multiplatforma, tutte le piattaforme che supportano il linguaggio devono essere in grado di eseguire un codice *Java* compilato senza effettuare nuovamente la compilazione.

Compilando un codice *Java* si ottiene un file *Java bytecode* (con estensione `.class`) che sarà eseguito sulla *JVM* (Java Virtual Machine).

È proprio la *JVM* ad essere utilizzabile sulla maggior parte delle piattaforme.



Figura 2.2: Logo di Java

2.3 Eclipse

Eclipse è un IDE (Integrated Development Environment) per lo sviluppo software realizzato in Java.

Eclipse unisce in un'unica interfaccia grafica:

- Scrittura del codice sorgente
- Compilazione
- Debugging



Figura 2.3: Logo di Eclipse

2.4 Framework Spring

Spring è un framework open source per lo sviluppo di applicazioni in Java. *Spring* è strutturato in diversi moduli ma consente l'utilizzo solo di quelli di cui effettivamente si necessita.

Nello specifico, per la realizzazione del progetto sono stati utilizzati i moduli descritti in seguito.

2.4.1 Spring Boot

L'utilizzo di questo modulo consente di creare applicazioni Java standalone, pronte all'esecuzione.

Spring Boot consente di scegliere quale tool utilizzare per effettuare la build, in questo progetto è stato utilizzato Maven.

Alla base di ogni build con *Spring Boot* e *Maven* c'è il file *pom.xml* (acronimo di Project Object Model) in cui sono descritte tutte le impostazioni e le dipendenze necessarie alla build in un linguaggio *simil-XML*.

2.4.2 Spring Data

Spring Data fornisce un modello di programmazione per l'accesso ai dati indipendentemente dal tipo di database utilizzato.

Nello specifico, per la realizzazione del progetto è stata utilizzata la specifica di *Spring Data* chiamata *JPA* (Java Persistence API) per:

- Gestione del database
- Creazione di tabelle
- Esecuzione di query



Figura 2.4: Logo di Spring

2.5 Azure DevOps

Azure DevOps è una piattaforma fornita da Microsoft™ che consente di pianificare il lavoro, creare e distribuire applicazioni.

Nello specifico, per la realizzazione del progetto sono state utilizzate le applicazioni descritte in seguito:

- **Azure Repos** per la creazione e la gestione di repository *Git* per il controllo e il versionamento del codice sorgente.
- **Azure Pipelines** per l'automatizzazione di build e deploy dell'intero progetto.
- **Azure Artifacts** per la condivisione degli artefatti *Maven*.



Figura 2.5: Logo di Azure

2.6 Docker

Docker è un progetto open source per la creazione di container portabili e multiplatforma.

Docker utilizza il kernel Linux per isolare i processi in modo da poterli eseguire in maniera indipendente.

Ogni container è basato su un'immagine, solitamente un intero Sistema Operativo, a scelta tra quelle fornite all'interno di Docker Hub (raccolta ufficiale di tutte le immagini disponibili) o un'immagine "custom", realizzata appositamente dal singolo sviluppatore per un determinato scopo.

Grazie all'organizzazione in container si ha un alto livello di sicurezza, come se i sistemi in esecuzione fossero fisicamente separati.

Nella realizzazione del progetto è stata utilizzata l'immagine ufficiale di *Kong Gateway* (argomento approfondito nel paragrafo successivo) per la creazione del container.



Figura 2.6: Logo di Docker

2.7 Kong Gateway

Un API gateway è uno strumento che si interpone tra un client e un back end per la gestione delle API (Application Programming Interface) che si comporta come un proxy inverso accettando tutte le richieste indirizzate alle API gestite, consentendo di configurare *services* e *routes*.

Kong Gateway è un API gateway cloud-native che fornisce tutte le caratte-

ristiche descritte sopra e, inoltre, consente l'utilizzo di plugin.

Una volta installato è possibile configurarlo accedendo alle seguenti pagine:

- **Kong Manager**, porta 8000, consente di utilizzare un'interfaccia grafica per configurare *services*, *routes* e *plugins*.
- **Pagina delle configurazioni**, porta 8002, raccoglie tutte le configurazioni del gateway in formato JSON.

L'argomento sarà approfondito nel paragrafo 4.3, contestualmente al suo utilizzo nella realizzazione del progetto.



Figura 2.7: Logo di Kong Gateway

2.8 PostgreSQL

PostgreSQL è un DBMS (Database Management System) open source relazionale a oggetti che supporta la gran parte delle istruzioni del linguaggio SQL standard alle quali aggiunge diverse feature quali:

- Query complesse
- Foreign keys
- Triggers
- Views aggiornabili
- Integrità dei dati nelle transazioni
- Controllo concorrente del versionamento

Inoltre fornisce la possibilità di aggiungere tipi di dato, funzioni, operatori ecc.



Figura 2.8: Logo di PostgreSQL

2.9 Lua

Lua è un linguaggio di scripting open source che combina la sintassi procedurale a costrutti di dati basati su array associativi.

È un linguaggio tipizzato dinamicamente, viene eseguito interpretando un bytecode e gestisce la memoria in modo automatico tramite un *garbage collector*.

È stato scelto per la realizzazione del plugin per le sue caratteristiche quali:

- Velocità
- Portabilità
- Leggerezza
- Embed-easy



Figura 2.9: Logo di Lua

2.10 JWT

JWT (JSON Web Token) è un open standard che definisce un metodo sicuro per la trasmissione di informazioni tra le parti sottoforma di oggetto JSON. Le informazioni inviate sono firmate digitalmente tramite un *secret*, utilizzando l'algoritmo HMAC, oppure tramite una coppia di chiavi pubblica/privata, con gli algoritmi RSA o ECDSA. Questa tipologia di token risulta utile in diversi scenari, come:

- **Autorizzazioni**, una volta che l'utente effettua una richiesta includendo il token sarà autorizzato ad accedere a tutte le risorse che gli sono permesse.
- **Scambio di informazioni**, utilizzando i token JWT firmati con una coppia di chiavi è possibile in ogni momento verificare l'autenticità del mittente o del destinatario.

Un token JWT composto da tre parti principali, in questo ordine:

- Header
- Payload
- Signature

ed ha una forma del tipo:

xxxxx.yyyyyy.zzzzz

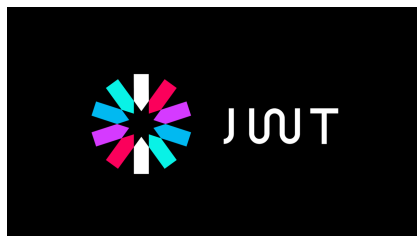


Figura 2.10: Logo di JWT

2.11 Postman

Postman è una piattaforma API per la creazione, sviluppo e testing di APIs. Nello sviluppo del progetto è stato utilizzato per la fase di testing dato che permette di effettuare delle richieste HTTP/S, offrendo la possibilità di configurare il body della stessa e di ricevere la risposta.



POSTMAN

Figura 2.11: Logo di Postman

Capitolo 3

Analisi dei requisiti

3.1 Casi d'uso

Capitolo 4

Progettazione e sviluppo

In questo capitolo saranno esposti gli approcci teorici e pratici per lo sviluppo del progetto.

4.1 Introduzione ai microservizi

Prima di introdurre il concetto di architettura a microservizi è bene introdurre il concetto di architettura monolitica.

Un'architettura monolitica è una metodologia di sviluppo secondo la quale tutti i processi coinvolti sono strettamente legati tra di loro e sono erogati come un singolo servizio.

Questa tipologia di approccio porta ad avere sistemi nei quali modificare le funzionalità diventa più complesso in quanto si deve agire sull'intero sistema e non solo sulle parti effettivamente interessate.

Inoltre, utilizzare un'architettura monolitica porta a correre dei rischi per quanto riguarda la disponibilità dell'applicazione, in quanto anche se solo uno dei processi coinvolti avesse un malfunzionamento, questo si propagherebbe nell'intera applicazione.

Per quanto riguarda le architetture a microservizi, queste sono diametralmente opposte alle architetture monolitiche.

Nelle architetture a microservizi l'obiettivo è quello di scomporre l'applicazione da realizzare nelle sue funzioni (*servizi*) di base.

Ogni servizio può essere compilato e distribuito in modo indipendente; quindi i singoli servizi possono funzionare o non funzionare senza compromettere gli altri.

Utilizzare i microservizi significa riuscire a gestire criticità inevitabili, poter sfruttare la scalabilità dinamica e semplificare l'integrazione di nuove carat-

teristiche.

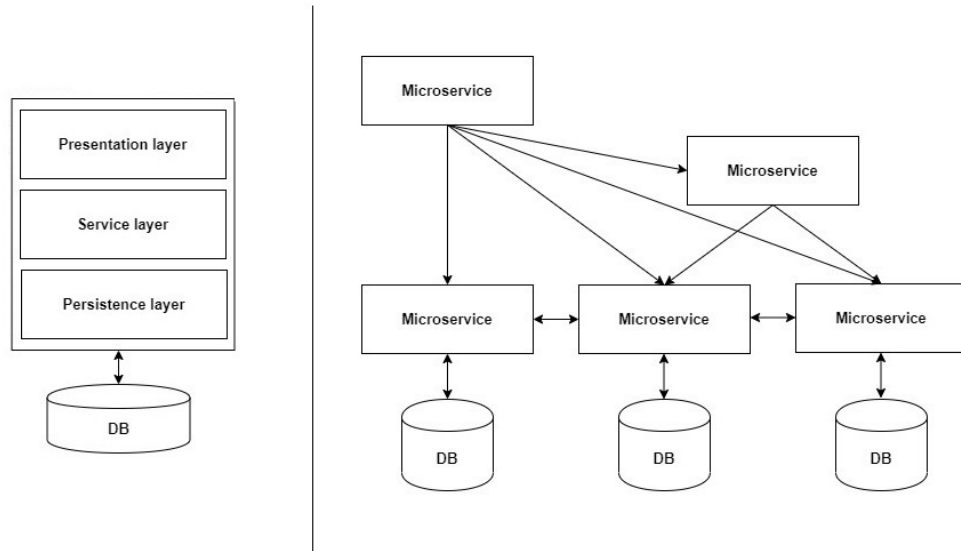


Figura 4.1: Architettura monolitica e architettura a microservizi a confronto

Oggi i container Linux permettono di eseguire più parti di un'applicazione in modo indipendente con un controllo superiore sui singoli componenti. I microservizi containerizzati rappresentano la base delle applicazioni cloud native.

Di seguito sono elencati alcuni dei vantaggi di un'architettura a microservizi:

- **Agilità**

Essendo ogni applicazione suddivisa in servizi di base, i team di sviluppo agiscono in contesti ridotti, semplificando il lavoro e riducendo i tempi del ciclo di sviluppo.

- **Scalabilità**

Lavorare con un microservizio consente di scalare in modo indipendente per rispondere alla richiesta di un determinato servizio.

In questo modo è possibile misurare il carico di lavoro di un singolo servizio e adattarlo di conseguenza; il che può portare a dei vantaggi anche in termini economici per il mantenimento dell'applicazione.

- **Semplicità di distribuzione**

I microservizi supportano l'approccio *CI/CD* (Continuous Integration/Continuous Delivery), così da semplificare l'integrazione e il testing

di nuove funzionalità avendo comunque la possibilità di effettuare un rollback in caso di problemi.

- **Codice riutilizzabile**

Uno dei vantaggi del suddividere un'applicazione in servizi è la possibilità di poter riutilizzare codici di servizi già esistenti per altre applicazioni.

- **Resilienza**

Avendo un'applicazione a servizi indipendenti si aumenta la resilienza in caso di errori.

Si possono gestire completamente gli errori di un servizio isolando la funzionalità senza bloccare l'intera applicazione.

4.2 Il microservizio CheckEmail

Il microservizio realizzato verifica l'avvenuto acquisto di un modulo applicativo da parte di un utente identificato, ai fini del microservizio, tramite un indirizzo e-mail.

Lo scopo principale del microservizio è quello di ricevere una richiesta HTTP con all'interno del body un indirizzo e-mail.

Una volta estratto l'indirizzo e-mail il microservizio effettua una *query* all'interno del Database per controllare che l'indirizzo sia presente nella tabella degli utenti abilitati all'utilizzo di un determinato servizio.

In base al risultato della query il microservizio restituisce un codice HTTP come risposta alla richiesta.

La tipologia di richieste descritte sono effettuate tramite il metodo GET di HTTP, quindi vengono richiesti dei dati dal server.

I codici che possono essere restituiti sono:

Codice HTTP	Significato	Descrizione
200	OK	Il microservizio ha risposto correttamente e l'utente è autorizzato a procedere.
402	PAYMENT REQUIRED	Il microservizio ha risposto correttamente ma l'utente non è autorizzato a procedere.
500	INTERNAL SERVER ERROR	Messaggio di errore generico, non si è potuto raggiungere il microservizio.
503	SERVICE UNAVAILABLE	Server non disponibile, non si è potuto raggiungere il microservizio.

Tabella 4.1: Codici HTTP restituiti dal microservizio



Figura 4.2: Schema di comunicazione Utente - microservizio

4.2.1 Sviluppo

Database

Per sviluppare ed utilizzare il microservizio si è resa necessaria la creazione del Database PostgreSQL per la memorizzazione dei dati degli utenti. È bene premettere che il Database è stato volutamente creato nella maniera

più semplice e facilmente manutenibile, dato che durante lo sviluppo questo doveva servire solo per scopi di testing.

Il Database è composto da due tabelle:

- **users**
Contiene le informazioni di tutti gli utenti abilitati ad accedere ad un determinato servizio.
- **email**
Contiene gli indirizzi e-mail di tutti gli utenti della piattaforma, quindi non è garantito che tutti avranno accesso a tutti i servizi.

email	
PK	<u>id integer SERIAL NOT NULL</u>
	email text NOT NULL

users	
PK	<u>id integer SERIAL NOT NULL</u>
	email text NOT NULL
	name text
	surname text

Figura 4.3: Modello Entità Relazione del Database

Microservizio

Per lo sviluppo del microservizio, come accennato in precedenza, è stato deciso di utilizzare il linguaggio di programmazione Java e il framework Spring. Il progetto è stato realizzato utilizzando l'IDE Eclipse ed è composto da cinque *package*.

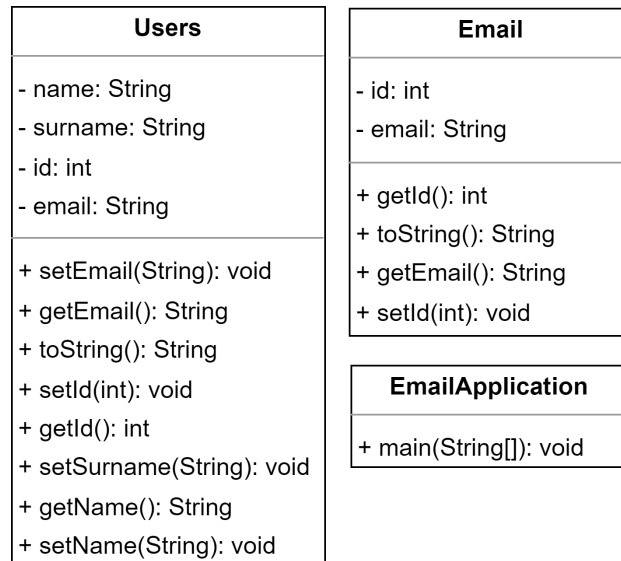
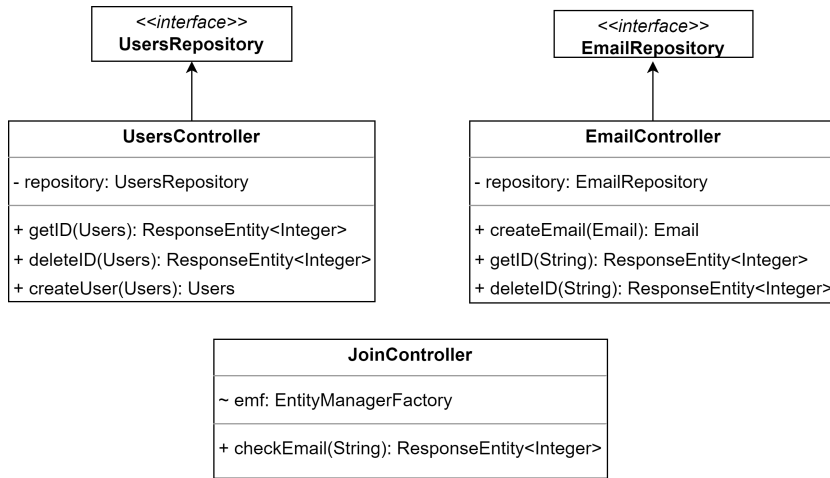


Figura 4.4: Diagramma UML delle classi principali del progetto

Descrizione dei package:

- `com.aesys.valeriodesiati.mail`
Contiene la classe principale del microservizio.
- `com.aesys.valeriodesiati.mail.model`
Contiene le classi relative alle Entità all'interno del Database.
Spring Boot JPA consente la creazione di entità in un Database a partire da una normale classe Java tramite l'aggiunta di annotazioni fornite dal framework.
Un esempio di utilizzo di un *model* è l'Algoritmo 1.
Il risultato sarà quindi la creazione di una tabella nel Database con i campi specificati dalle relative variabili annotate.
- `com.aesys.valeriodesiati.mail.repository`
Contiene le interfacce relative alle entità del Database che ereditano l'interfaccia `JpaRepository<T, ID>`.
Tale interfaccia contiene le API per tutte le operazioni CRUD di base. CRUD (Create Read Update Delete) è un acronimo che indica le quattro operazioni fondamentali per creare un'applicazione che abbia uno storage persistente.
- `com.aesys.valeriodesiati.mail.controller`
Contiene le classi *Controller* del microservizio, ovvero le classi annotate con `@RestController`.

Figura 4.5: Diagramma UML delle classi *controller* del progetto

In Spring una classe annotata come *Controller* è una classe che sarà utilizzata come handler di richieste web.

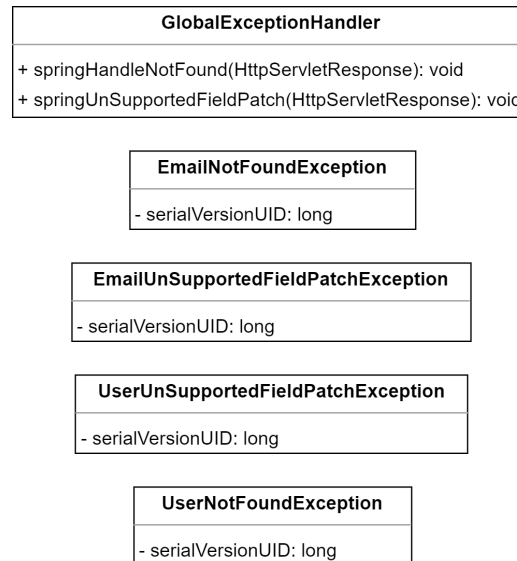
Un esempio di utilizzo di un `@RestController` è l'Algoritmo 2.

- `com.aesys.valeriodesiati.mail.exception`
 Contiene le classi per la definizione di *custom exception*.
 L'implementazione di tali classi si è resa necessaria per poter comprendere meglio gli errori in fase di sviluppo e di debug.

Algoritmo 1 Utilizzo di un *model* di Spring Boot JPA

```

1  @Entity
2  @Table (name="email")
3  public class Email {
4
5      @Id
6      @GeneratedValue(strategy=GenerationType.IDENTITY)
7      private int id;
8      @Column(name = "email")
9      private String email;
10
11     public Email() {}
12
13     public Email(int id, String email) {
14         this.id = id;
15         this.email = email;
16     }
17
18     //getters, setters and toString()...
19 }
  
```

Figura 4.6: Diagramma UML delle *custom exception* del progetto

Andando ad analizzare più a fondo le specifiche dell'Algoritmo 2 è possibile notare, in testa al metodo, l'annotazione

```
@GetMapping("/join/checkemail/{mail}")
```

tramite la quale diventa possibile intercettare delle richieste HTTP GET effettuate in un determinato percorso. Il parametro `{mail}` può essere utilizzato all'interno della funzione.

Si passa poi alla creazione di un `EntityManager` che consente di eseguire query all'interno delle *entity* create tramite i *model*.

I passaggi successivi sono per la creazione e l'esecuzione della query, che effettua un'operazione di join tra le tabelle `Users` e `Email`, andando semplicemente a controllare che l'indirizzo mail specificato nella richiesta HTTP GET sia effettivamente presente all'interno della tabella `Users` e quindi sia abilitato all'accesso.

Il tutto è svolto all'interno di un blocco `try{ } catch(Exception){ }` al fine di prevenire comportamenti anomali: se viene lanciata l'eccezione `NoResultException` o `NullPointerException`, si ritorna il codice HTTP 402 `Payment Required`. Se si termina l'esecuzione della query senza eccezioni e la variabile `result` non è impostata a `null`, si ritorna il codice HTTP 200 `OK`.

Algoritmo 2 Accesso al Database

```
1      @Transactional
2      @GetMapping("/join/checkemail/{mail}")
3      public ResponseEntity<Integer> checkEmail(@PathVariable("mail") String mail) {
4
5          EntityManager em = emf.createEntityManager();
6          em.getTransaction().begin();
7          String result = null;
8
9          try {
10             result = (String) em.createQuery("SELECT u.email
11                                           FROM Users u, Email e
12                                           WHERE u.email = e.email
13                                           AND u.email = :email")
14                                   .setParameter("email", mail)
15                                   .getSingleResult();
16         }
17         catch(NoResultException | NullPointerException e) {
18             return ResponseEntity.status(HttpStatus.PAYMENT_REQUIRED).body(402);
19         }
20
21         if(result == null)
22             return ResponseEntity.status(HttpStatus.PAYMENT_REQUIRED).body(402);
23         else
24             return ResponseEntity.status(HttpStatus.OK).body(200);
25
26     }
```

4.3 Introduzione a Kong Gateway

Come accennato nel paragrafo 2.7, *Kong Gateway* è un API gateway cloud-native che fornisce l'opportunità di configurare *services* e *routes* e, oltre a questi, anche *plugin* e *consumer*.

4.3.1 Service

Un *service* in Kong Gateway è un'astrazione di tutti i servizi upstream custom che si aggiungono alla configurazione. Con *servizio upstream custom* si intende un microservizio custom che prende dati dalla richiesta inoltrata al gateway e ne restituisce altri al gateway stesso, che si occuperà di comunicarli al client.

Solitamente ad ogni *service* è associata una o più *routes*.

4.3.2 Route

Una *route* è una regola definita per indirizzare correttamente le richieste del client.

L'associazione di una (o più) route ad un servizio consente di realizzare un meccanismo di routing molto potente, dato che è possibile configurare nel dettaglio il percorso che si vuole realizzare (protocolli da utilizzare, livello di sicurezza ecc.).

4.3.3 Consumer

Un *consumer* in Kong Gateway può essere inteso come un utente di uno specifico servizio e può essere identificato tramite un `id` univoco.

4.3.4 Plugin

Un *plugin* è un'entità che sarà eseguita durante tutto il ciclo di vita di una richiesta o risposta HTTP/S (HyperText Trasfer Protocol / Secure).

È il modo in cui Kong Gateway fornisce la possibilità di ottenere funzionalità aggiuntive per un *service* o una *route*.

I plugin configurabili possono essere sia proprietari (attivabili da Kong Manager) sia custom. Per la realizzazione di un plugin custom si ha la possibilità di scegliere tra vari linguaggi di programmazione per lo sviluppo quali Go, Python, JavaScript e Lua (linguaggio utilizzato per lo sviluppo del plugin custom utilizzato nel progetto).

4.4 Architettura del plugin checkemail

Un plugin in Lua per Kong Gateway si compone principalmente di due file:

- `handler.lua`

Questo file contiene tutta la logica del plugin, devono essere implementate tutte le funzioni coinvolte nel ciclo richiesta/risposta.

- `schema.lua`

Racchiude tutte le configurazioni aggiuntive, se necessarie, come ad esempio coppie chiave/valore o altre impostazioni per modificare il comportamento del plugin.

Il funzionamento e l'utilizzo di questi file sarà approfondito nel paragrafo successivo e nel paragrafo 4.5.

4.4.1 Sviluppo

Come detto in precedenza la logica del plugin è interamente contenuta nel file `handler.lua`.

È richiesto il seguente comportamento dal plugin:

1. Analizzare il token ricevuto.
2. Parse del token.
3. Inviare una richiesta HTTP al microservizio `CheckEmail`.
4. Ottenere e inoltrare al Gateway il codice di risposta ottenuto.

Il primo punto è realizzato dalla funzione `SplitToken(token)`, che analizza il token JWT ricevuto e lo suddivide nelle tre parti di cui è composto: Header, Body e Signature.

La funzione restituisce un array contenente le tre componenti del token.

Algoritmo 3 Suddivisione token JWT

```
1      local function SplitToken(token)
2          local segments = {}
3          for str in string.gmatch(token, "([^\.\.]+)") do
4              table.insert(segments, str)
5          end
6          return segments
7      end
```

Successivamente si procede con il parse del token, mediante la funzione `ParseToken(token)` che inizia chiamando la funzione `SplitToken(token)` per ottenere l'array delle parti, per poi proseguire applicando una decodifica ad ogni parte controllando anche l'eventuale presenza di errori.

La funzione può restituire tre o quattro risultati: nel caso in cui non ci siano stati errori vengono restituite solo le tre componenti del token decodificate, altrimenti vengono restituiti quattro risultati, i primi tre impostati a `null` (rappresentano i campi del token) e il quarto come stringa con un messaggio di errore.

Da notare come tutte le parti vengano prima suddivise e poi decodificate anche se il dato ricercato (l'indirizzo mail) si trova solo nel Body, questo per favorire e semplificare la ricerca di eventuali componenti del token corrotte.

Algoritmo 4 Parse token JWT

```
1      local function ParseToken(token)
2          local segments = SplitToken(token)
3          if #segments ~= 3 then
4              return nil, nil, nil, "Invalid token"
5          end
6
7          local header, err = cJSON_safe.decode(base64url_decode(segments[1]))
8          if err then
9              return nil, nil, nil, "Invalid header"
10         end
11
12         local body, err = cJSON_safe.decode(base64url_decode(segments[2]))
13         if err then
14             return nil, nil, nil, "Invalid body"
15         end
16
17         local sig, err = base64url_decode(segments[3])
18         if err then
19             return nil, nil, nil, "Invalid signature"
20         end
21
22         return header, body, sig
23     end
```

Il ciclo del funzionamento del plugin termina quindi l'estrazione dell'indirizzo mail dal token decodificato e l'inoltro della richiesta HTTP al micro-servizio **CheckEmail**.

Il microservizio è raggiungibile al link

<http://restservice-springid.azurewebsites.net/join/checkemail/>,
unendo al link l'indirizzo mail che si desidera controllare.

Algoritmo 5 Inoltro richiesta HTTP dal plugin

```
1      local body, code, headers, status =
2          http.request("http://restservice-springid.azurewebsites.net
3                      /join/checkemail/"..bodyTok.email)
```

L'ultimo controllo che si effettua è quello sul codice di ritorno HTTP ricevuto dal microservizio, quindi si invia una risposta da Kong Gateway all'utente.

Algoritmo 6 Inoltro risposta dal plugin a Kong Gateway

```
1   if code == 200 then
2       return kong.response.exit(200, "Success")
3   end
4
5   if code == 402 then
6       return kong.response.error(402, "Payment Required")
7   end
8
9       --response codes 500 and 503...
```

4.5 Configurazione Kong Gateway

Kong Gateway è stato containerizzato tramite Docker utilizzando l'immagine ufficiale presente su Docker Hub.

Il container è stato reso raggiungibile tramite una macchina virtuale su Azure con immagine Debian.

Sono state create delle pipeline CI/CD per automatizzare i processi di containerizzazione e di installazione del plugin.

La pipeline è azionata automaticamente dai cambiamenti nella repository principale del progetto e si occupa di far eseguire, all'interno della macchina virtuale, uno script per l'aggiornamento dei *services*, *routes*, *consumers*.

Il gateway può essere configurato in modi diversi: Interfaccia grafica di Kong Manager, disponibile collegandosi da browser alla porta 8002 del container. Tramite file JSON contenenti le informazioni necessarie che saranno aggiunte al file di configurazione principale (`kong.conf`), inoltrati con delle richieste HTTP POST verso il container in esecuzione.

Come detto in precedenza, gli aspetti configurabili sono:

- *Services*
- *Routes*
- *Consumers*
- *Plugin*

Ognuno con un file JSON di configurazione dedicato.

I plugin custom, come quello realizzato, non necessitano di un file di configurazione dedicato, è sufficiente copiare i file relativi nella directory `/usr/local/share/lua/5.1/kong/plugins/nome_plugin/` del container.

Tutta la configurazione è effettuata tramite lo script bash `addplugin` che ha i seguenti compiti:

- Avviare il container.
- Copiare all'interno i file relativi ai plugin custom.
- Riavviare il container (per consentire la lettura dei file dei plugin caricati)
- Eseguire il comando `curl` per tutti i file JSON presenti relativi a tutte le altre configurazioni necessarie.

Di seguito alcune righe dello script di configurazione:

Algoritmo 7 Script per la configurazione di Kong Gateway

```
1      sudo docker exec -it --user root $container rm -rf
2                                     /usr/local/share/lua/5.1/kong/plugins/$dir
3      sudo docker exec -it --user root $container mkdir
4                                     /usr/local/share/lua/5.1/kong/plugins/$dir
5      sudo docker cp . $container:/usr/local/share/lua/5.1/kong/plugins/$dir/
6      curl -s -X POST -H "Content-Type: application/json"
7          -d @./config/checkemail/services.json
8          http://checkemail.westeurope.cloudapp.azure.com:8001/services > /dev/null
```

Capitolo 5

Testing

”

ۛۛۛ

Conclusione

Bibliografia

- [aaaaaaaaaaaaaaaaaaaaaaaaaaaa, 2010] aaaaaaaaaaaaaaaaaaaaaaaaaaaaa (2010). *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education.
- [Allen and Zilberstein, 2009] Allen, M. and Zilberstein, S. (2009). Complexity of decentralized control: Special cases. In *23rd Annual Conference on Neural Information Processing Systems 2009, 7-10 December, Vancouver, British Columbia, Canada*, pages 19–27. Curran Associates, Inc.
- [Bernstein et al., 2002] Bernstein, D. S., Givan, R., Immerman, N., and Zilberstein, S. (2002). The complexity of decentralized control of markov decision processes. *Mathematics of operations research*, 27(4):819–840.
- [De Weerd et al., 2003] De Weerd, M., Bos, A., Tonino, H., and Witteveen, C. (2003). A resource logic for multi-agent plan merging. *Annals of Mathematics and Artificial Intelligence*, 37(1-2):93–130.
- [Durfee, 2001] Durfee, E. H. (2001). *Distributed Problem Solving and Planning*, pages 118–149. Springer Berlin Heidelberg, Berlin, Heidelberg.
- [Gelfond and Lifschitz, 1998] Gelfond, M. and Lifschitz, V. (1998). Action languages. *Electron. Trans. Artif. Intell.*, 2:193–210.
- [van Beek, 2019] van Beek, H. (2019). Digital Forensics as a Service. <https://www.forensicinstitute.nl/research-and-innovation/latest-forensic-innovations/hansken>.

Appendice A

Appendice

Il codice sorgente dell'intero progetto è disponibile al link:
<https://github.com/valeriodesiati/RepoTesi>