UNIVERSITÀ DEGLI STUDI
DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche
e Matematiche

Corso di Laurea in Informatica

Progetto e sviluppo di un SDK Python per la gestione di microservizi in ambito aziendale

Daniele Costoli

Tesi di Laurea

Relatore:
Prof. Riccardo Martoglia

Anno Accademico 2021/2022

RINGRAZIAMENTI

Ringrazio il prof. Riccardo Martoglia per la propria disponibilità e l'aiuto datomi durante la scrittura di questa tesi.
Un ringraziamento speciale va alla mia famiglia ed ai miei amici che in questi anni mi hanno sempre sostenuto e incoraggiato.

PAROLE CHIAVE

Python

Flask

Business Logic

Configurazione

JSON

Indice

INTRODUZIONE 1

I - IL CASO DI STUDIO 3

1 LO SCOPO DEL PROGETTO 4

1.1 INTRODUZIONE 4
1.2 BUSINESS LOGIC 5
1.3 CONFIGURAZIONE 6
1.4 MANIFEST 10
1.5 STARTUP AND SHUTDOWN 14
1.6 HEALTH CHECKS 15
1.7 LOGGING 17
1.8 RIASSUMENDO 18

2 TECNOLOGIE UTILIZZATE 19

2.1 INTRODUZIONE 19
2.2 FRAMEWORK FLASK 20
2.3 HEALTHCHECK 21
2.4 LOGGING 23
2.5 DYNACONF 26

II - PROGETTO E SVILUPPO 29

3 PROGETTAZIONE 30

3.1 INTRODUZIONE 30
3.2 BUSINESS LOGIC 30
3.2.1 Gestione e validazione dei dati in ingresso e in uscita 30
3.2.2 Lavorazione dei dati in ingresso per generare quelli in uscita (process) 31

3.3 CONFIGURAZIONE 33
3.3.1 Configurazione con file di serializzazione (YAML) *34*
3.3.2 Configurazione con variabili di ambiente *34*
3.4 MANIFEST 34
3.5 STARTUP AND SHUTDOWN 35
3.6 HEALTH CHECKS 35
3.7 LOGGING 36
4 IMPLEMENTAZIONE 38
4.1 INTRODUZIONE 38
4.2 BUSINESS LOGIC 39
4.2.1 Gestione e validazione dei dati in ingresso e in uscita *39*
4.2.2 Lavorazione dei dati in ingresso per generare quelli in uscita (process) *41*
4.3 CONFIGURAZIONE 42
4.4 MANIFEST 43
4.5 STARTUP AND SHUTDOWN 45
4.6 HEALTH CHECKS 46
4.7 LOGGING 46
4.8 FRAMEWORK FLASK 48
CONCLUSIONI 50
BIBLIOGRAFIA 52

Elenco delle figure

3.1 Schema per la gestione della Business Logic................................pag. 32
4.1 Gestione struttura dati ingresso da codice....................................pag. 38
4.2 Gestione struttura dati ingresso da file..pag. 39
4.3 Gestione struttura dati ingresso da file..pag. 39
4.4 Funzione handler della classe Logic..pag. 41
4.5 Configurazione da file yalm...pag. 41
4.6 Configurazione con variabili di ambiente....................................pag. 42
4.7 Inizializzazione campi manifest...pag. 42
4.8 Stampa manifest..pag. 43
4.9 Stampa manifest da terminale..pag. 44
4.10 Inserimento funzioni di controllo...pag. 45
4.11 Dizionario dati di log..pag. 46
4.12 Funzione per generare i log...pag. 46
4.13 Utilizzo di Flask..pag. 48

Introduzione

Questo progetto è stato svolto in collaborazione con l'azienda Expert.ai (ex Expert System), una software house italiana fondata a Modena nel 1989. Inizialmente nata in un garage da un'idea di un paio di ragazzi universitari, oggi è un'azienda internazionale quotata in Borsa e collabora con alcune delle più importanti organizzazioni e agenzie governative in Europa, Medio Oriente, Nord e Sud America. La società è specializzata nell'analisi e nella comprensione del linguaggio naturale allo scopo di estrarre informazioni utili dalle più diverse fonti di dati, tutto ciò mediante semplici ma potenti strumenti di intelligenza artificiale. In particolare, utilizza un approccio ibrido che combina machine learning e semantica, offrendo il modo più efficace e pragmatico per gestire informazioni non strutturate anche nei casi più complessi, mettendole a disposizione dei suoi clienti.

Lo scopo del progetto è di realizzare un SDK Python in grado di creare facilmente microservizi da integrare nella sezione NLFlow della piattaforma aziendale Expert.ai, attualmente è già presente un SDK con questo scopo ma risulta essere antiquato e utilizzabile solo da coloro che intendono sviluppare utilizzando Java (essendo questo SDK scritto utilizzando questo linguaggio). L'obbiettivo ultimo è quindi quello di realizzare un SDK Python che prenda spunto dalla versione Java, ottimizzando e migliorandone diversi aspetti.
Per ottenere questo risultato non si è partiti da zero, ma si è utilizzata una versione primordiale e limitata dell'SDK, che da questo momento in poi sarà chiamata 'SDK Python base', lo scopo è quello di migliorare alcune funzionalità e inserirne di nuove.

La tesi è suddivisa in 4 capitoli, nei quali verranno affrontate tutte le fasi che hanno contribuito alla realizzazione di questo progetto.
Nel primo verranno discussi nel dettaglio gli elementi caratterizzanti del progetto e i suoi obiettivi principali.
Nel secondo capitolo verranno spiegate le tecnologie utili per lo sviluppo dello stesso.
Nel terzo verranno analizzate le scelte relative alla progettazione delle sue funzionalità, nonché i relativi problemi affrontati e le soluzioni trovate per risolverli.
Nel quarto ed ultimo capitolo verranno riassunte le principali implementazioni effettuate per raggiungere gli obbiettivi previsti utilizzando le idee discusse durante la fase di progettazione.

Parte I

Il caso di studio

Capitolo 1
Lo scopo del progetto

1.1 Introduzione

In questo capitolo verranno analizzate gli aspetti principali del progetto, nonché le varie funzionalità che dovranno essere implementate. Per fare questo verranno analizzati 2 progetti, i quali saranno utili durante le fasi successive per avere un'idea più chiara degli obbiettivi da raggiungere e delle difficoltà che si sono incontrare durante il percorso.

Il primo è un SDK Java realizzato dall'azienda avente le stesse funzionalità che si vogliono implementare all'interno di questo progetto.
Il secondo è una versione limitata e primordile dell'SDK Python che si vuole realizzare, il quale verrà utilizzato come progetto di partenza con lo scopo di migliorarne alcune funzionalità e inserirne di nuove per ottenere il risultato pianificato.

In ognuna delle seguenti sezioni verrà analizzata una singola funzionalità, andando a spiegarne: lo scopo all'interno del progetto, come è stata implementata all'interno dell'SDK Java e se sia già stata implementata (anche solo in parte) nella versione dell'SDK Python base.

1.2 Business Logic

Uno dei punti più importanti di questo SDK è quello di fornire la possibilità all'utente finale di definire la business logic relativa al proprio servizio, ovvero tutta quella logica applicativa che rende operativa un'applicazione. In questo caso specifico la logica applicativa che è necessario fornire è riassumibile nei seguenti 3 punti:

- Gestione e validazione dei dati in ingresso
- Lavorazione dei dati in ingresso per generare quelli in uscita (process)
- Gestione e validazione dei dati in uscita

SDK Java

Nell'SDK Java questo aspetto è gestito nel modulo 'logic', all'interno del quale vengono gestiti i diversi aspetti. L'interfaccia dà la possibilità di passare POJO di input e output, che sono i tipi generici forniti all'interfaccia. I POJO vengono poi validati tramite uno schema Json che può essere generato dall'SDK stesso oppure può essere fornito come risorsa dal progetto. È anche possibile estendere la classe 'DefaultValidator' in modo da personalizzare ulteriormente la validazione dei dati. Il risultato delle validazioni è un codice di stato http, con le seguenti caratteristiche:

- 200: convalida con esito positivo, nessun problema;
- 400 e un messaggio di errore: convalida fallita perché il POJO scelto dall'utente non rispetta i vincoli e le condizioni necessarie;
- 500 e un messaggio di errore: convalida fallita per errori interni del server.

È possibile personalizzare il modo in cui i POJO vengono serializzati/deserializzati in JSON, per fare questo è necessario operare sulla configurazione utilizzando le funzionalità fornite dal framework Micronaut.

Per la gestione del processo, ovvero per generare i dati di output partendo da quelli di input, è necessario implementare il metodo 'AnalysisResponse' all'interno di una propria classe che implementi l'interfaccia 'NLFlowLogic'.

SDK Python base

I dati di input e output sono considerati come due semplici dizionari Python e vengono validati utilizzando degli schemi Json che devono essere inseriti come risorse al progetto.

Per la gestione del processo è necessario implementare la funzione 'process' che ha caratteristiche simile a quella Java, in quanto come campo in ingresso avrà il dizionario con i dati di input e dovrà ritornare quello con i dati di output. Questa stessa funzione verrà poi passata come parametro alla funzione 'process' della classe 'Logic'.

Per testare il corretto funzionamento di questo e altri servizi l'SDK Python base utilizza il framework Flask per creare un server integrato molto semplice, in modo che gli utenti possano accedere facilmente a queste funzionalità.

In particolare per il testing di questo servizio è necessario mandare in esecuzione l'SDK e utilizzare uno strumento di testing delle API (come Postman) per testare l'url '127.0.0.0:18080/api/v1/nlflow', usando il metodo POST. Inoltre bisognerebbe inserire nella chiamata i dati di input (coerenti con lo schema JSON fornito) e ricevere quindi un output adeguato alla logica configurata; attualmente in questa versione dell'SDK l'output è sempre 200 di default e per far funzionare tutto correttamente sono necessarie ulteriori implementazioni.

1.3 Configurazione

L'SDK deve fornire la possibilità all'utente di definire alcune configurazioni relative al proprio servizio, che verranno utilizzate durante il metodo di process.

Per ottenere questo risultato deve essere possibile fare uso di:

- file di serializzazione (YAML), che dovranno essere importati come risorsa al progetto;
- variabili di ambiente.

SDK Java

Per gestire le configurazioni è possibile definire una classe che deve essere annotata con @NLFlowConfig, la quale verrà esposta nel contesto dell'applicazione e può essere iniettata nel proprio bean logico. È possibile definire i campi usando i tipi java simple o anche le mappe se vi è la necessità di utilizzare chiavi dinamiche.

È possibile dare valore al proprio bean di configurazione utilizzando il file ‘application.yml’, il quale deve essere inserito all’interno dei un’apposita cartella. È possibile utilizzare le variabili di ambiente, ad esempio, sostituendo ogni punto o trattino con un trattino basso, tutto maiuscolo.

La radice per la propria configurazione personalizzata, se non specificata, è custom, quindi se si vuole assegnare un valore alla proprietà 'key', è possibile usare semplicemente la variabile d’ambiente CUSTOM_KEY. Si può scegliere il metodo che si preferisce, l'effetto è che il proprio bean di configurazione verrà popolato con i valori forniti. È possibile usare la notazione camelCase nel tuo bean java e kebab-case in yaml indipendentemente dalla conversione, questo viene fatto dal framework per impostazione predefinita.

Si può specificare @NLFlowConfig su più classi se si preferisce dividere logicamente la propria configurazione personalizzata, l'importante è che ogni chiave di configurazione sia raggiungibile dal giusto percorso, dato dalla combinazione dall’attributo @NLFlowConfig ("custom" se non specificato) e nomi di campo.

Nell'esempio sopra, "nestedKey" è raggiungibile dai seguenti percorsi di configurazione:

- custom.nested.nestedKey (file yaml)
- custom.nested.nested-key (file yaml)
- CUSTOM_NESTED_NESTED_KEY (variabile d'ambiente)

Oltre alla configurazione personalizzata, ci sono altre configurazioni di servizio che devono essere impostate affinché il servizio funzioni completamente. Un esempio completo di un 'application.yml' file contenente tutte le configurazioni del servizio è il seguente:

```yaml
service:
  id:
  name:
  version:
  description: ""
  author: ""
  category: "custom"
  image: ""
  liveness-probe:
    initial-delay-seconds: 0
    period-seconds: 10
    timeout-seconds: 1
    success-threshold: 1
    failure-threshold: 3
  readiness-probe:
    initial-delay-seconds: 0
    period-seconds: 10
    timeout-seconds: 1
    success-threshold: 1
    failure-threshold: 3
  startup-probe:
    initial-delay-seconds: 0
    period-seconds: 10
    timeout-seconds: 1
    success-threshold: 1
```

```yaml
      failure-threshold: 3
    default-timeout: 3s
    default-replicas: 1
    default-memory: 512Mi
    default-cpu: 500m
    input:
    output:
    schema-version: 2019-09

custom:
    #your custom properties, following example above:
    key: value
    dynamic-map:
      dynamic-key: dynamic-value
    nested:
      nested-key: nested-value
```

I campi id, name e version sono campi obbligatori, gli altri sono facoltativi.
I valori nell'esempio sopra per i campi facoltativi sono quelli predefiniti. Tutti i campi nella sezione del servizio sono utili per guidare la generazione di manifest.

Ruolo dei campi:

- id, name, version, description, author sono usati per descrivere il servizio;
- category è la sezione dell'interfaccia utente in cui il tuo servizio verrà inserito come manifest;
- image è il percorso completo in cui risiederà l'immagine della finestra mobile;
- liveness-probe, readiness-probe, startup-probe sono usati per guidare Kubernetes sui tempi delle sonde;
- default-timeout, e default-replicas vengono utilizzati per guidare l'utilizzo delle risorse predefinite del servizio nel cluster Kubernetes.default-memorydefault-cpu;

- input e output vengono utilizzati se si desidera fornire il proprio schema Json per POJO di input e output ed evitare di generarli utilizzando l'SDK. Il valore è il nome del file di quegli schemi nel percorso classe (ad esempio nella cartella delle risorse). Attenzione, se si forniscono gli schemi Json, questi devono essere definiti utilizzando bozze specifiche per essere convalidati correttamente. Sono ammesse le seguenti versioni: draft-04, draft-06, draft-07, 2019-09;
- schema-version viene utilizzato se si desidera specificare la versione con cui verranno generati gli schemi Json, se si sceglie di consentire a SDK di generarli. I valori possibili sono draft-06, draft-07, 2019-09.

È anche possibile modificare la porta di servizio, se necessario. Per fare ciò, è sufficiente sovrascrivere questo campo (8080 è il valore predefinito):

```yaml
micronaut:
  server:
    port: 8080
```

SDK Python base

È possibile inserire il file ‘config.yml’, il quale deve essere presente all’interno della directory ‘/src/resources/’. La struttura del file che ci si aspetta è equivalente a quella del file ‘application.yml’ della versione Java.

Nessuna gestione delle variabili di ambiente, questo aspetto dovrà essere implementato nella versione finale.

1.4 Manifest

L’SDK deve dare la possibilità di generare un manifest, ovvero un documento XML in grado di descrivere il contenuto dell’applicazione. Le informazioni utili

alla creazione di questo file dovranno essere ricavate da un insieme di configurazioni di servizio impostate e schemi rappresentanti la struttura dei dati in input e in output.

SDK Java

L'SDK offre la possibilità di generare manifest che verrà realizzato utilizzando un mix di configurazioni di servizio e i dati degli schemi Json di input/output, che possono essere forniti manualmente o generati con la funzione appropriata. Per ottenere il manifest è sufficiente eseguire l'applicazione passando generateManifest come argomento, ad esempio:

```
./gradlew run --args="generateManifest"
```

È possibile anche generare manifest dalla propria immagine docker costruita passando lo stesso argomento come argomento di docker run.

```
docker run -it nlflow-application generateManifest
```

Per ottenere un manifest completo si ha bisogno di un paio di altre cose:

- il riferimento all'immagine: si può semplicemente aggiungere informazioni sull'immagine a manifest sovrascrivendole utilizzando variabili di ambiente, come si fa con qualsiasi altra configurazione personalizzata. La variabile di ambiente da valutare è SERVICE_IMAGE. Se si vuole semplicemente eseguire l'immagine docker è sufficiente utilizzare l'istruzione:

```
docker run -it -e SERVICE_IMAGE=925204499437.dkr.ecr.us-
east-2.amazonaws.com/essentia-nlflow-cogito:0.3.2 nlflow-
application generateManifest
```

- envVars, parameters_descriptor, confMappings: se ne ha bisogno per veicolare le proprietà dall'interfaccia utente ai valori richiesti dal proprio servizio. Per aggiungerli al manifest, si può semplicemente creare un file nel percorso di classe (ad esempio sotto src/main/resources) chiamato manifest-additions.json contenente quelle proprietà. Saranno aggiunti al manifest come parametri funzionali. Ecco alcuni dettagli di ogni sezione:
    - **envVars** sono quelli usati dalla propria applicazione, i valori che si prevede vengano popolati per il funzionamento del servizio. Può essere semplicemente una coppia di nome (chiave della variabile di ambiente) e valore, oppure si può anche specificare una configMapproprietà che identifica il nome di una Configmap nel cluster Kubernetes che possiede i valori. In questo caso, il valore corrisponde alla chiave della configmap;
    - **parameters_descriptor** sono proprietà esposte all'interfaccia utente. Questa sezione prevede più proprietà a seconda del tipo di parametro:
        - name: nome del parametro (Stringa)
        - description: descrizione dettagliata del parametro (Stringa)
        - type: tipo del parametro (Stringa). Può essere:
            - enum: per i valori dell'elenco
            - int: per valori interi
            - boolean: per valori booleani
            - string: per valori stringa
            - map: per valori oggetto
            - code_js: per i valori javascript
            - json: per valori json
            - model: per i valori del modello pubblicati
            - time_measure: per valori di timeout

        - memory_measure: per i valori di utilizzo della memoria
        - cpu_measure: per i valori di utilizzo della CPU
    - default: il valore predefinito del parametro.
    - mandatory: se la proprietà deve essere popolata (Boolean)
    - editable: se la proprietà è modificabile (Boolean)
    - choices: elenco di possibili valori se il tipo è 'enum'
    - aggregation: tipo di aggregazione se il tipo è 'enum'. I valori possibili sono:
        - set: per pulsante di opzione
        - list: per le caselle di controllo
        - dropdown: per i menu a discesa
    - minval: utilizzato solo se il tipo è 'int'. Questo è il valore minimo consentito, utilizzato per la convalida (Intero)
    - maxval: utilizzato solo se il tipo è 'int'. Questo è il valore massimo consentito, utilizzato per la convalida (Intero)
    - regex: utilizzato solo se il tipo è 'string'. Questa è un'espressione regolare utilizzata per convalidare il valore (Stringa)
    - modelType: utilizzato solo se il tipo è 'modello'. È il tipo di modello di piattaforma a cui è limitato questo esagono (Stringa)
- **confMappings** è l'associazione tra envVars e parameters_descriptor. È composto da due proprietà:
    - valueScript, necessario per estrarre il valore da parameters_descriptor. C'è una struttura di dati incorporata config, che contiene i valori dei parametri e a cui è possibile accedere con '..'
    - jsonPath, scritto utilizzando la sintassi jsonPath . Questo è necessario per abbinare il valore estratto dalla proprietà precedente con la definizione envVars. Con jsonPath è

possibile specificare la sezione del manifest in cui verrà inserito il valore.

SDK Python base

Genera un manifest parziale, con solo i seguenti campi: version, id e autore.
Non è possibile richiedere la generazione del manifest da linea di comando, per visualizzare il manifest è necessario avviare l'SDK e inserire nella barra di ricerca di un browser '127.0.0.0:18080/api/v1/manifest'. Questo visualizzerà nel body della pagina la stampa parziale del manifest riferita all'applicazione. Per ottenere un manifest completo (equivalente a quello presente nella versione Java) sono necessarie ulteriori implementazioni.

1.5 Startup and shutdown

L'SDK deve fornire la possibilità all'utente di inserire funzioni o procedure che verranno eseguite automaticamente durante la fase di avvio o/e durante la fase di terminazione dell'applicativo.

SDK Java

Per sfruttare questa possibilità l'SDK permette di creare una propria classe che deve estende da NLFlowEventListener, all'interno della quale si può effettuare l'override dei metodi onStartup() e onShutdown(), nei quali si possono inserire tutte le funzionalità che si vogliono eseguire in automatico rispettivamente nella fase di avvio e in quella di terminazione dell'applicativo.

Esempio:

```
public class ServerEvents extends NLFlowEventListener {

    @Override
    public void onStartup() {
        log.info("Doing something on startup");
```

```
    }

    @Override
    public void onShutdown() {
        log.info("Doing something on shutdown");
    }
}
```

SDK Python base

Non sono presenti funzionalità a supporto di questo servizio, dovranno essere implementate interamente.

1.6 Health checks

Poiché il servizio verrà esposto come microservizio nel cluster Kubernetes, sono necessari controlli di integrità per garantire che il servizio sia attivo e pronto. Utilizzando l'SDK si deve avere un controllo dello stato integrato che verifichi che il server sia correttamente in esecuzione. Inoltre deve essere possibile l'inserimento di ulteriori funzioni di controllo personalizzate, questo per analizzare aspetti specifici.

SDK Java

Se lo si desidera, è possibile aggiungere più controlli di liveness o readiness semplicemente creando 2 classi che estendo rispettivamente da NLFlowLivenessIndicator e NLFlowReadinessIndicator, inserendo l'annotazione @Livenessoa nella prima e @Readinessseconda nella seconda.
L'health checks risultante sarà l'intersezione di tutti i controlli definiti, sia per la liveness che per la readiness. Tali controlli saranno esposti sotto gli endpoint /health/liveness, /health/readinessendpoint e un cumulativo /health che è la combinazione di due.
Per esempio:

```
@Liveness
public class LivenessCheck extends NLFlowLivenessIndicator {

    @Override
    protected NLFlowHealthResult getHealthInformation() {
        return NLFlowHealthResult.builder()
                .withStatus(NLFlowHealthResult.Status.UP)
                .withDetails(Map.of("key", "liveness"))
                .build();
    }

    @Override
    protected String getName() {
        return "TestLiveness";
    }

    @Override
    public int getOrder() {
        return super.getOrder();
    }
}

@Readiness
public class ReadinessCheck extends NLFlowReadinessIndicator
{

    @Override
    protected NLFlowHealthResult getHealthInformation() {
        return NLFlowHealthResult.builder()
                .withStatus(NLFlowHealthResult.Status.UP)
                .withDetails(Map.of("key", "readiness"))
                .build();
    }

    @Override
    protected String getName() {
        return "TestReadiness";
```

```
    }

    @Override
    public int getOrder() {
        return super.getOrder();
    }
}
```

La funzione getHealthInformation dà la possibilità di fornire alcuni dettagli sul' health checks. Se non è necessario superare il controllo, lancia semplicemente un'eccezione di runtime.
Con getName() si può dare un nome al proprio health checks.
Con getOrder() è possibile specificare l'ordine di esecuzione relativi al controllo dello stato, con valori più bassi come priorità. È possibile utilizzare Ordered.LOWEST_PRECEDENCE o Ordered.HIGHEST_PRECEDENCE per definire la priorità massima e minima per il controllo dello stato. 0 è l'impostazione predefinita.

SDK Python base

Non sono presenti funzionalità a supporto di questo servizio, dovranno essere implementate interamente.

1.7 Logging

Nell'SDK deve essere presente una gestione dei log, in modo da catalogare e tenere traccia delle modifiche o degli errori rivelati durante l'esecuzione.

SDK Java

SDK fornisce un file di logback da includere in logback.xml all'interno del proprio progetto se si vuole accedere a Json con un formato ben noto. L'ideale è usare la versione SDK quando crei un'applicazione per la distribuzione di produzione.

Un esempio di file logback.xml che dovrebbe entrare in un'applicazione containerizzata è il seguente:

```
<configuration>
    <include resource="logback-ls.xml"/>
</configuration>
```

Questa inclusione di logback fornisce informazioni standard utili per il debug, come l'ID di correlazione per la traccia, eventuali dati extra inseriti dall'applicazione, informazioni su thread e classi e uno stacktrace per i log degli errori. L'SDK usa un filtro per intercettare la chiamata HTTP della logic e recuperare l'ID correlazione dall'intestazione, se esiste, o generarne uno nuovo in caso contrario. Si può sempre recuperare questo Id di correlazione tramite MDC. Viene adottato automaticamente un registro degli accessi della chiamata con indicazione della latenza.

SDK Python base

Non sono presenti funzionalità a supporto di questo servizio, dovranno essere implementate interamente.

1.8 Riassumendo

Nelle sezioni precedenti sono state descritte tutte le funzionalità che devono essere presenti nell'SDK per ottenere un risultato ottimale richiesto dall'azienda.

L'idea è quella di partire dalla versione dell'SDK Python base per ottenere risultati quanto più possibili equivalenti alla versione Java, sfruttando le tutte funzionalità possibili offerte dal linguaggio Python. Nei prossimi capitoli verranno analizzati nel dettaglio tutti gli aspetti riguardanti le tecnologie utilizzate, le scelte progettuali e le relative implementazioni fatte, necessarie per il raggiungimento di tale obbiettivo.

Capitolo 2
Tecnologie utilizzate

2.1 Introduzione

Python è un linguaggio di programmazione dinamico, ideato negli anni Novanta si è diffuso rapidamente all'interno della community dei programmatori diventando oggi uno dei linguaggi di programmazione più utilizzati e che trova una sua applicazione in svariati campi. Questo linguaggio è caratterizzato da una sintassi semplice che comporta un codice altamente comprensibile. Inoltre è un linguaggio di alto livello, facilmente versatile, adatto alla programmazione orientata agli oggetti, alla programmazione strutturale e a quella funzionale.

Si riportano di seguito alcune librerie che sono state essenziali per la realizzazione di questo progetto.

2.2 Framework Flask

Flask [1] è un framework Python utilizzato per lo sviluppo Web, il quale riprende l'idea di praticità del linguaggio Python riuscendo a mettere a disposizione varie funzionalità utilizzando una sintassi semplice ma potente, rendendo immediato l'avvio e flessibile l'utilizzo. Al contempo offre un ampio livello di personalizzazione sotto diversi aspetti, tra cui l'integrazione del database, gli account e l'autenticazione.

Installazione:

```
pip install flask
```

Esempio:

il codice che segue mostra una web application, dove la pagina generata avrà le seguenti caratteristiche:

Se l'argomento **nome** è presente nell'url:

- verrà stampato la parola *"Ciao"* seguita dal valore dell'argomento **nome**.
- altrimenti verrà stampato *"Ciao anonimo"*.

```
from flask import Flask, request
app = Flask(__name__)

@app.route("/", methods=["GET"])
def hello():
    nome = request.args.get("nome")
    if nome:
        return "Ciao " + nome
    return "Ciao anonimo"

if __name__ == "__main__":
    app.run()
```

Caratteristiche principali:

- *Methods*: specifica il metodo con il quale recuperare i dati da un URL specificato, si può scegliere fra GET e POST (anche entrambi)

  - *GET*: il metodo più comune, viene inviato un messaggio GET e il server restituisce i dati. Se il parametro non è definito allora questo è il valore di default.
  - *POST*: utilizzato per inviare i dati del modulo HTML al server, per esempio utilizzando un form. I dati ricevuti dal metodo POST non vengono memorizzati nella cache dal server.
- *Request*: oggetto che mette a disposizione vari attributi e metodi utili per gestire i dati ricevuti dalla richiesta, fra i principali ci sono:
  - *request.args.get('arg1'):* se tra gli argomenti passati nella richiesta ve ne è uno denominato **arg1**, allora questo metodo restituisce il valore di tale argomento, altrimenti il risultato sarà **None**.
  - *get_json(force=False , silent=False , cache=True)*:

    questo metodo analizza i dati della richiesta JSON in entrata e li restituisce.

    Parametri opzionali:
    - *force*: se impostato su True viene ignorato.
    - *silent*: se impostato su True questo metodo fallirà silenziosamente e restituirà None.
    - *cache*: se impostato True i dati JSON analizzati vengono ricordati nella richiesta.
- *headers[]*: dizionario contenente le intestazioni HTTP della richiesta

2.3 HealthCheck

Healthcheck [2] esegue il wrapping di un oggetto dell'app Flask e aggiunge un modo per inserire semplici funzioni di controllo dello stato, le quali possono essere utilizzate per monitorare l'applicazione. È utile per verificare che le dipendenze siano attive e in esecuzione e che la propria applicazione sia in grado di rispondere alle richieste HTTP. Le funzioni Healthcheck sono esposte tramite un percorso Flask definito dall'utente in modo da poter utilizzare un'applicazione

di monitoraggio esterna (monit, nagios, Runscope, ecc.) per controllare lo stato e il tempo di attività dell'applicazione.

Healthcheck, inoltre, offre anche un semplice percorso Flask per visualizzare alcune informazioni sull'ambiente della propria applicazione. Per impostazione predefinita, include i dati sul sistema operativo, dell'ambiente Python, del processo corrente e della configurazione dell'applicazione. È possibile personalizzare le sezioni presenti o aggiungerne di nuove.

Installazione:

```
pip install flask
```

Esempio:

Il codice che segue inserisce una funzione di controllo, in questo caso banale verifica che l'addizione fra 2 numeri sia corretta (1+1=2).

```
from flask import Flask
from healthcheck import HealthCheck

app = Flask( __name__ )
health = HealthCheck(app, "/healthcheck" )

# funzione di controllo
def addition_works():
        if 1 + 1 == 2:
                return True, "addition works"
        else:
                return False, "the universe is broken"

health.add_check(redis_disponibile)
```

Caratteristiche principali:

- *Funzione di controllo*:

Le funzioni di controllo non accettano argomenti e devono restituire una tupla del tipo (bool, str).

- Il booleano indica se il controllo è stato superato o meno.
- La stringa indica l'output di cui eseguire il rendering per questo controllo. Utile per messaggi di errore/debug.

2.4 Logging

I log forniscono una serie di informazioni sul flusso che un'applicazione sta attraversando. Possono memorizzare informazioni, ad esempio quale utente o IP ha effettuato l'accesso all'applicazione. Se si verifica un errore, possono fornire più informazioni rispetto a una traccia dello stack comunicando qual era lo stato del programma prima che arrivasse alla riga di codice in cui si è verificato l'errore. Python fornisce un sistema di logging come parte della sua libreria standard, quindi è possibile aggiungere rapidamente il logging alla propria applicazione. [3]

Installazione:

```
pip install logging
```

Caratteristiche principali:

- *Livelli:*

  Per impostazione predefinita sono disponibili 5 livelli standard che indicano la gravità degli eventi. Ciascuno ha un metodo corrispondente che può essere utilizzato per registrare gli eventi a quel livello di gravità. I livelli definiti, in ordine crescente di gravità, sono i seguenti:

  - debug
  - info
  - warning
  - error
  - critical

  Esempio di chiamata dei log:

```
import logging

logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

Esempio di output:

```
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

L'output mostra il livello di gravità prima di ogni messaggio insieme a root. Questo formato, che mostra il livello, il nome e il messaggio separati da due punti (:), è il formato di output predefinito che può essere configurato per includere elementi come timestamp, numero di riga e altri particolari.
Si noti che i messaggi debug() e info() non sono stati registrati. Questo perché, per impostazione predefinita, il modulo di logging registra i messaggi con un livello di gravità uguale a WARNING o superiore. È possibile cambiarlo configurando il modulo di logging per registrare eventi di tutti i livelli. È possibile anche definire i propri livelli di gravità modificando le configurazioni, ma in genere non è consigliato in quanto può creare confusione con i log di alcune librerie di terze parti che potrebbero essere utilizzate.

- *Handlers:*

Gli handler entrano in gioco quando si desidera configurare i propri logger e inviano i messaggi di log a destinazioni configurate, come il flusso di output standard o su un file o su HTTP o alla propria posta elettronica tramite SMTP. Un logger creato può avere più di un handler, il che significa che è possibile configurarlo per essere salvato in un file di log oppure per l'invio tramite e-mail. Come i logger, è possibile anche impostare il livello di gravità negli handler. Ciò è utile se si desidera impostare più handler per lo stesso logger ma si desiderano livelli di gravità diversi per ciascuno di essi.

Esempio:

```
# logging_example.py

  import logging

# Create a custom logger
  logger = logging.getLogger(__name__)

# Create handlers
  c_handler = logging.StreamHandler()
  f_handler = logging.FileHandler('file.log')
  c_handler.setLevel(logging.WARNING)
  f_handler.setLevel(logging.ERROR)

# Create formatters and add it to handlers
  c_format = logging.Formatter('%(name)s - %(levelname)s -
  %(message)s')
  f_format = logging.Formatter('%(asctime)s - %(name)s -
  %(levelname)s - %(message)s')
  c_handler.setFormatter(c_format)
  f_handler.setFormatter(f_format)

# Add handlers to the logger
```

```
logger.addHandler(c_handler)
logger.addHandler(f_handler)

logger.warning('This is a warning')
logger.error('This is an error')
```

Nell'esempio precedente si vogliono registrare sulla console tutti i log di livello uguale e superiore al tipo WARNING, inoltre tutti i log di livello uguale e superiore al tipo ERROR vengono salvati in un file.

2.5 Dynaconf

Dynaconf [4] è un OSM (Object Settings Mappper), può leggere le variabili delle impostazioni da un insieme di diversi archivi di dati come file di impostazioni Python, variabili di ambiente, redis, memcached, file ini, file json, file yaml. È anche possibile personalizzare dynaconf per leggere da qualsiasi file si voglia.

Installazione:

```
pip install dynaconf
```

Caratteristiche:

- *Settings Files:*

  Dynaconf cercherà i file specificati nell'opzione **settings_file** avviando l'albero di ricerca sulla directory di lavoro corrente (la directory in cui si trova il programma).

```
from dynaconf import Dynaconf

settings = Dynaconf(settings_files=
["settings.toml","/etc/program/foo.yaml"])
```

Nell'esempio sopra, dynaconf proverà a caricare settings.toml dalla stessa directory in cui si trova il programma e poi continuerà ad attraversare le cartelle in ordine all'indietro finché non si trova la radice, nella quale può essere:

- il percorso in cui è stato richiamato il programma;
- la radice del sistema operativo;
- la radice specificata in root_path.

Il percorso "/etc/program/foo.yaml" verrà riconosciuto da dynaconf come un percorso assoluto e proverà a caricarlo direttamente dalla posizione specificata.
Inoltre per ogni file specificato in settings_files dynaconf proverà anche a caricare un percorso opzionale *nome_file*.**local**.*estensione_file*.
Ad esempio, settings_files=["settings.toml"] farà in modo che dynaconf cerchi prima settings.toml e poi anche settings.local.toml.

Parte II

Progetto e sviluppo

Capitolo 3
Progettazione

3.1 Introduzione

In questo capitolo verranno spiegate le varie fasi di progettazione che hanno portato alla realizzazione del progetto in questione, analizzando le problematiche riscontrate e le scelte trovate per risolverle. Per facilitare la comprensione le analisi relative alla progettazione verranno suddivise tenendo conto delle funzionalità espresse nel primo capitolo della tesi; anche se durante il tirocinio sono state affrontate parallelamente.

3.2 Business Logic

Questa funzionalità prevede 2 requisiti funzionali:

- Gestione e validazione dei dati in ingresso e in uscita
- Lavorazione dei dati in ingresso per generare quelli in uscita (process)

3.2.1 Gestione e validazione dei dati in ingresso e in uscita

Sia per i dati in ingresso che in uscita l'SDK deve fornire la possibilità all'utente finale di scegliere tra 2 opzioni possibili:

- Inserimento di un apposito file contenente lo schema (dell'input o dell'output) strutturato in una certa modalità;
- oppure la possibilità di definire lo schema inserendo delle istruzioni direttamente nel codice.

Se l'utente desidera inserire dei file JSON per definire lo schema, analogamente alla versione Java, questi devono essere inseriti in una apposita directory, che sarà specificata nella fase di implementazione.

Se l'utente desidera definire lo schema (input o output) direttamente nel codice è necessario ricercare fra le librerie Python quella più affine e adatta allo scopo. Inoltre è necessario testarla e fornire un modello di esempio in modo da facilitare il lavoro dell'utente finale.

È inoltre necessario stabilire una priorità, cosa succede se l'utente per definire uno schema inserisce sia un file JSON che delle istruzioni nel codice? La scelta progettuale è stata quella di dare precedenza all'istruzione scritta nel codice.

3.2.2 Lavorazione dei dati in ingresso per generare quelli in uscita (process)

Questo aspetto è il cuore del progetto, ovvero il fatto di dare la possibilità all'utente di inserire il proprio metodo process (che ha lo scopo di specificare come ottenere un risultato partendo da alcuni dati presi in ingresso), facendo in modo che sia l'SDK a verificare la correttezza della procedura ed evitare vari possibili errori.

Per indicare la presenza o meno di un errore durante la fase di process si utilizzano i codici di errore http, in questa situazionev sono previsti 3 possibili casi:

- 200: nessun errore riscontrato, stato generato l'output correttamente
- 400: si è verificato un errore, dovuto alla struttura non valida dei dati di input o output
- 500: si è verificato un errore, dovuto ad altri problemi

L'idea iniziale era quella di prendere i dati in ingresso, usare il metodo process realizzato dall'utente, generare l'output, dopodiché effettuare tutti i controlli previsti per garantire il rispetto dei vincoli.

Usando questo approccio però si è notato che se vengono inseriti dei dati di input scorretti (ovvero con una struttura diversa rispetto a quella prevista, si veda il punto 3.2.1) l'errore generato non era di tipo 400 (che correttamente segnalava la presenza di un errore relativo allo schema non conforme) ma invece generava un errore generico 500.

Questo perché veniva controllata la correttezza dello schema dei dati di input solo successivamente alla generazione dell'output, in questo modo il metodo process falliva e veniva generato un errore sbagliato.

Per risolvere questo problema è bastato suddividere i controlli sui dati in 2 fasi separate, prima veniva controllata la correttezza dei dati di input, poi generato l'output utilizzando il metodo process e infine controllata la correttezza dei dati di output ottenuti.
Usando questo approccio nella fase di test si sono verificati molti meno errori generici 500 rispetto all'approccio iniziale.

Per controllare se i dati rispettassero o meno la struttura prevista dallo schema si è scelto di utilizzare il metodo validate della libreria jsonschema, la quale lancerà un'eccezione di tipo SchemaError o ValidationError nel caso nello schema siano stati riscontrati dei problemi, niente altrimenti. Utilizzando il metodo in questione e una giusta rete di try ed except è possibile ottenere il risultato desiderato.

Nella Figura 3.1 viene rappresentato uno schema con lo scopo di riassumere le analisi fatte in questa sezione.

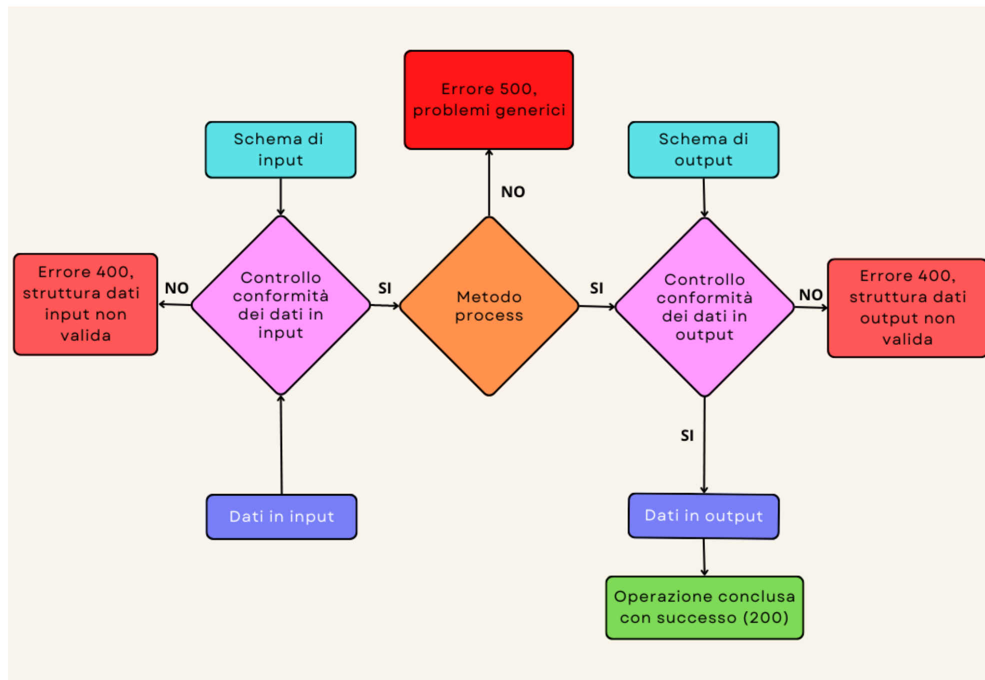

Figura 3.1: Schema per la gestione della Business Logic

3.3 Configurazione

Questa parte dell'SDK ha il compito di fornire all'utente gli strumenti per definire nelle modalità più semplici ed efficaci i dettagli relativi agli aspetti di configurazione del proprio progetto.

Per ottenere questo risultato si prevedono 2 possibilità:

- poter definire i dettagli di configurazione mediante un file di serializzazione (YAML)
- poter definire i dettagli di configurazione mediante l'utilizzo di variabili di ambiente.

Questi 2 approcci non sono mutualmente esclusivi, ovvero l'utente è libero di utilizzarli entrambi contemporaneamente; se questo accade e in entrambe le modalità viene fatto riferimento a un dettaglio specifico, la priorità viene data alla configurazione definita dalla variabile di ambiente.

3.3.1 Configurazione con file di serializzazione (YAML)

Per fornire una configurazione dinamica delle impostazioni del progetto è stato scelto l'utilizzo di Dynaconf, in grado di importare le specifiche di configurazione direttamente da un file fornito dall'utente (per maggior dettaglio a riguardo guardare la sezione 2.5).

Durante l'implementazione sarà inoltre necessario verificare che dato un certo file di serializzazione importato dall'SDK Java, questo dia luogo a un insieme di impostazioni di configurazione che siano equivalenti alla versione Python che si sta perfezionando.

3.3.2 Configurazione con variabili di ambiente

Il modo più efficace per modificare o aggiungere delle variabili d'ambiente all'interno di un programma è farlo utilizzando il modulo os. In particolare, utilizzando l'operatore [] os.environ, il quale è un dizionario che ha come chiave il nome della variabile di ambiente (da creare o modificare) e come valore il valore relativo alla variabile d'ambiente; entrambi i parametri devono essere delle stringhe, altrimenti viene generato un errore.

3.4 Manifest

Per la generazione del manifest deve essere creata una classe apposita avente come attributi tutti i campi che sono considerati necessari per rappresentarla, ovvero gli stessi campi previsti per l'SDK Java. Il valore di questi campi deve essere settato durante l'esecuzione del programma, utilizzando gli opportuni dati di configurazione oppure utilizzando adeguati valori di default. La stampa del manifest deve avere una struttura identica alla versione prevista dall'SDK Java.

Deve essere prevista la possibilità da parte dell'utente di inserire informazioni aggiuntive nel manifest utilizzando un opportuno file. L'idea è di verificare la presenza o meno di questo file prima di procedere con la generazione del

manifest, nel caso in cui il file fosse presente vengono estratte le informazioni necessarie e aggiunte al manifest generato.

Deve essere possibile generare il manifest anche da linea di comando, questo è possibile realizzarlo dando all'utente la possibilità di inserire un parametro opzionale durante l'avvio del programma. Se questo parametro viene inserito, in automatico il programma deve prevedere la stampa sulla linea di comando del manifest generato.

3.5 Startup and shutdown

L'SDK deve fornire la possibilità all'utente di inserire funzioni o procedure che verranno eseguite automaticamente durante la fase di avvio o/e durante la fase di terminazione dell'applicativo. Per ottenere questo risultato bisogna dare la possibilità all'utente di scrivere nel proprio main funzioni la cui intenzione è di essere richiamate durante le fasi di avvio o terminazione del programma. Queste funzioni dovranno poi essere passate come parametri a una funzione che le richiamerà nel momento opportuno.

3.6 Health checks

Per controllare l'integrità del programma si è scelto di utilizzare il modulo Healthcheck, il quale fornisce la possibilità di visualizzare alcune informazioni utili sullo stato del sistema e di definire alcune funzioni per verificarne l'affidabilità (per maggior dettaglio a riguardo guardare la sezione 2.3).

L'utente avrà la possibilità di inserire delle proprie funzioni di controllo direttamente nel proprio main. Queste funzioni dovranno poi essere passate come parametri a una funzione che le richiamerà nel momento opportuno.

L'utente che voglia visualizzare le informazioni relative allo stato del sistema deve avere entrambe le possibilità già previste per la creazione del manifest, ovvero:

- Poterle visualizzare sul terminale inserendo un apposito parametro durante l'istruzione di avvio;
- Poterle visualizzare sulla pagina web '127.0.0.0:18080/api/v1/healthcheck.

3.7 Logging

Per la gestione dei log si utilizza il modulo logging (per maggior dettaglio a riguardo guardare la sezione 2.4).

L'idea è di definire un modulo log avente al suo interno:

- Un dizionario contenente le informazioni essenziali per la generazione di un log.
- Una funzione che verrà richiamata durante l'esecuzione in determinate situazioni particolari e stamperà su stdout (o stderr) una serie di informazioni collegate all'evento in questione, attraverso uno specifico log.

Il dizionario conterrà i seguenti dati relativi alla generazione di un log:

- Nome del servizio che lo ha generato
- Data e orario in cui è stato generato
- Nome del livello
- Messaggio
- Latency: tempo che ha impiegato il servizio per dare una risposta
- Correlation id: id univoco che identifica il processo
- dati extra relativi a chi ha generato il log:
    - nome del processo
    - nome del metodo/funzione
    - nome del file
    - numero della linea
    - nome del thread
    - nome del percorso

La funzione del modulo, quando richiamata, avrà il compito di utilizzare lo schema del dizionario definito e i dati in possesso per la generazione di uno specifico log.

Tra i vari campi è necessario dare importanza al calcolo del correlation id, il quale può essere realizzato in 2 modi:

- trasmesso come parametro alla funzione del modulo
- in alternativa, generato casualmente all'interno della funzione stessa

Capitolo 4

Implementazione

4.1 Introduzione

In questo capitolo verranno commentate le pozioni di codice più significative che hanno portato alla realizzazione dei requisiti previsti nella fase di progettazione.

Il progetto è stato strutturato nei seguenti moduli Python:

- Config.py: utilizzato per importare correttamente gli elementi di configurazioni presenti nel file di serializzazione inserito dall'utente;
- Log.py: contiene il codice relativo alla generazione dei log;
- Manifest.py: definisce la classe Manifest, necessaria per la generazione del manifest;
- Logic.py: definisce la classe Logic, la quale definisce gli elementi della logica prevista;
- Service.py: definisce la classe Service, si occupa di rendere disponibili all'utente tutti i servizi dell'SDK
- App.py: questo è il modulo principale, ovvero quello che richiama tutti gli altri. In particolare i due metodi principali sono:
    - fly: che crea un'istanza della classe Service e rende operativa l'erogazione dei servizi;
    - main: viene richiamato direttamente dall'utente, il quale può inserire una serie di parametri, ovvero:
        - process: obbligatorio, funzione da utilizzare per la generazione dell'output;
        - input: opzionale, classe per lo schema dati di input;
        - output: opzionale, classe per lo schema dati di output;

- onStartup: opzionale, funzione da richiamare all’avvio;
- onShutdown: opzionale, funzione da richiamare al termine;
- healthcheck: opzionale, regole di healthcheck da inserire.

Inoltre richiama il metodo fly.

I moduli descritti sopra non sono visibili o modificabili dall’utente finale, il quale li importerà nel proprio progetto come una qualsiasi libreria Python.

Inoltre per facilitarne l’utilizzo immediato sono stati previsti anche un modulo contenente un esempio di utilizzo (Application.py) e un file di documentazione che ne spiega le caratteristiche principali (README.md)

4.2 Business Logic

4.2.1 Gestione e validazione dei dati in ingresso e in uscita

Nel caso in cui l’utente voglia inserire lo schema dei dati in ingresso direttamente nel proprio codice lo può fare definendo una classe apposita, la quale deve essere sottoclasse della classe BaseModel [5] della libreria pydantic; si può fare in modo analogo per lo schema dei dati in uscita. Nell’immagine 4.1 vi è un esempio della possibile classe.

```
class InputModel(BaseModel):
    """
    This is the description of the input model
    """
    url: str = Field(
        #"https://expert.ai",  #defualt
        title='url',
        description='url to download and scrape',
    )
    def validate_input(self):
        if self["url"] == "" :
            raise ValidationError("Il campo url non può essere vuoto")
    class Config:
        title = 'Input'
```

Figura 4.1: Gestione struttura dati ingresso da codice

Per utilizzare la classe realizzata è sufficiente assegnarla al parametro opzionale previsto durante l’invocazione del metodo app.main (metodo main della classe app).

Nel caso in cui invece non si crei una classe apposita per definire lo schema, e di conseguenza il relativo parametro opzionale del metodo app.main non viene utilizzato, l'SDK cercherà la definizione dello schema in un file specifico JSON.

La porzione di codice che effettua questo controllo (figura 4.2 e 4.3) si trova nel modulo Service.py all'inizio del metodo listen, ovvero il metodo principale presente nel modulo, il quale per ricercare il file JSON utilizza il percorso scelto dall'utente presente nel file di configurazione. Nel caso in cui l'utente abbia scelto di definire lo schema direttamente da codice verrà generato automaticamente il relativo file JSON, utilizzando sempre il percorso presente nel file di configurazione.

```
path = os.getcwd()
input_schema = None
try:
    #Se non è stato definita la struttura dell'input allora la recupera dal file json
    if self.nlflow.input == None:
        with open(os.path.join(path, config.service.input.jsonSchema)) as f:
            data = f.read()
            if data != "":
                try:
                    self.nlflow.input = json.loads(data)
                except Exception as e:
                    logging.error(e)
    else:
        with open(os.path.join(path, config.service.input.jsonSchema), 'w') as f:
            try:
                json_schema=self.nlflow.input.schema_json(indent=2)
                f.write(json_schema)
                self.nlflow.input=json.loads(json_schema)
            except Exception as e:
                logging.error(e)
except:
    json_schema=self.nlflow.input.schema_json(indent=2)
    self.nlflow.input=json.loads(json_schema)
input_schema = self.nlflow.input
```

Figura 4.2: Gestione struttura dati ingresso da file

```
output_schema = None
try:
    #Se non è stato definita la struttura dell'output allora la recupera dal file json
    if self.nlflow.output == None:
        with open(os.path.join(path, config.service.output.jsonSchema)) as f:
            data = f.read()
            if data != "":
                try:
                    self.nlflow.output = json.loads(data)
                except Exception as e:
                    logging.error(e)
    else:
        with open(os.path.join(path, config.service.output.jsonSchema), 'w') as f:
            try:
                json_schema=self.nlflow.output.schema_json(indent=2)
                f.write(json_schema)
                self.nlflow.output=json.loads(json_schema)
            except Exception as e:
                logging.error(e)
except:
    json_schema=self.nlflow.output.schema_json(indent=2)
    self.nlflow.output=json.loads(json_schema)
output_schema = self.nlflow.output
```

Figura 4.3: Gestione struttura dati output da file

4.2.2 Lavorazione dei dati in ingresso per generare quelli in uscita (process)

Come precedentemente anticipato nel modulo Logic.py è presente la classe Logic che ha come scopo quello di definire e gestire gli aspetti riguardanti la business logic dell'SDK.

I suoi attributi e il relativo scopo sono i seguenti:

- Input: schema di input
- Output: schema di output
- Processfn: funzione di process da richiamare per generare l'output
- Handlerfn: funzione definita dinamicamente all'interno del metodo process
- Initfn: funzione da richiamare all'avvio del programma
- Closefn: funzione da richiamare al termine del programma

Il metodo principale della classe è process, il quale richiede come parametri di ingresso i valori da assegnare ai seguenti attributi di classe: input, output e processfn. Inoltre in questa funzione viene anche generata un'altra funzione, ovvero quella da assegnale all'attributo di classe Handlerfn, la quale verrà richiamata tutte le volte che l'utente vuole generare un output partendo da alcuni dati ricevuti in ingresso.

Per fare questo la funzione in questione (Figura 4.4) richiede come parametro di ingresso un variabile contenente i dati di input, i quali vengono validati utilizzando lo schema di input precedentemente assegnato come visto nella sezione 4.2.1. Successivamente viene generato l'output utilizzando la funzione di process e i dati di input, per poi procedere con la validazione dell'output utilizzando il relativo schema e il ritorno dati validati.

Per assicurarsi che durante questo procedimento non si verifichino degli errori e nel caso questi vengano gestiti opportunamente è stato utilizzato l'approccio ideato nella fase di progettazione, presente nella sezione 3.2.2.

```python
def handlerfn(input):
    try:
        input_schema = self.input
        output_schema = self.output
        validate(input, input_schema)
        self.with_input(input)
    except (SchemaError, ValidationError) as e:
        output = {
            "erroreType":"Errore validation input",
            "error": json.dumps(f"{e}"),
            "input": input,
            "inputSchema": input_schema,
        }
        return output, 400
    except Exception as e:

        output = {
            "error": json.dumps(f"{e}")
        }
        return output, 500

    try:
        output = self.processfn(input)
        validate(output, output_schema)
        self.with_output(output)
        return output
    except (SchemaError, ValidationError) as e:
        output = {
            "erroreType":"Errore validation output",
            "error": json.dumps(f"{e}"),
            "output": output,
            "outputSchema": output_schema
        }
        return output, 400
    except Exception as e:
        output = {
            "error": json.dumps(f"{e}")
        }
        return output, 500
```

Figura 4.4: Funzione handlerfn della classe Logic

4.3 Configurazione

Il modulo config.py contiene le istruzioni necessarie per recuperare i dati di configurazione scelti dall'utente partendo da un proprio file di configurazione. Come anticipato durante il capitolo relativo alla progettazione nella sezione 3.3.1, viene utilizzato Dynaconf per importare le specifiche presente nel file di serializzazione, il quale deve essere presente all'interno della cartella resources del progetto.

```python
from dynaconf import Dynaconf
import os

path = os.getcwd()

config = Dynaconf(
    envvar_prefix="NLFLOW",
    settings_files=[os.path.join(path, 'src/resources/config.yaml'), '.secrets.yaml'],

)
```

Figura 4.5: Configurazione da file YALM

In alternativa o in aggiunta è possibile inserire degli elementi di configurazione utilizzando le variabili di ambiente, nella Figura 4.6 vi un esempio di utilizzo presente nel modulo Application.py.

```
os.environ['NLFLOW_CUSTOM__SERVICE__NAME'] = 'test2'
os.environ['NLFLOW_CUSTOM__SERVICE__ID'] = 'id_23'
os.environ['NLFLOW_CUSTOM__SERVICE__INPUT__JSONSCHEMA'] = "\\src\\resources\\inputSchema.json"
```

Figura 4.6: Configurazione con variabili di ambiente

4.4 Manifest

Il modulo manifest.py contiene la classe Manifest, la quale contiene i seguenti metodi: __init__, to_json e manifest_additions.

Il metodo __init__ (Figura 4.7) definisce e inizializza le variabili della classe assegnandoli i relativi valori. Questi valori vengono recuperati partendo dai parametri di configurazione presenti nel file di serializzazione o impostati con l'utilizzo delle variabili di ambiente o in alternativa vengono settati utilizzando dei parametri di default.

```
class Manifest:
    def __init__(self):
        self.version = '1.0'
        self.id = config.get('service.id', None)
        self.autore = config.get('service.autore',"")
        self.name = config.get('service.name',None)
        self.type = config.get('service.type',"process.http")
        self.description = config.get('service.description','')
        self.input = {}
        self.output = {}
        self.groups = config.get('security.groups',[])
        self.params = config.get('workflow',{})
        self.image = config.get('service.image', '')
        self.envVars = config.get('service.envVars',[])
        self.confMappings = config.get('service.confMappings',[])
        self.parameters_descriptor = config.get('service.parameters_descriptor',[])
        self.categories = config.get('service.categories', [])
```

Figura 4.7: Inizializzazione campi manifest

Il metodo to_json ha il compito di restituire un dizionario contenente la stampa del manifest con una struttura prestabilita, la quale è ideantica a quella generata dal progetto Java.

```
def to_json(self):
    self.manifest_additions()
    return {
        "id": self.id,
        "version": self.version,
        "name": self.name,
        "description": self.description,
        "autore": self.autore,
        "type": self.type,
        "input": self.input,
        "output": self.output,
        "categories": self.categories,
        "image": self.image,
        "envVars": self.envVars,
        "params": self.params,
        "confMappings": self.confMappings,
        "groups": self.groups,
        "parameters_descriptor": self.parameters_descriptor
    }
```

Figura 4.8: Stampa manifest

Il metodo manifest_additions ha il compito di gestire il caso in cui l'utente voglia inserire alcune informazione aggiuntive nel proprio manifest, infatti il metodo ricercherà la presenza di questo file e cercherà di verificare se al suo interno vi sono delle informazioni utili da assegnare ad alcune variabili di classe che alternativamente rimarrebbero settate a un valore nullo.

Inoltre per permettere la generazione del manifest dalla linea di comando inserendo il campo opzionale 'generateManifest' durante l'esecuzione del programma; infatti nel modulo service.py vi è presente un controllo (Figura 4.9) in grado di rivelare la presenza di questo parametro e in caso positivo genera e stampa sul termina il manifest richiesto.

```
def generateManifest():
    manifest = Manifest()
    try : inputHideList = config.service.input.hideList
    except : inputHideList= []
    try : outputHideList = config.service.output.hideList
    except : outputHideList= []

    manifest.input = {
        "jsonSchema_I": input_schema,
        "hideList_I": inputHideList
    }
    manifest.output = {
        "jsonSchema_O": output_schema,
        "hideList_O": outputHideList
    }
    return manifest.to_json()

if 'generateManifest' in sys.argv:
    print("Manifest: ")
    print(generateManifest())
```

Figura 4.9: Stampa manifest da terminale

4.5 Startup and shutdown

Se l'utente vuole inserire delle funzioni da eseguire durante la fase di avvio o durante quella di terminazione è sufficiente assegnare queste funzioni al corrispondente parametro opzionale durante l'invocazione del metodo app.main. In particolare al parametro onStartup deve essere assegnata la funzione da eseguire durante l'avvio e al parametro onShutdown quella da eseguire durante la terminazione.

Nel modulo app.py, una volta che viene richiamato il metodo fly, questo per prima cosa controlla se è stata passata una funzione tramite il parametro onStartup, in questo caso viene eseguita la funzione in questione. Successivamente viene istanziato un oggetto di tipo Service e viene lanciato il metodo listen della classe, il quale resta in attesa di eventuali richieste per l'erogazione dei servizi offerti dall'SDK.

Una volta che l'utente non è più interessato a richiedere determinati servizi il metodo fly effettua un altro controllo per verificare se il parametro onShutdown sia una funzione, in caso positivo viene eseguita e il programma termina l'esecuzione.

4.6 Health checks

Nel modulo health.py vi è un semplice creazione dell'oggetto health di classe HealthCheck, utilizzata per controllare lo stato del sistema e l'inserimento di funzioni di controllo personalizzate.

Se l'utente vuole utilizzare delle proprie funzioni di controllo lo può fare assegnando queste al parametro opzionale previsto durante l'invocazione del metodo app.main, ovvero healtcheck.

All'interno nel metodo main del modulo app verrà verificata la presenza di queste funzioni (Figura 4.10), in caso positivo vengono inserire alle regole di controllo del sistema, ovvero all'oggetto check utilizzando il corrispondente metodo di classe add_check.

```
def main(process,input=None,output=None,onStartup=None, onShutdown=None,healthcheck=None ):
    if healthcheck:
        health.add_check(healthcheck)
```

Figura 4.10: Inserimento funzioni di controllo

L'utente, come nel caso del manifest, può richiedere la stampa su terminale dello stato del sistema all'avvio del programma, per fare questo è sufficiente inserire il campo opzionale 'generateHealth' durante la fase di avvio; infatti nel modulo service.py vi è presente un controllo in grado di rivelare la presenza di questo parametro e in caso positivo genera e stampa sul terminale le informazioni correnti sullo stato del sistema, oltre che alle eventuali funzioni di controllo presenti inserite dall'utente stesso.

4.7 Logging

Nel modulo log.py viene utilizzato il modulo di logging e vengono implementati gli aspetti previsti nel capitolo di progettazione nella sezione 3.7.

In particolare nella Figura 4.11 vi è la definizione del dizionario dict_log, il quale contiene le informazioni previste per la generazione di uno specifico log. Per fare questo si utilizza un'apposita formattazione delle stringhe con lo scopo di recuperare le informazioni necessarie.

```
name=(ntpath.basename(os.getcwd()))
my_date = datetime.now().strftime('%Y-%m-%dT%H:%M:%S.%fZ')
extra = {
    "name":"%(name)s",
    'processName': '%(processName)s',
    'method': '%(funcName)s',
    'file': '%(filename)s',
    'line': '%(lineno)d',
    'thread': '%(thread)d',
    'threadName': ' %(threadName)s',
    'pathname':'%(pathname)s'
}

dict_log = {
    "essentia-service-name":name,
    "timestamp":my_date,
    "level":"%(levelname)s",
    "message":"%(message)s",
    "latency": "0ms",
    "correlationId":str(uuid.uuid4()),
    "extra":extra
}
```

Figura 4.11: Dizionario dati di log

Nella Figura 4.12 vi è la definizione della funzione configure_logging, il cui scopo è quello di generare uno specifico log in base alla situazione nella quale è stata richiamata. All'interno della funzione viene modificato il valore del dizionario relativo all'orario di stampa, aggiornandolo con quello relativo all'istante in questione.

```
console_handler=None

def configure_logging():
    global console_handler
    dictConfig(DEFAULT_LOGGING)

    dict_log ["timestamp"] = datetime.now().strftime('%Y-%m-%dT%H:%M:%S.%fZ')
    default_formatter = (logging.Formatter(json.dumps(dict_log), "%d/%m/%Y %H:%M:%S"))

    try:
        logging.root.removeHandler(console_handler)
    except:
        pass
    console_handler = logging.StreamHandler()
    console_handler.setLevel(logging.DEBUG)
    console_handler.setFormatter(default_formatter)

    logging.root.setLevel(logging.DEBUG)
    logging.root.addHandler(console_handler)
```

Figura 4.12: Funzione per generare i log

4.8 Framework Flask

Il metodo listen della classe Service ha il compito di rimanere in ascolto per rispondere alle richieste dell'utente offrendo i relativi servizi.

La prima parte del metodo è già stata commentata precedentemente nella sezione 4.2.1 (Figure 4.2 e 4.3).
La seconda parte (Figura 4.13) si occupa proprio delle richieste dell'utente, le quali vengono gestite tramite l'utilizzo del framework Flask (sezione 2.2 per maggiori dettagli). In particolare si accettano richieste per l'erogazione di tre servizi:

- generazione del manifest (metodo GET);
- stampa delle informazioni relative allo stato del sistema (metodo GET);
- gestione della business logic (metodo POST).

Per quanto riguarda l'ultimo punto, essendo l'aspetto più delicato e complesso vengono riportati di seguito i dettagli relativi alla sua implementazione:

- vengono analizzati ed estratti i dati JSON dell'oggetto request, nei quali ci si aspetta si trovino dei dati aventi un formato corrispondente allo schema di ingresso previsto;
- questi dati in ingresso vengono passati alla funzione handlerfn (Figura 4.4) e, almeno che non si verifichino errori i quali vengono già gestiti all'interno della funzione stessa, assegna il risultato dei dati ottenuti alla variabile ret;
- si controlla il caso in cui l'utente abbia specificato il correlation id relativo al processo all'interno degli headers, in caso contrario viene generato casualmente;
- viene calcolata la latenza, ovvero il tempo impiegato per l'esecuzione del servizio, ciò avviene effettuando una differenza tra il tempo registrato al termine del servizio e all'inizio della sua fruizione;
- viene richiamato il metodo configure_logging per stampare su terminale un log contenente le informazioni relative all'esito del processo;

- vengono restituiti i dati in output (generati dal metodo process) alla pagina web dove ne è stata generata la richiesta.

```
app = Flask(self.name)
CORS(app)

@app.route('/api/v1/manifest', methods=["GET"])
def manifest_handler():
    return generateManifest()

# Add a flask route to expose information
@app.route('/api/v1/health', methods=["GET"])
def health_handler():
    return health.run()
    #return str(datetime.now())

@app.route('/api/v1/nlflow', methods=["POST"])
def nlflow_handler():
    start=time.time()
    ret = self.nlflow.handlerfn(request.get_json())

    try:
        corId = request.headers['X-Correlation-Id']
        dict_log["correlationId"]=corId
    except:
        dict_log["correlationId"]=str(uuid.uuid4())
    end=time.time()
    dict_log["latency"] = end - start

    configure_logging()
    logging.info("nlflow")
    return ret
```

Figura 4.13: Utilizzo di Flask

Conclusioni

Il lavoro svolto descritto in questa tesi ha portato alla realizzazione dell'SDK NLFlow Python.

Per ottenere questo traguardo è stato fondamentale poter prendere spunto dalla versione Java dell'SDK in questione, in particolare per osservarne i risultati ottenuti e valutarne gli aspetti critici da migliorare. Inoltre è stato particolarmente utile partire lo sviluppo da un progetto almeno parzialmente già definito e strutturato, questo ha fatto si che il tempo a disposizione fosse dedicato unicamente al miglioramento e all'ampiamento dei servizi offerti dall'SDK. Tutto ciò ha portato a una facilitazione delle fasi di progettazione e implementazione, anche se ha reso più impegnative la fase inziale riguardante lo studio del problema affrontato, in quanto vi erano 2 progetti (quello Java completo e quello Python parziale) che andavano studiati e capiti nel dettaglio.

Quindi partendo dalla fase iniziale dove si è dedicato gran parte del tempo allo studio del problema affrontato e alla definizione degli obbiettivi principali, si è passati alla fase di progettazione dove sono state definite le modalità con la quale raggiungere suddetti obbiettivi, per poi concludersi con l'implementazione e la realizzazione vera e propria del progetto in questione.

Gli obbiettivi definiti durante la fase di progettazione sono stati raggiunti, in definitiva è stato realizzato un SDK Python in grado di fornire varie funzionalità per i futuri sviluppatori dell'azienda; dalla gestione della business logic alla

generazione del manifest, dal controllo sullo stato del sistema alla configurazione iniziale, dall'utilizzo di funzioni specifiche durante la fase di avvio e terminazione del programma fino alla gestione dei log.

Per quanto riguarda gli sviluppi futuri il prossimo passo sarà quello di rendere disponibile questo SDK ai dipendenti dell'azienda tramite la piattaforma aziendale, per facilitarne l'utilizzo ho anche provveduto alla realizzazione di un apposito file di documentazione, utile per intraprendere i primi passi nell'utilizzo di questo strumento e sfruttarne efficacemente tutti i vantaggi e le potenzialità.

Bibliografia

[1] “Flask” URL: https://flask.palletsprojects.com/en/2.2.x/quickstart/

[2] “Healthcheck” URL: https://libraries.io/pypi/healthcheck

[3] “Logging in Python” URL: https://realpython.com/python-logging/

[4] “dynaconf” URL: https://pypi.org/project/dynaconf/0.5.4/

[5] “BaseModel” URL: https://pydantic-docs.helpmanual.io/usage/models/