



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

APPLICAZIONE DI FACPL PER LA GESTIONE
DELLE POLICY NEI SISTEMI CLOUD: UN
CASO DI STUDIO CONCRETO CON
OPENNEBULA

APPLYING FACPL TO POLICY MANAGEMENT
IN CLOUD SYSTEMS: A PRACTICAL CASE
STUDY WITH OPENNEBULA

MATTEO MONICOLINI

Relatore: *Rosario Pugliese*

Anno Accademico 2023-2024

INDICE

Elenco delle figure	3
1 INTRODUZIONE	7
1.1 Guida alla lettura	8
2 IL LINGUAGGIO FACPL E IL SOFTWARE OPENNEBULA	9
2.1 Il linguaggio FACPL	9
2.2 Struttura delle policy e delle richieste di FACPL	10
2.3 Componenti del sistema FACPL	11
2.4 Modalità di utilizzo di FACPL	12
2.5 Il software OpenNebula	13
2.6 Architettura di OpenNebula	14
2.7 XML-RPC e OpenNebula	15
3 INTEGRAZIONE DI OPENNEBULA CON FACPL	17
3.1 Idea di base	17
3.2 OpenNebula API	18
3.3 Logging	20
3.4 Gestione delle virtual machine generiche	21
3.5 Gestione delle virtual machine di OpenNebula	23
3.6 Interazione con il ContextStub e le PEPActions	26
3.7 Accesso dall'esterno del progetto	28
3.8 Utilizzo di Maven	32
3.9 Introduzione alla Web-app e ideazione del front-end	34
3.10 Sviluppo del backend	36
3.11 Integrazione della web-app con Maven	40
3.12 Testing	41
3.13 Rilascio del progetto	44
4 UTILIZZO DEI PROGETTI IN UN SERVER REALE	47
4.1 Installazione dei due progetti	48
4.2 Specifiche dell'esempio	49
4.3 Risultati dell'esecuzione	50
5 CONCLUSIONI E SVILUPPI FUTURI	55
5.1 Criticità e punti di forza della libreria Java di FACPL	55
5.2 Sviluppi futuri	56
Ringraziamenti	59
Bibliografia	61

ELENCO DEI LISTING

Listing 1	Esempio di policy in FACPL	10
Listing 2	Esempio di richiesta in FACPL	11
Listing 3	Metodo make di FileLoggerFactory	20
Listing 4	Costruttore ContextStub_Default	21
Listing 5	VirtualMachineService	22
Listing 6	VMsInfo logging e filtro	22
Listing 7	Classe astratta per i comandi	23
Listing 8	Classe per avviare una VirtualMachine	24
Listing 9	Classe per freezzare(sospendere) una VirtualMachine	25
Listing 10	Context di OpenNebula	26
Listing 11	Classe PEPAction adattata	28
Listing 12	Classe FACPLHandlingTemplate	29
Listing 13	Metodo di setup per OpenNebula	30
Listing 14	Metodo processFolderContent	31
Listing 15	Interfaccia FolderContentStrategy	32
Listing 16	Repository locale	33
Listing 17	Dipendenza di OpenNebula	33
Listing 18	Esclusione di logback	36
Listing 19	PolicyController	37
Listing 20	ServiceConfig	38
Listing 21	Metodo validatePolicies	39
Listing 22	GlobalExceptionHandler	40
Listing 23	CreateVMTest	42
Listing 24	PolicyControllerTest	43
Listing 25	config.properties	49

ELENCO DELLE FIGURE

Figura 1	Processo di valuatione di FACPL	11
Figura 2	Front-end di policy manager (web-app)	35
Figura 3	Dashboard di OpenNebula con una sola virtual machine	50
Figura 4	Richieste di P_1 e P_2	50
Figura 5	Valutazioni di FACPL sulle due richieste	51
Figura 6	Dashboard di OpenNebula con i due host bilanciati	51
Figura 7	Dashboard con le virtual machine alternate fra i due host	51
Figura 8	Valutazione di FACPL	52
Figura 9	Dashboard con entrambi gli host al 100% di utilizzo	52
Figura 10	virtual machine stoppate automaticamente	52
Figura 11	Richiesta di rilascio	53
Figura 12	valutazione di FACPL	53
Figura 13	Virtual machine stoppata	53

"Sarà che prendo troppo spesso Trenitalia ma io non credo nelle coincidenze"
— *Pinguini tattici nucleari "Test di ingresso di medicina"*

INTRODUZIONE

In questa tesi si cercherà di dare una risposta al problema della gestione delle policy nei sistemi cloud, con particolare interesse riguardo le politiche di creazione delle virtual machine e di bilanciamento delle risorse. Il linguaggio utilizzato per le policy è FACPL, ancora poco esplorato nonostante i molti vantaggi rispetto alle alternative presenti in letteratura.

Il sistema di gestione cloud scelto è OpenNebula, un software completamente open-source che permette di gestire infrastrutture cloud su diversi livelli e integra molti servizi utili per la gestione delle risorse.

L'obiettivo finale è quello di fornire un'implementazione concreta di FACPL come gestore di richieste di creazione di virtual machine che sia in grado di funzionare in un ambiente cloud reale su cui è installato OpenNebula. Questo permetterà quindi di decidere le policy con cui si permette a specifici utenti di creare virtual machine su specifici host, così come di bilanciare le risorse tra i vari host o di decidere di spegnere alcune macchine in base a determinate condizioni.

L'implementazione sarà corredata con adeguata metodologia di logging delle informazioni e utilizzo delle pratiche di buona programmazione per rendere il codice facilmente mantenibile e ampliabile in futuro.

I principali problemi che saranno affrontati e risolti sono quelli scaturiti dalla necessità di far interagire le API di OpenNebula con la struttura fornita da FACPL senza snaturare nessuno dei due progetti in modo anche da dimostrare la facile adattabilità di FACPL. Un'altra questione che verrà trattata sarà l'integrazione di un software di gestione della build come Maven, con due progetti distribuiti esclusivamente come file .jar.

Per concludere verrà fornito uno spunto di web-app da cui sarà possibile provare le funzionalità del progetto sviluppato oltre che partire per una

possibile implementazione in un vero server che integra un meccanismo di autenticazione e autorizzazione.

Questa tesi si pone l'obiettivo di mostrare la semplicità di utilizzo di FACPL e di evidenziare i suoi pro e contro di modo che una persona che intende realizzare un progetto in cui è richiesta la valutazione di policy di accesso, possa scegliere FACPL come linguaggio se fa al caso suo e con la possibilità di avere un'idea su come può essere utilizzato in un caso concreto.

1.1 GUIDA ALLA LETTURA

I capitoli che seguono sono così organizzati:

- *capitolo 2*: introduce i concetti di base su FACPL e OpenNebula, con particolare attenzione alle componenti usate in questa tesi;
- *capitolo 3*: descrive nel dettaglio l'implementazione realizzata e le scelte di design effettuate;
- *capitolo 4*: mostra un'esempio di utilizzo dei progetti sviluppati in un sistema cloud reale;
- *capitolo 5*: conclude la tesi con una sintesi dei risultati ottenuti e dei possibili sviluppi futuri.

IL LINGUAGGIO FACPL E IL SOFTWARE OPENNEBULA

L'obiettivo di questo capitolo è introdurre i concetti di base del linguaggio FACPL e del software OpenNebula, necessari per comprendere il lavoro svolto in questa tesi. In particolare, sarà discusso il linguaggio FACPL, le sue caratteristiche principali e le motivazioni che hanno portato al suo utilizzo. Successivamente verrà presentato il software OpenNebula, sarà descritta la sua architettura e saranno evidenziate funzionalità ritrovabili all'interno dei progetti descritti nei capitoli successivi.

2.1 IL LINGUAGGIO FACPL

FACPL [7] è stato ideato nel 2012 come risposta alla necessità di disporre di sistemi di controllo degli accessi più flessibili e adattabili rispetto a quelli già esistenti. In particolar modo il linguaggio viene spesso paragonato con XACML, questo per due motivi principali:

- XACML è attualmente il linguaggio più largamente utilizzato per il controllo degli accessi;
- FACPL è stato sviluppato proprio come risposta ad alcune evidenti problematiche di XACML.

Il problema principale che FACPL punta a risolvere è la mancanza di una semantica definita formalmente in XACML, il che rende difficile ideare delle tecniche di analisi. Trae quindi larga ispirazione da XACML, in particolar modo per la struttura base delle policy e anche per una parte della terminologia.

2.2 STRUTTURA DELLE POLICY E DELLE RICHIESTE DI FACPL

In FACPL le policy sono composte da regole e insiemi di policy, indicati come Rule e PolicySet:

- le Rule sono le policy basilari e presentano un nome, la decisione positiva o negativa che è data dalla loro valutazione (che può essere Permit o Deny) e un'espressione che viene confrontata con la richiesta che sta venendo valutata;
- i PolicySet sono insiemi di policy e presentano un nome, un algoritmo, detto *combining algorithm*, e un'espressione che valuta se l'insieme di policy è applicabile alla richiesta. I *combining algorithm*¹ sono funzioni che prendono in input le decisioni delle policy dell'insieme e restituiscono una decisione unica. Le policy in un PolicySet possono essere sia delle Rule che altri PolicySet annidati.
- le obligation sono delle componenti opzionali sia nelle regole che negli insiemi di policy che rappresentano delle azioni da eseguire. Permettono di specificare se devono essere eseguite in caso di decisione positiva o negativa. Possono essere obbligatorie o opzionali, questo significa che nel primo caso se l'azione espressa dalla obligation non viene eseguita con successo la valutazione non sarà positiva, mentre nel secondo caso sì.

In questo listing si può vedere un esempio di policy in linguaggio FACPL:

Listing 1: Esempio di policy in FACPL

```

1 PolicySet Online_Generated{ permit-overrides
2   policies:
3   PolicySet Release_Policies { permit-overrides
4     target:equal("RELEASE", action/action-id)
5     && (equal("P_1", subject/profile-id) || equal("P_2", subject/
6       profile-id))
7     policies:
8     Rule hyper_1_release (permit
9       target:( in (resource/vm-name, system/hyper1.vm-names))
10      obl: [M release("0", resource/vm-name)]
11    )
12    Rule hyper_2_release (permit
13      target:( in (resource/vm-name, system/hyper2.vm-names))
14      obl: [M release("1", resource/vm-name)]
15    )
16  }

```

¹ <https://facpl.readthedocs.io/en/latest/txt/facpl.html?highlight=policy#evaluation>

Il linguaggio è pensato per essere scritto direttamente in questa sintassi, da cui è poi possibile convertirlo in diversi altri linguaggi, come verrà esposto nel capitolo 2.4.

Le richieste sono semplicemente composte da coppie attributo-valore. L'idea è che almeno parte di questi attributi siano predefiniti e strettamente collegati con l'utente che sta eseguendo la richiesta, mentre altri possano essere impostati dall'utente in base a cosa intende fare. In questo listing si può vedere un esempio di richiesta in linguaggio FACPL:

Listing 2: Esempio di richiesta in FACPL

```
1 Request: { Online_Generated
2   (action/action-id, "CREATE")
3   (subject/profile-id, "P_2")
4   (resource/vm-type, "4")
5 }
```

2.3 COMPONENTI DEL SISTEMA FACPL

La Figura 1 mostra le componenti del processo di valutazione delle policy del sistema FACPL:

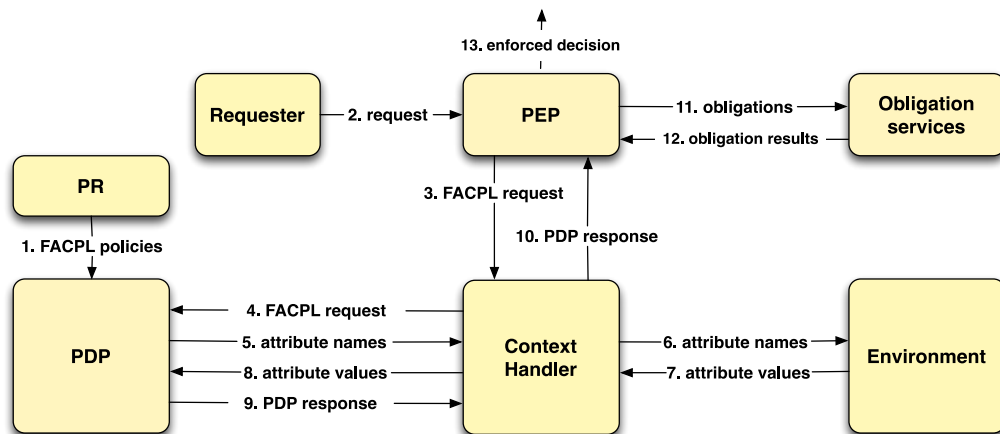


Figura 1: Processo di valutazione di FACPL

- *PEP*: Policy Enforcement Point, è il componente che si occupa di verificare la compatibilità della richiesta con le policy. Nessuna operazione di valutazione è svolta in questo componente, infatti come si può vedere in Figura 1 la richiesta viene inviata al *PDP* per essere valutata e questo restituisce il risultato da usare nel *PEP*. Per

l'esecuzione delle obbligazioni ci si affida agli *Obligation services*. In FACPL il *PEP* è però anche portato a svolgere una scelta in situazioni in cui il *PDP* non è in grado di fornire una risposta;

- *PDP*: Policy Decision Point, è il componente che riceve le richieste dal *PEP* e le valuta in base alle policy presenti nel *PR*. Il risultato della valutazione è restituito al *PEP*;
- *PR*: Policy Repository, è il componente che si occupa di memorizzare le policy e di fornirle al Policy Decision Point;
- *Context Handler*: come si può vedere è in mezzo nel percorso fra *PEP* e *PDP*, e sia le richieste che risposte passano sempre da questo componente. Il suo compito è quello di fornire il contesto necessario per valutare le richieste, infatti come si può vedere è strettamente legato ad *Environment* a cui chiede i valori degli attributi necessari per la valutazione;
- *Environment*: questo componente non è definito in modo specifico, deve infatti essere implementato in base al sistema in uso ed ha come compito quello di fornire valori attuali di specifici attributi;
- *Requester*: è il componente che invia le effettive richieste al *PEP*;
- *Obligation services*: è il componente che attua le obbligazioni e verifica l'esito dell'esecuzione. La decisione del *PEP* dipende dal risultato dell'esecuzione delle obbligazioni; se queste non sono state eseguite con successo la richiesta non dà esito positivo.

2.4 MODALITÀ DI UTILIZZO DI FACPL

FACPL viene distribuito come una repository p2 di Eclipse; questo permette di installarlo direttamente tramite la UI di Eclipse in modo molto facile. FACPL richiede l'utilizzo di Java 8 anche se non risulterebbe difficile integrarlo in progetti che utilizzano versioni di Java successive con qualche accorgimento.

Il metodo principale per sviluppare un progetto FACPL è crearlo direttamente con l'interfaccia di Eclipse, in questo modo viene creato un progetto con tutte le dipendenze necessarie. Si nota come FACPL sia pensato per essere usato tramite la sua sintassi di alto livello e non tramite Java direttamente, di conseguenza è di solito richiesto al programmatore di scrivere codice Java solo per tre componenti del sistema, il *Context*

Handler, l'*Environment* e gli *Obligation services*.

Il *Context Handler* deve soltanto essere leggermente modificato per adattarsi ad interagire con l'*Environment*, che invece è la parte fondamentale da scrivere. L'*Environment* può essere scritto come si preferisce, basta che sia in grado di comunicare con il *Context Handler* quindi può potenzialmente essere scritto in qualsiasi linguaggio di programmazione. Questa logica rende FACPL facilmente adattabile ad un grande numero di casi concreti. Per quanto riguarda gli *Obligation services* occorre integrare quelle che sono chiamate *PEPAction*, ovvero i comandi che vengono effettivamente lanciati come obbligazioni, in questo caso il contratto da rispettare è che siano tutte classi che implementano l'interfaccia *IPepAction* e quindi che presentino il metodo *evaluate* con cui vengono chiamate dal *PEP*.

Per la scrittura di file *.fpl* in sintassi FACPL è fornito un editor apposito realizzato con Xtext [3] ed accessibile comodamente da Eclipse automaticamente dopo l'installazione della repository p2. L'editor presenta alcune funzioni tipiche degli IDE come l'identificazione di *warning* e *errors* che permettono di capire se si sono verificati degli errori di sintassi, oltre che l'highlighting della sintassi. Vengono messi inoltre a disposizione dei generatori automatici di codice Java, XACML e SMT-LIB a partire da un file *.fpl*; questi generatori sono accessibili dalla UI di Eclipse. Nella repository del progetto su github² sono inoltre presenti diversi codici di esempio.

2.5 IL SOFTWARE OPENNEBULA

OpenNebula [2] è definita come una piattaforma open-source potente, ma semplice da utilizzare, che si può usare per costruire e gestire infrastrutture Cloud. L'idea dietro OpenNebula è quella di fornire una struttura per la gestione unificata per le infrastrutture IT e le applicazioni, evitando di dover utilizzare strumenti closed-source e proprietari per cui di solito è richiesto un alto costo di acquisto e gestione.

OpenNebula permette di gestire diversi hypervisor come KVM, VMware, Xen e LXDM, consentendo quindi di creare diversi tipi di virtual machine e container e permette anche di scegliere diverse modalità per la gestione dello storage.

² <https://github.com/andreamargheri/FACPL>

2.6 ARCHITETTURA DI OPENNEBULA

OpenNebula è una piattaforma che si compone di diversi componenti, ognuno dei quali svolge un ruolo ben preciso all'interno del sistema. I componenti sono presentati tutti all'interno della documentazione ufficiale³, in questo capitolo verranno presentati solo quelli fondamentali per capire il funzionamento di OpenNebula in un caso di studio come quello descritto in questa tesi:

- *OpenNebula Daemon*: Questo è il servizio centrale della piattaforma cloud. Gestisce i nodi, le reti, lo storage, i gruppi e gli utenti. Questo servizio è responsabile di coordinare tutte le operazioni e risponde alle chiamate XML-RPC fatte con le apposite API. Solitamente questo servizio è eseguito su un nodo che può anche essere usato come host.
- *Host*: Ogni host rappresenta un server che è gestito da OpenNebula e può essere utilizzato per eseguire delle virtual machine. Ogni host deve installare tutte le dipendenze necessarie e deve successivamente essere registrato in OpenNebula. Ogni host può presentare un solo hypervisor, anche se ci sono modi per cercare di aggirare questa limitazione.
- *User*: OpenNebula permette di gestire utenti e gruppi in modo del tutto separato dal modo in cui sono gestiti in Unix. Ogni utente ha un *ID* univoco e può appartenere ad uno o più gruppi. Di default viene creato l'utente `oneadmin` che ha tutti i permessi e può creare nuovi utenti e gruppi. Gli utenti del gruppo `oneadmin` hanno gli stessi permessi dell'utente `oneadmin` mentre gli utenti creati e inseriti in altri gruppi possono avere diversi permessi.
- *Virtual Machine Template*: In OpenNebula le virtual machine sono definite a partire da un template, che contiene tutte le informazioni necessarie per crearle, come capacità di memoria, CPU e dischi da utilizzare. I template possono essere definiti a partire da zero, usando l'appropriata sintassi ma sono disponibili anche presso degli store online e possono essere modificati a piacimento. Conoscere il template con cui è stata creata una virtual machine fornisce moltissime informazioni su come è configurata. La UI e la CLI forniscono strumenti per verificare la correttezza di un template

³ https://docs.opennebula.io/6.8/overview/opennebula_concepts/opennebula_overview.html#idl

che si intende salvare.

- *Sunstone*: OpenNebula fornisce diverse interfacce per la gestione delle risorse; quella più user-friendly è sicuramente Sunstone ovvero una Web-UI che è utilizzabile sia da utenti che da supervisori. Quando viene fatta l'installazione di OpenNebula, Sunstone è installata automaticamente ma è gestita da un demone separato, di conseguenza può essere spenta se non utilizzata. L'alternativa a Sunstone è utilizzare la CLI, che per alcuni comandi rimane comunque più veloce.

2.7 XML-RPC E OPENNEBULA

Il protocollo XML-RPC [16] è un protocollo che permette di eseguire chiamate a procedure remote (*Remote Procedure Call*) attraverso la rete. Come indicato dal nome, utilizza lo standard XML per la codifica della richiesta e si basa su HTTP per il trasporto dei dati.

L'idea dietro questo protocollo è di rimanere il più semplice possibile permettendo però anche di gestire strutture dati complesse. Il protocollo è stato sviluppato da Dave Winer e Microsoft nel 1998 e da allora è stato largamente utilizzato ma anche mantenuto e migliorato. A partire dal 2019 il protocollo ha una nuova implementazione in JavaScript che permette anche l'utilizzo di sintassi Json oltre che XML.

OpenNebula utilizza XML-RPC per mettere in comunicazione i componenti del sistema e fornire delle API, come si può vedere direttamente nella documentazione ufficiale⁴. Per eseguire una chiamata occorre essere autenticati e avere i permessi necessari. Quindi la prima cosa che viene fatta quando si esegue una richiesta XML-RPC è autenticare il token; a quel punto è presente un gestore delle richieste che crea una richiesta di autorizzazione per una o più operazioni. Grazie a questa logica è possibile sviluppare delle API wrapper per ogni linguaggio si ritenga necessario. Alcune di queste, come quelle per Java⁵ sono disponibili direttamente sulla documentazione di OpenNebula e sono mantenute costantemente dalla community.

4 https://docs.opennebula.io/6.8/integration_and_development/system_interfaces/api.html

5 https://docs.opennebula.io/6.8/integration_and_development/system_interfaces/java.html

INTEGRAZIONE DI OPENNEBULA CON FACPL

Come già discusso nel capitolo 2, grazie al modo in cui FACPL [8] è costruito risulta molto facile andare a fornire un'implementazione concreta dei PEP (*Policy Enforcement Point*) e PDP (*Policy Decision Point*) avendo conoscenza di quali sono le esigenze a cui devono rispondere.

Per validare ulteriormente questo punto abbiamo deciso di utilizzare FACPL in una situazione in cui fosse necessario interfacciarsi con un software già esistente così da mostrare le effettive potenzialità e semplicità di implementazione della libreria. Il caso concreto che abbiamo deciso di considerare deriva da un possibile sviluppo futuro già proposto in [10], ovvero un'integrazione con un sistema di IaaS open-source sul cloud come OpenNebula[2], software che è stato già presentato nel capitolo 2.

3.1 IDEA DI BASE

Per iniziare abbiamo considerato due esempi di gestione delle risorse disponibili aderenti alla realtà, presentati in [10]. Abbiamo inoltre potuto osservare due primi tentativi di implementazione descritti sempre in [10]:

- il primo sviluppato basandosi su una completa astrazione, ovvero dei files che simulano un sistema con diverse virtual machine, che è presente anche in [9] fra gli esempi nella cartella *EXAMPLES*;
- il secondo basato su uno XEN Hypervisor, che però non è presente fra gli esempi forniti assieme alla libreria e di cui non vengono esplicitati i dettagli di implementazione.

Con questa conoscenza l'obiettivo iniziale è stato quello di riuscire ad interagire con un sistema reale su cui è installato un cloud manager per fare sì che questo esegua i comandi necessari per effettuare le PEP action

da noi richieste e ci esponga tutte le informazioni necessarie al PDP per valutare le richieste. Nel concreto era quindi necessario pensare a due parti distinte:

- una che svolgesse operazioni sulle singole virtual machine (avviarle, inserirle nell'host corretto, fermarne l'esecuzione);
- una che recuperasse le informazioni generali sugli host e sul sistema tutto.

Il cloud manager che abbiamo deciso di utilizzare è OpenNebula per i motivi presentati nel capitolo 2.

3.2 OPENNEBULA API

Il primo approccio che avevamo pensato di percorrere era quello di utilizzare il codice Java per invocare dei comandi da shell che fossero in grado di interagire con OpenNebula. Questo approccio sembrava il più immediato anche dato lo studio preliminare di OpenNebula che avevamo svolto che era stato soprattutto attraverso la shell (oltre che la Web-UI).

Per utilizzare i comandi da shell che permettono di interagire con OpenNebula occorre aver effettuato l'autenticazione come utente OpenNebula¹. Questa soluzione aveva il principale vantaggio di poter fornire un'interfaccia generica che permetteva in un futuro di far interagire con sforzo minimo FACPL anche con comandi di natura completamente diversa, tuttavia presentava anche diverse problematiche come la forte dipendenza dalla versione di OpenNebula installata nel sistema e una grande inefficienza nello svolgimento di alcune operazioni, oltre che alcune difficoltà in fase di test.

La strada su cui ci siamo orientati è stata quindi quella di utilizzare le API di OpenNebula per Java² in quanto queste semplificavano l'ottenimento di alcune informazioni. Come è possibile leggere dal sito di OpenNebula stesso, queste API sono a loro volta un wrapper dei metodi XML-RPC³. Le API utilizzate sono quelle della versione 5.12.0⁴ dato che

1 https://docs.opennebula.io/6.8/management_and_operations/users_groups_management/manage_users.html

2 https://docs.opennebula.io/6.8/integration_and_development/system_interfaces/java.html

3 https://docs.opennebula.io/6.8/integration_and_development/system_interfaces/api.html#api

4 <https://downloads.opennebula.io/packages/opennebula-5.12.0/>

dopo la major version 5 sono compilate con la versione di Java 11 (al contrario di quanto riportato sul sito stesso) e di conseguenza non erano compatibili con la versione di Java 8 con cui è stato sviluppato FACPL. I principali attori che sono stati utilizzati dalle API⁵ sono stati:

- *Client*: è la classe principale che gestisce la connessione fra il core di OpenNebula e le chiamate XML-RPC, quasi tutti gli altri oggetti delle API richiedono di passare un oggetto di questo tipo per poter essere istanziati. Questo oggetto può essere istanziato passando uno *username* e una *password* al costruttore, ma presenta anche un costruttore che non richiede parametri e permette di derivare *username password* dall'utente corrente.
- *ClientConfigurationException*: è la classe che rappresenta l'eccezione che viene lanciata se si tenta di istanziare un *Client* con delle impostazioni di autorizzazione sbagliate, in particolare se *username password* sono errate oppure se si tenta di usare il costruttore vuoto lanciando il programma da un utente che non è uno user di OpenNebula.
- *OneResponse*: è la classe che incapsula le risposta XML-RPC di OpenNebula, viene istanziata con un *boolean* e una *String* che rappresentano rispettivamente l'esito della richiesta (positivo o negativo) e un eventuale messaggio che lo descrive. Quasi tutte le azioni eseguibili sui *PoolElement* ritornano un oggetto di questo tipo.
- *Pool*: è la classe che rappresenta un insieme di *PoolElement* e fornisce la possibilità di scorrere gli stessi ed eseguire in modo agevolato alcune operazioni
- *PoolElement*: è la superclasse della maggior parte delle classi che rappresentano gli elementi di OpenNebula, quelli più interessati nel progetto sono stati
 - *VirtualMachine*
 - *Host*
 - *Template*

⁵ <https://docs.opennebula.io/doc/6.4/oca/java/org/opennebula/client/package-summary.html>

3.3 LOGGING

Prima ancora di pensare ai comandi da eseguire sulle virtual machine si è reso necessario pensare ad una modalità di logging. Dato l'ambito di applicazione si è resa fondamentale la scrittura di logs su file di modo da rendere gli stessi facilmente accessibili in futuro, anche a distanza di tempo.

All'interno del progetto è quindi fornita una classe `FileLoggerFactory` che utilizza il design pattern *Static Factory Methods* [1] e permette di creare dei file di log, di modo da evitare la necessità di interagire coi file in ogni classe che intende eseguire il log di informazioni oltre che gestire in modo consono gli handler dei file evitando duplicati. Questa classe permette di creare dei `Logger` con un'implementazione di `Level` e `Formatter` di default oppure di specificare questi parametri in input come mostra il seguente listing:

Listing 3: Metodo `make` di `FileLoggerFactory`

```

1 public static Logger make(String fileName, Formatter formatter, Level
   level) {
2     Throwable t = new Throwable();
3     StackTraceElement directCaller = t.getStackTrace()[1];
4     String loggerName = directCaller.getClassName() + "-" + fileName;
5
6     Logger logger = Logger.getLogger(loggerName);
7
8     if (!isFileHandlerAttached(logger, fileName)) {
9         try {
10             FileHandler fileHandler = createFileHandler(fileName,
11                 formatter, level);
12             logger.addHandler(fileHandler);
13             logger.setUseParentHandlers(false);
14         } catch (IOException e) {
15             throw new RuntimeException("Failed to initialize logger
16                 handler.", e);
17         }
18     }
19     return logger;
20 }

```

Nonostante la presenza di questa classe, tutte le classi all'interno del progetto hanno i logger passati tramite dependency injection e forniscono un'implementazione di default che esegue logging nello standard out-

put (solitamente la console). L'unica classe che fa eccezione in questo è proprio `ContextStub_Default` che è il primo punto di ingresso della dipendenza.

L'idea è che nel caso in cui si preferisca eseguire logging in modo diverso si possa definire un logger diverso al posto dell'implementazione proposta. Per farlo occorre semplicemente modificare una riga all'interno del costruttore di `ContextStub_Default`:

Listing 4: Costruttore `ContextStub_Default`

```
1 public static ContextStub_Default getInstance() {
2     if (instance == null) {
3         try {
4             Configuration config = new
3             Configurations().properties(CONFIG_FILE);
5             hyper1HostId = config.getString("hyper1.host.id");
6             hyper2HostId = config.getString("hyper2.host.id");
7             ContextStub_Default.oneClient = new Client();
8             Logger logger =
3             FileLoggerFactory.make("logs/virtualMachines.log");
                initializeStub(oneClient, logger);
9             instance = new ContextStub_Default();
10        } catch (ClientConfigurationException e) {
11            throw new RuntimeException("Failed to initialize Client: " +
                e.getMessage(), e);
12        } catch (ConfigurationException e) {
13            throw new RuntimeException(
14                "Errors in the config gile: " + e.getMessage(), e);
15        } catch (Exception e) {
16            throw new RuntimeException(
17                "Unexpected error during ContextStub_Default
                initialization: " + e.getMessage(), e);
18        }
19    }
20    return instance;
21 }
```

3.4 GESTIONE DELLE VIRTUAL MACHINE GENERICHE

Per gestire le virtual machine l'approccio iniziale è stato quello di fornire un'interfaccia che permettesse in un futuro di poter interagire con le stesse anche in modo diverso. E' stata creata una classe wrapper chiamata

VMDescriptor che contiene le informazioni sulle virtual machine utili per il nostro progetto. Questa classe serve per creare una prima astrazione dalle API utilizzate, in effetti questa stessa classe permetterebbe, ad esempio, di fornire un'implementazione come quella precedentemente pensata basata sui comandi da shell. L'interfaccia VirtualMachineService presenta due metodi che servono per lavorare con degli oggetti di tipo VMDescriptor.

Listing 5: VirtualMachineService

```
1 public interface VirtualMachineService {  
2     List<VMDescriptor> getVirtualMachinesInfo();  
3     List<VMDescriptor> getRunningVirtualMachineInfo();  
4 }
```

La scelta di avere due metodi separati per restituire tutte le macchine virtuali oppure solo quelle attualmente in esecuzione deriva dal fatto che nella pratica spesso queste due liste vorranno essere utilizzate in modo distinto. Di solito le VirtualMachine presenti nel sistema sono molte più di quelle effettivamente in esecuzione e quindi non inserire il secondo metodo avrebbe portato una buona parte delle classi che volevano utilizzare un'implementazione di questa interfaccia, a dover sempre inserire un controllo sullo stato delle virtual machine.

Nelle altre classi da me scritte viene utilizzata l'implementazione concreta di VirtualMachineService chiamata OpenNebulaVMService che, interfacciandosi con le API di OpenNebula già descritte, riesce ad ottenere tutte le informazioni richieste sulle virtual machine e a popolare le due liste.

Una volta ottenuta una lista di VMDescriptor è possibile filtrare le virtual machine sia a partire dal nome che a partire da altre loro caratteristiche come l'ID o il *template* utilizzato per crearle come spiegato nel capitolo 2. È stata implementata un'ulteriore classe di comodo che mette a disposizione la possibilità di eseguire alcuni tipi di filtraggio sulle virtual machine in automatico. Questa classe fornisce inoltre un meccanismo di logging ogni volta che viene eseguita un'operazione di filtraggio, come si può vedere nel codice di esempio:

Listing 6: VMsInfo logging e filtro

```
1 public class VMsInfo {  
2     ...  
3     private List<VMDescriptor> getAndLogVMs() {  
4         List<VMDescriptor> vmDescriptors =
```



```

        vmService.getRunningVirtualMachineInfo();
5     logger.info("The VMs running are: " + vmDescriptors.toString());
6     return vmDescriptors;
7 }
8
9 public List<VMDescriptor> getRunningVMsByHostTemplate(String host, String
    templateId) {
10     return getAndLogVMs().stream()
11         .filter(x -> x.getHostId().equals(host) &&
            x.getTemplateId().equals(templateId))
12         .collect(Collectors.toList());
13 }
14 ...
15 }

```

Non è necessario usare questa classe, però risulta comoda per fare sì che le classi che si occupano di eseguire un comando su una o più virtual machine non abbiano anche il compito di eseguire il filtraggio e fare logging.

3.5 GESTIONE DELLE VIRTUAL MACHINE DI OPENNEBULA

Da qui in avanti saranno presentate tutte le classi che interagiscono effettivamente con degli oggetti che rappresentano delle `VirtualMachine` come indicato nelle API di OpenNebula. La prima classe implementata è `OpenNebulaActionContext`; questa classe può essere istanziata usando un `Client` (e opzionalmente anche un `Logger`) e fornisce semplicemente uno stato da utilizzare per eseguire i comandi che richiedono informazioni sulle `VirtualMachine` attualmente in esecuzione, che in questo caso concreto saranno tutti tranne la creazione di una nuova virtual machine. La classe per eseguire i comandi ha la seguente base:

Listing 7: Classe astratta per i comandi

```

1 public abstract class OpenNebulaActionBase implements IPepAction{
2     protected final OpenNebulaActionContext ONActionContext;
3
4     public OpenNebulaActionBase(OpenNebulaActionContext
        ONActionContext) {
5         this.ONActionContext = ONActionContext;
6     }
7
8     public abstract void eval(List<Object> args);
9
10    protected void logResponse(OneResponse response) {
11        if (response.isError()) {

```

```

12     ONActionContext.getLogger().severe(response.getErrorMessage());
13 } else {
14     ONActionContext.getLogger().info(response.getMessage());
15 }
16 }
17 }

```

Per questa classe è stato valutato l'utilizzo di un Template method, tuttavia risultava abbastanza scomodo da applicare dato che alcune delle classi concrete potrebbero dover seguire un workflow diverso fra loro (es. sospensione di una virtual machine e creazione di una virtual machine) per il modo in cui le API di OpenNebula sono scritte. Inoltre si suppone la possibilità di scrivere comandi che agiscano su più di una virtual machine in un solo comando. Questa logica si è resa necessaria per poter utilizzare alcune policy.

La scelta del nome `eval` è obbligata dal modo in cui FACPL accederà alla classe per eseguire i comandi. L'approccio scelto è quello di costruire l'oggetto di tipo `VirtualMachine` corrispondente all'*ID* ottenibile dal `VMdescriptor`. Questo passaggio può sembrare controintuitivo perchè partendo da un oggetto `VirtualMachine` si ottengono le sue informazioni per poi andare a ricreare un oggetto sostanzialmente identico per eseguire le operazioni, tuttavia questi passaggi hanno diversi lati positivi:

- rendono il codice indipendente dalla modalità con cui si ottengono le informazioni sulle virtual machine;
- rendono il codice più aperto a modifiche e future implementazioni;
- rendono il codice molto più semplice da testare;
- isolano l'ottenimento delle informazioni ad una classe che estende `OpenNebulaVMService`.

Le classi concrete per eseguire i comandi sulle virtual machine a partire da questa classe hanno tutte chiaramente una forma simile sebbene con alcuni accorgimenti.

Listing 8: Classe per avviare una `VirtualMachine`

```

1 public class CreateVM extends OpenNebulaActionBase {
2     public CreateVM(OpenNebulaActionContext ONActionContext) {
3         super(ONActionContext);
4     }
5
6     public void eval(List<Object> args) {
7         ONActionContext.getLogger().info("Starting VM: " + "[" + args.get(2) + ", " +
            args.get(1) + "]");

```

```

8      Template template = new Template((int) args.get(2),
9          ONActionContext.getClient());
10     OneResponse instantiateResponse = template.instantiate((String) args.get(1));
11     logResponse(instantiateResponse);
12     if (!instantiateResponse.isError()){
13         VirtualMachine vm =
14             new VirtualMachine(instantiateResponse.getIntMessage(),
15                 ONActionContext.getClient());
16         logResponse(vm.deploy((int) args.get(0)));
17     }

```

La classe CreateVM, utilizza un oggetto di tipo Template, che, come già anticipato all'interno del capitolo 2 permette di definire le caratteristiche per la creazione di una specifica virtual machine. Nel nostro caso concreto l'oggetto template risulta utilissimo per fare sì che la definizione delle caratteristiche delle virtual machine da utilizzare sia fatta attraverso la UI di OpenNebula (o comunque con dei file appositi validati da OpenNebula tramite UI o comando da shell).

Nel caso di studio in [10] vengono considerati due tipi di virtual machine istanziabili; quindi nei nostri test abbiamo spesso considerato due template che fossero in grado di riprodurre i comportamenti richiesti. Tuttavia grazie al Template si apre la strada all'utilizzo di virtual machine dalle caratteristiche più disparate senza neanche bisogno di cambiare il codice Java. Infatti grazie al modo in cui è scritto il codice basta definire in OpenNebula un nuovo template e utilizzare il suo ID all'interno di policy e richieste FACPL per poter creare (o distruggere, frezzare ecc.) delle virtual machine di quel tipo.

Listing 9: Classe per freezare(sospendere) una VirtualMachine

```

1 public class FreezeVM extends OpenNebulaActionBase {
2     public FreezeVM(OpenNebulaActionContext ONActionContext) {
3         super(ONActionContext);
4     }
5
6     public void eval(List<Object> args) {
7         ONActionContext.getLogger().info("Stopping (Freezing) 1 VM of [host,
8             template]: " + "[" + args.get(0) + " " + args.get(2) + "]");
9         List<VMDescriptor> suspendList =
10             ONActionContext.getVMsInfo()
11                 .getRunningVMsByHostTemplate((String)args.get(0),
12                     (String)args.get(2));
13         if (suspendList.isEmpty()) {
14             ONActionContext.getLogger().severe("No VM found");
15             return;
16         }
17         logResponse(
18             new VirtualMachine(Integer.parseInt(suspendList.get(0).getVmId()),
19                 ONActionContext.getClient())

```

```

17         .stop();
18     }
19 }

```

La classe `suspendVM` raffigura la struttura che hanno tutti i comandi che agiscono sulle virtual machine attualmente in esecuzione:

1. viene recuperata la lista delle virtual machine attualmente in esecuzione;
2. viene filtrata la lista per trovare le virtual machine che rispettano i criteri richiesti;
3. viene verificato che la lista non sia vuota;
4. viene creato un (o più) oggetto di tipo `VirtualMachine` su cui eseguire in effettivo il comando;
5. viene eseguito il comando.

Queste operazioni sono sempre eseguite con appropriato logging a contorno e in futuro lasciano spazio alla possibilità di eseguire filtraggio in modo diverso o di utilizzare più di una virtual machine presente nella lista, se necessario.

3.6 INTERAZIONE CON IL CONTEXTSTUB E LE PEACTIONS

La gestione della classe `ContextStub_Default` è definita dall'implementazione in Java di FACPL; la classe è stata quindi soltanto modificata affinché potesse recuperare le informazioni necessarie ad eseguire le valutazioni sullo stato del sistema, in particolare sono state aggiunte le seguenti variabili di sistema:

Listing 10: Context di OpenNebula

```

1 @Override
2 public Object getContextValues(AttributeName attribute) {
3     ...
4     if (attribute.getCategory().equals("system") &&
5         attribute.getIDAttribute().equals("vm-name")) {
6         return UUID.randomUUID().toString();
7     }
8     if (attribute.getCategory().equals("system") &&
9         attribute.getIDAttribute().equals("hyper1.vm-names")) {
10        Set runningHyper1VMs = new Set();
11        vmsInfo.getRunningVMsByHost(hyper1HostId).forEach(vm ->
12            runningHyper1VMs.addValue(vm.getVmName()));

```

```

11     return runningHyper1VMs;
12 }
13 if (attribute.getCategory().equals("system") &&
    attribute.getIDAttribute().equals("hyper2.vm-names")) {
14     Set runningHyper2VMs = new Set();
15     vmsInfo.getRunningVMsByHost(hyper2HostId).forEach(vm ->
        runningHyper2VMs.addValue(vm.getVmName()));
16     return runningHyper2VMs;
17 }
18 if (attribute.getCategory().equals("system") &&
    attribute.getIDAttribute().equals("hyper1.vm1-counter")) {
19     return vmsInfo.getRunningVMsByHostTemplate(hyper1HostId,
        type1Template).size();
20 }
21 if (attribute.getCategory().equals("system") &&
    attribute.getIDAttribute().equals("hyper2.vm1-counter")) {
22     return vmsInfo.getRunningVMsByHostTemplate(hyper2HostId,
        type1Template).size();
23 }
24 if (attribute.getCategory().equals("system")
    && attribute.getIDAttribute().equals("hyper1.availableResources")) {
25     return hostInfo.getAvailableCpu(hyper1HostId);
26 }
27 }
28 if (attribute.getCategory().equals("system")
    && attribute.getIDAttribute().equals("hyper2.availableResources")) {
29     return hostInfo.getAvailableCpu(hyper2HostId);
30 }
31 }
32 return null;
33 }

```

Guardando questo snippet notiamo alcune caratteristiche da evidenziare:

- `UUID.randomUUID().toString()` è un metodo che permette di ottenere un identificativo unico che verrà usato come nome per le virtual machine. In OpenNebula le virtual machine non sono istanziabili a partire da un ID, dato che questo viene assegnato dal software alla creazione, tuttavia è possibile assegnare un nome ad una virtual machine e quindi quello che abbiamo deciso di fare è usare il nome come identificativo all'interno del nostro progetto. Questa logica permette di assegnare degli identificativi più significativi in un futuro se fosse necessario e permette di disaccoppiare la nostra logica dalla logica con cui OpenNebula associa gli ID.
- `Set` è una classe fornita da FACPL per Java che permette di creare un insieme di oggetti; l'utilizzo di un oggetto di questo tipo è obbligatorio per usare i metodi di confronto presenti nella libreria. Usare un oggetto della classe `Set` incluso nelle `Collections` causerà un errore nell'analisi delle Policy.
- `hostInfo` è un oggetto della classe `HostInfo`; questa non è stata

presentata ma, come si può notare dal codice, è semplicemente una classe che permette di ottenere informazioni riguardo CPU e quantità di memoria disponibili.

- Come si può notare ci sono diverse parti di codice in cui si fa riferimento a `hyper1HostId` e `hyper2HostId`; questi due parametri sono letti direttamente dal file `config.properties` locato nella directory del progetto grazie all'utilizzo di alcuni metodi forniti all'interno della libreria *Apache Commons*[15], come si può vedere guardando il codice del costruttore (listing 4).

Per quanto riguarda le `PEPAction`, anche in questo caso basandosi sulla struttura fornita dalla libreria FACPL non ci sono molte scelte implementative che si possono fare e il risultato è il seguente:

Listing 11: Classe `PEPAction` adattata

```

1 public class PEPAction{
2     public static HashMap<String, IPepAction> getPepActions() {
3         ContextStub_Default.getInstance();
4         HashMap<String, IPepAction> pepAction = new HashMap<String,
5             IPepAction>();
6         pepAction.put("release",
7             new ReleaseVM(ContextStub_Default.getONContext()));
8         pepAction.put("create",
9             new CreateVM(ContextStub_Default.getONContext()));
10        pepAction.put("freeze",
11            new FreezeVM(ContextStub_Default.getONContext()));
12        return pepAction;
13    }
14 }
```

3.7 ACCESSO DALL'ESTERNO DEL PROGETTO

Per il modo in cui il progetto è scritto ci sono diversi punti di entrata da cui una persona esterna può iniziare a sviluppare codice che sfrutti le classi sopra discusse, alcuni dei quali sono anche stati già esposti durante la presentazione delle classi stesse. Il progetto è distribuito con due package contenenti le due implementazioni concrete delle tecniche di gestione presentate in [10] oltre che con il codice FACPL che le ha generate. Di conseguenza aprendo il progetto con Eclipse si può facilmente cominciare a generare nuove richieste e trasformarle in codice Java tramite la UI di

Eclipse⁶. Inoltre è anche fornita la cartella */opennebula_context_actions* che contiene i file java delle classi *PEPAction* e *ContextStub_Default*.

Nonostante questo si è pensato di fornire delle classi che permettessero, dato un file FACPL, di: validarlo, generare il codice Java corrispondente, compilarlo e, nel caso lo si voglia, anche di eseguire direttamente il main di *MainFACPL*. Il motivo principale per cui le classi che saranno discusse in questo paragrafo sono state ideate è che all'interno della documentazione di FACPL non è mai esplicitato un workflow da seguire per poter eseguire a runtime la decisione delle policy e/o la generazione di una nuova richiesta; di conseguenza si è reso utile idearne uno.

Listing 12: Classe *FACPLHandlingTemplate*

```

1 public abstract class FACPLHandlingTemplate {
2
3     private final String CONFIG_FILE = "config.properties";
4     protected Logger logger;
5     protected CodeExecutorInterface executor;
6     protected String javaFilesDir;
7     protected ClassSetup setupper;
8
9     public FACPLHandlingTemplate(String logFilePath, String javaFilesDir) throws
        IOException {
10         this.logger = FileLoggerFactory.make(logFilePath);
11         this.javaFilesDir = javaFilesDir;
12     }
13
14     public FACPLHandlingTemplate(Logger logger, String javaFilesDir) throws
        IOException {
15         this.logger = logger;
16         this.javaFilesDir = javaFilesDir;
17     }
18
19     public final void execute(String[] args) throws Exception {
20         try {
21             List<String> fileLocations = Arrays.asList(args[0]);
22             initializeConcreteSetupperExecutor(fileLocations);
23             setup();
24             compile();
25             postProcess();
26         } catch (Exception e) {
27             logger.severe("An error occurred: " + e.getMessage());
28             throw e;
29         }
30     }
31
32     protected abstract void initializeConcreteSetupperExecutor(List<String>
        fileLocations) throws Exception;
33
34     protected void setup() throws Exception {
35         setupper.setup(new Configurations())

```

⁶ <https://facpl.readthedocs.io/en/latest/txt/facpl.html>

```

36         .properties(CONFIG_FILE).getString("context.file.location"), javaFilesDir);
37     }
38
39     protected void compile() throws Exception {
40         boolean success = executor.compileJavaFiles();
41         if (success) {
42             logger.info("Compilation successful.");
43         } else {
44             logger.severe("Compilation failed.");
45             throw new RuntimeException("Compilation failed");
46         }
47     }
48
49     protected abstract void postProcess() throws Exception;
50 }

```

La classe base che è stata scritta è `FACPLHandlingTemplate`; come si può intuire dal nome e dalla struttura della classe, è una rappresentazione del pattern *Template Method* [4]. Questa classe permette di istanziare degli oggetti a partire da una locazione dove verranno inseriti e successivamente compilati, i file Java. Lanciando il comando `execute` infatti quello che succede effettivamente è:

1. Vengono inizializzati concretamente gli oggetti che servono per compilare, eseguire e posizionare i file Java;
2. i file Java prodotti a partire dai file FACPL vengono messi tutti nella cartella di destinazione finale, che è determinata dal parametro con cui viene eseguito il metodo `execute`;
3. i file aggiuntivi (solitamente `PEPAction` e `ContextStub_Default`) vengono messi nella cartella finale;
4. i file vengono compilati;
5. i file compilati possono essere eseguiti o, più in generale, utilizzati per qualunque scopo si voglia definire.

Questi passaggi all'apparenza molto semplici in realtà nella loro implementazione concreta necessitano di diversi passaggi intermedi.

Le implementazioni concrete di questa classe astratta che vengono fornite sono `ApplyPolicy` e `RequestExecution`, che servono rispettivamente per applicare una policy al sistema e eseguire la valutazione di una richiesta, ed entrambe utilizzano come `setter` per la classe `OpenNebulaFACPLClassSetup` che presenta il seguente metodo `setup`:

Listing 13: Metodo di setup per `OpenNebula`

```

1 @Override

```



```
2 public void setup(String additionalFilesFolder, String outputFolder) {
3     logger.info("Starting setup...");
4
5     try {
6         classGenerator.generateClasses("tmp/FACPLFiles");
7         logger.info("Class generation completed successfully.");
8     } catch (Exception e) {
9         logger.severe("Failed to generate classes: " + e.getMessage());
10        e.printStackTrace();
11        return;
12    }
13
14    try {
15        FolderContentHandler folderManager = new
16            FolderContentHandler(logger);
17        folderManager.processFolderContents("tmp/FACPLFiles/",
18            outputFolder, new MoveStrategy());
19        folderManager.processFolderContents(additionalFilesFolder,
20            outputFolder, new CopyStrategy());
21        logger.info("Folder contents handled successfully.");
22    } catch (IOException e) {
23        logger.severe("Failed to move folder contents: " +
24            e.getMessage());
25        e.printStackTrace();
26    }
27 }
```

In questo caso si evidenziano due punti necessari di ulteriori spiegazioni:

- `classGenerator` è un oggetto creato a partire dalla classe `OpenNebulaFACPLClassGenerator` che sfrutta la libreria di FACPL e il generatore fornito come esempio dalla stessa, per generare tutte le classi Java necessarie;
- `FolderContentHandler` è una classe che grazie ad uno strategy [4] applicato ad un metodo, permette di definire diversi modi per gestire i contenuti di due cartelle.

Listing 14: Metodo `processFolderContent`

```
1 public void processFolderContents(String sourceDir, String targetDir,
2     FolderContentStrategy strategy) throws IOException {
3     Path sourcePath = Paths.get(sourceDir);
4     Path targetPath = Paths.get(targetDir);
5     processFolderContentsInternal(sourcePath, targetPath, strategy);
6 }
```

Come si può notare il suo metodo `processFolderContents` isola inoltre le classi che la utilizzano dalla logica dei Path, usando-

li internamente ma chiedendo soltanto stringhe nei suoi metodi pubblici.

Listing 15: Interfaccia FolderContentStrategy

```
1 interface FolderContentStrategy {
2     void processFile(String source, String target) throws IOException;
3     String getOperationName();
4 }
```

L'interfaccia dello strategy è semplice; oltre all'operazione di base che si vuole che sia in grado di eseguire, basta specificare un `operationName` così da sapere che operazione viene svolta dal metodo che l'ha invocata, se questo è interessato a conoscerla. In questo caso vengono utilizzate due implementazioni dello strategy: `MoveStrategy` e `CopyStrategy`, rispettivamente per muovere e copiare i file dalla prima cartella nella seconda.

Il modo in cui questa parte di codice è scritta lascia molta possibilità in un futuro di implementare classi concrete che gestiscano i file FACPL in modi completamente diversi. Il package `entryPoint` infatti è pensato per essere utile anche in progetti di altra natura rispetto a quello in esame, fornendo però un metodo semplice per eseguire il logging su un file e uno scheletro del workflow da seguire.

3.8 UTILIZZO DI MAVEN

L'implementazione di Maven [14] è stata fatta verso la fine della scrittura del codice del progetto, questo perchè il codice FACPL è rilasciato senza utilizzare alcun tipo di tool per la gestione del progetto. Infatti anche nella documentazione⁷ il modo indicato per utilizzare la libreria è quello di scaricare un file .zip manualmente dalla pagina github [9]. Le API di OpenNebula sono inoltre distribuite sotto forma di file .jar⁸. Per questi motivi in prima battuta si è cercato di importare tutte le dipendenze a mano ma questa opzione si è rivelata impraticabile nel lungo periodo. Dato il largo utilizzo di librerie esterne che è stato fatto e la necessità di utilizzare diversi package con versioni specifiche per garantire la compatibilità con la versione di Java 8 utilizzata da FACPL, cercare e scaricare i file .jar giusti stava diventando praticamente impossibile senza

⁷ <https://facpl.readthedocs.io/en/latest/txt/install.html>

⁸ <https://downloads.opennebula.io/packages/opennebula-5.12.0.4/>

grandi perdite di tempo.

Utilizzare Maven è stata una scelta molto comoda anche per il rilascio del codice una volta ultimato, dato che ha permesso di fornire delle indicazioni di utilizzo e installazione chiare.

Il primo passo in questo senso è stato quello di utilizzare la UI di Eclipse per convertire automaticamente il progetto Java in un progetto Maven⁹. La conversione automatica in questo caso era abbastanza funzionante, tuttavia si è reso necessario rimuovere le dipendenze dalle librerie locali di OpenNebula e FACPL e inserire queste dipendenze all'interno del file `pom.xml` affinché il progetto fosse utilizzabile in un ambiente nuovo. Per fare questo passaggio sono state valutate due strade:

- richiedere agli utente di installare tramite `mvn install` a mano le librerie prima di poter avviare il progetto;
- installare queste librerie in una cartella locale che sarà poi usata come repository nel file `pom.xml`.

Per rendere più comoda l'installazione all'utente finale si è optato di scegliere la seconda strada, includendo nel progetto la cartella *libs* e installando in quella cartella le librerie di OpenNebula e FACPL. Dopodiché è stato sufficiente inserire le seguenti righe nel file `pom.xml` per far sì che le librerie in questione fossero accessibili come tutte le altre dipendenze disponibili nei repository online:

Listing 16: Repository locale

```
1 <repositories>
2   <repository>
3     <id>local-libs</id>
4     <url>file://${project.basedir}/libs</url>
5   </repository>
6 </repositories>
```

Listing 17: Dipendenza di OpenNebula

```
1 <dependencies>
2 ...
3   <dependency>
4     <groupId>com.opennebula</groupId>
5     <artifactId>opennebula</artifactId>
6     <version>5.12.0.4</version>
7   </dependency>
```

⁹ https://wiki.eclipse.org/Converting_Eclipse_Java_Project_to_Maven_Project

```
8 ...  
9 <dependencies>
```

3.9 INTRODUZIONE ALLA WEB-APP E IDEAZIONE DEL FRONT-END

Al termine dello sviluppo della logica discussa in questo capitolo fino alla sezione precedente, il progetto era adatto per essere utilizzato ed implementato in un ambiente reale, tuttavia non veniva fornito alcun modo diretto per interagire con le classi senza aprire il progetto con un IDE. Non era presente un esempio delle potenzialità del package `entryPoint` dato che non si definiva una forma di interfaccia utente che permettesse di creare dei file `.fpl` senza passare da un IDE, che è proprio il punto fondamentale dell'esistenza delle classi in qual package.

Il progetto fino ad ora discusso sarà riferito da qui in avanti come *resource_management* mentre il progetto che sarà definito in questo paragrafo prenderà il nome di *policy_manager*. L'esempio concreto che abbiamo deciso di implementare è una web-app simile a quella descritta all'interno di [10]. È stato quindi pensato quali caratteristiche avrebbe dovuto avere la UI e solo successivamente si è deciso come implementarle effettivamente sia a livello di frontend che a livello di backend. Le funzioni che doveva necessariamente fornire la UI erano:

- un modo per decidere le policy da avere attive nel sistema;
- un modo per inviare delle richieste;
- un modo per garantire che le policy attivate e le richieste inviate rispettassero la sintassi di FACPL.

Per fornire queste tre caratteristiche si è pensato alla creazione di una UI che permetta di scrivere policy e richieste e consenta prima di validarle e successivamente anche di inviarle. Per le richieste è stato fornito un menù a scelta multipla che permette di inviare richieste senza bisogno di ricordare la sintassi corretta, tuttavia tale menù mette anche a disposizione la possibilità di aggiungere a mano dei parametri diversi da quelli di default. Non tutti i parametri sono sempre necessari, di conseguenza è anche possibile lasciarne alcuni vuoti e avere comunque la richiesta validata. La validazione di una richiesta controlla soltanto che la sintassi sia corretta e di conseguenza anche richieste che non hanno logicamente senso vengono validate; tuttavia una volta inviate al sistema FACPL sarà

il PDP a rifiutarle.

Un'ulteriore logica che sarà necessario aggiungere prima di inserire l'applicazione in un vero sistema cloud è che, usando il sistema di accesso presente, si definiscano i diritti di accesso alle funzionalità dell'utente. A seconda dell'utente alcune opzioni potrebbero quindi essere bloccate, prima fra tutte la possibilità di cambiare le policy e la selezione del Profile ID che dovrebbe essere sicuramente legato strettamente all'utente che ha eseguito l'accesso. Alcuni utenti potrebbero anche del tutto rimanere all'oscuro delle policy e quindi vedere soltanto la parte relativa alle richieste.

Il semplice frontend che è fornito nel progetto si presenta come si può vedere nella figura 2, la tecnologia utilizzata è JavaScript con una configurazione di HTML e CSS per rendere il tutto più piacevole alla vista. Questo vuole però essere soltanto uno spunto che necessita di ulteriore sviluppo prima di essere inserito in un vero sistema cloud.

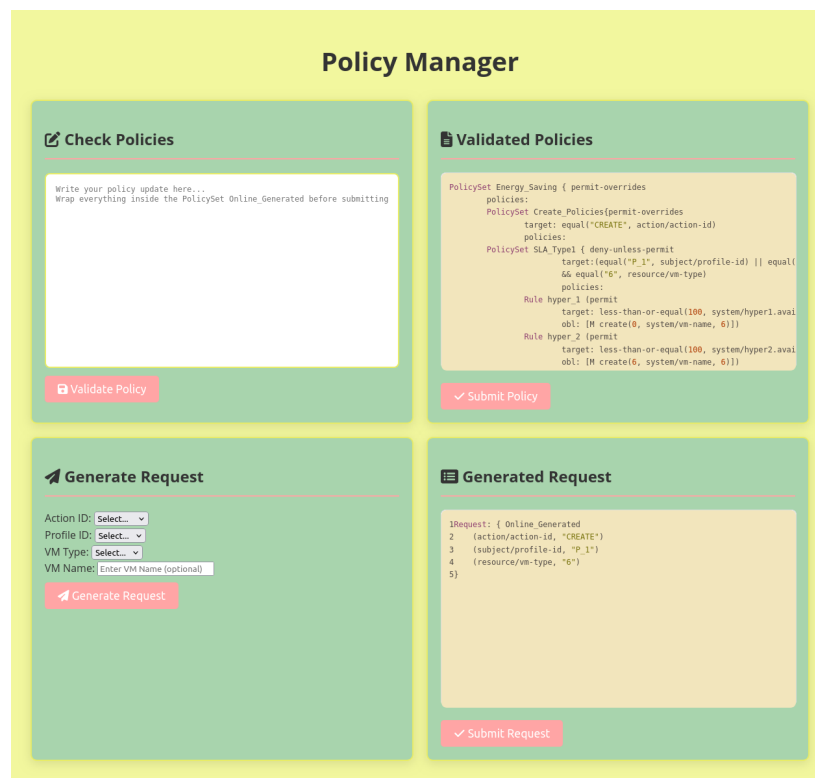


Figura 2: Front-end di policy manager (web-app)

A livello implementativo si evidenzia l'utilizzo della libreria *highlight.js* [5]

per colorare la sintassi delle policy e delle richieste. Questa libreria è stata scelta per la sua semplicità di utilizzo ma anche perchè permette di definire colori personalizzati per ogni linguaggio e anche di aggiungere ulteriori linguaggi con uno sforzo minimo grazie all'utilizzo di file JavaScript appositi, come si può leggere dalla documentazione ufficiale¹⁰.

3.10 SVILUPPO DEL BACKEND

Il backend è stato realizzando sfruttando il framework *Spring Boot* [11], La cui idea di base è quella di semplificare la realizzazione di applicazioni basate su Spring [12]. Usare Spring Boot Permette di evitare la configurazione manuale di Spring e nel nostro caso ci ha consentito di realizzare un backend per la nostra web-app in un modo molto semplice ma comunque modulare. Integrare la logica già descritta nei paragrafi precedenti è stato immediato sia grazie al modo in cui era stato realizzato il package `entryPoint` che grazie alla gestione automatica del server web che viene fatta da Spring Boot. In questo esempio infatti le funzionalità di Spring sono state esplorate soltanto in parte ma per il modo in cui è scritto il codice sarà semplice integrare in futuro un database, un servizio di autenticazione o qualunque altro tipo di servizio che si renderà necessario. Inoltre sebbene siano state lasciate le configurazioni di default per springboot e queste fossero sufficienti per il nostro caso di esempio, il framework ha un'ottima documentazione, una grande community e permette di adattarsi a un gran numero di casi concreti.

Gli unici due accorgimenti preliminari che sono stati necessari sono:

- scegliere una versione di Spring Boot compatibile con Java 8: a partire dalla major version 3 infatti, Spring Boot richiede obbligatoriamente una versione di Java 17 o maggiore e quindi la versione da noi è scelta è la 2.7, ovvero l'ultima compatibile con Java 8;
- scegliere un framework di logging: di default Spring Boot utilizza logback, ma nel nostro caso abbiamo scelto di usare log4j perchè sia FACPL che Spring utilizzano slf4j come *facade* rendendo impossibile mantenere entrambi i framework di logging. In questo caso risulta più comodo cambiare il framework usato da Spring piuttosto che da FACPL. Per fare questo è stato sufficiente aggiungere la seguente dipendenze al file `pom.xml`:

¹⁰ <https://highlightjs.readthedocs.io/en/latest/language-guide.html>

Listing 18: Esclusione di logback

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-web</artifactId>
4   <exclusions>
5     <exclusion>
6       <groupId>org.springframework.boot</groupId>
7       <artifactId>spring-boot-starter-logging</artifactId>
8     </exclusion>
9   </exclusions>
10 </dependency>
```

Grazie a queste righe Spring esclude la dipendenza da logback che è il framework di logging di default e sarà in grado di utilizzare log4j dato che FACPL lega slf4j a log4j. Si evidenzia come questa problematica si presenti solamente nell'ambiente di sviluppo di Eclipse perchè gestisce le dipendenze in modo diverso rispetto a Maven. Utilizzando Maven da terminale il risultato è diverso e viene invece in automatico utilizzato il framework di logging di Spring. Per rendere il codice facilmente avviabile anche da Eclipse si è deciso di escludere comunque logback ma si è dovuta aggiungere anche una dipendenza da log4j di modo che il programma sia avviabile anche da terminale con Maven.

Per quanto riguarda le classi che sono state create, si è iniziato creando due *RestController*, ovvero due classi che ricevono chiamate HTTP e sono in grado di dare risposte HTTP, una per le policy e una per le richieste. Queste classi sono state create in modo da rispondere a chiamate POST e GET:

Listing 19: PolicyController

```
1 @RestController
2 @RequestMapping("/policies")
3 public class PolicyController {
4     private final PolicyService policyService;
5
6     public PolicyController(PolicyService policyService) {
7         this.policyService = policyService;
8     }
9
10    @GetMapping
11    public ResponseEntity<List<String>> getPolicies() throws
12        IOException {
13        return policyService.getPolicies();
14    }
15 }
```

```

14
15 @PostMapping("/validate")
16 public ResponseEntity<String> validatePolicies(@RequestBody
    List<String> policies) throws IOException {
17     return policyService.validatePolicies(policies);
18 }
19
20 @PostMapping("/submit")
21 public ResponseEntity<String> submitPolicies() throws IOException {
22     return policyService.submitPolicies();
23 }
24 }

```

Il costruttore di questa classe riceve in input un oggetto di tipo `PolicyService` tuttavia all'interno del progetto non troviamo alcuna chiamata a questo costruttore, questo perchè Spring Boot si occupa di creare un oggetto di tipo `PolicyService` a partire da un'altra classe che definisce alcuni Bean che verranno poi iniettati nelle classi che li richiedono. I Bean sono oggetti che vengono utilizzati da Spring per passare degli specifici oggetti di un certo tipo in altre classi, in questo caso il Bean di tipo `PolicyService` è stato definito in una classe chiamata `ServiceConfig`:

Listing 20: ServiceConfig

```

1 @Configuration
2 public class ServiceConfig {
3     @Value("${logging.compile.file.name}")
4     private String compileLog;
5     @Value("${facpl.compile.folderpath}")
6     private String compilePath;
7     @Bean
8     public ApplyPolicy applyPolicy() throws IOException {
9         return new ApplyPolicy(compileLog, compilePath);
10    }
11    @Bean
12    public RequestExecution requestExecution() throws IOException {
13        return new RequestExecution(compileLog, compilePath);
14    }
15 }

```

Le classi `PolicyController` e `RequestController` sono sostanzialmente identiche, infatti le richieste da gestire sono le stesse e queste classi non contengono alcun tipo di logica. La logica per la formulazione delle risposte è rimandata alle classi `PolicyService` e `RequestService` che a loro volta rimandano buona parte della gestione dei file alle classi presenti nel package `entryPoint` di *resource_management*.

Per questi motivi anche le due classi `PolicyService` e `RequestService` sono molto simili. La parte più interessante di queste classi è sicuramente il metodo per la validazione dei file FACPL, che è stato implementato come segue:

Listing 21: Metodo `validatePolicies`

```
1 @Service
2 public class PolicyService {
3
4     @Value("${facpl.policies.filepath}") String policiesFilePath;
5     @Value("${facpl.temp.filepath}") String tempFilePath;
6 ...
7     public ResponseEntity<String> validatePolicies(List<String> policies) throws
        IOException {
8         Files.write(Paths.get(tempFilePath), policies, StandardOpenOption.CREATE,
            StandardOpenOption.TRUNCATE_EXISTING);
9         if (FACPLFileValidator.validate(tempFilePath)) {
10             Files.move(Paths.get(tempFilePath), Paths.get(policiesFilePath),
                StandardCopyOption.REPLACE_EXISTING);
11             return ResponseEntity.ok("Valid FACPL file!");
12         } else {
13             return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Policy
                validation failed");
14         }
15     }
16 ...
17 }
```

Per leggere la posizione di `policiesFilePath` e `tempFilePath` è stato utilizzato il file `application.properties` che è stato inserito nella cartella `src/main/resources` come di default per le applicazioni Spring. Questo lascia la libertà agli utilizzatori di cambiare la posizione dei file senza dover ricompilare il codice. Le posizioni di default sono chiaramente scelte di modo da essere correttamente leggibili e scrivibili da un utente con permessi di lettura e scrittura nella cartella del progetto. Nel caso in cui venissero cambiate è necessario assicurarsi che l'utente che lancia l'applicazione abbia i permessi necessari per leggere e scrivere nelle nuove cartelle impostate.

Questo metodo riceve da parte del metodo `getPolicies()` della classe `PolicyController`, una lista di stringhe che rappresentano le policy, scrive queste stringhe in un file temporaneo e poi chiama il metodo `validate` della classe `FACPLFileValidator` che è stato creato appositamente per validare i file FACPL. Questo metodo restituisce un booleano che indica se il file è valido o meno: in caso positivo il file temporaneo viene spostato

nella posizione definitiva, altrimenti viene restituito un errore. Si nota inoltre che il return è fatto con un oggetto `ResponseEntity` che permette di restituire una risposta HTTP con un codice di stato e un corpo; in questo caso il corpo è una stringa che indica se la validazione è andata a buon fine o meno.

Nel codice è poi presente una classe `GlobalExceptionHandler` che grazie alle annotazioni di Spring `@ControllerAdvice` e `@ExceptionHandler` permette di gestire le eccezioni che vengono lanciate all'interno del programma, in questo caso vengono gestite le due eccezioni `Exception` come errore generico e `IOException` in modo specifico.

Listing 22: `GlobalExceptionHandler`

```

1 @ControllerAdvice
2 public class GlobalExceptionHandler {
3
4     @ExceptionHandler(Exception.class)
5     public ResponseEntity<String> handleGeneralException(Exception e) {
6         return
7             ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("An
8             unexpected error occurred: " + e.getMessage());
9
10    }
11
12    @ExceptionHandler(IOException.class)
13    public ResponseEntity<String> handleIOException(IOException e) {
14        System.out.println("Got caught");
15        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("File
16        processing error: " + e.getMessage());
17    }
18 }

```

Il primo metodo restituisce un errore 500, ovvero `INTERNAL_SERVER_ERROR` e il secondo un errore 400, ovvero `BAD_REQUEST`. Questa classe è lasciata come esempio di possibile implementazione futura, anche se come si può vedere nel codice delle altre classi la maggior parte delle eccezioni sono gestite singolarmente in questo momento e quindi questa classe non verrà acceduta a meno di errori inattesi. In un'applicazione reale si potrebbe rendere necessario usare questa classe in modo più ampio.

3.11 INTEGRAZIONE DELLA WEB-APP CON MAVEN

`policy_validation`, al contrario di `resource_management` è stato pensato fin da subito come un progetto Maven e di conseguenza non c'è stato

bisogno di alcuna conversione. Spring Boot ha un'integrazione molto semplice con Maven e inoltre, avendo già definito il modo in cui si intendeva integrare dipendenze locali come visto nella sezione 3.8, far funzionare il progetto anche eseguendolo da terminale con Maven è stato immediato.

L'unica scelta da prendere riguardava la gestione della dipendenza diretta di *policy_validation* da *resource_management*. Una possibilità che è stata presa in considerazione è stata quella di accorpare i due progetti. Tuttavia, per la natura di *policy_validation*, si è valutato che parecchi utenti potrebbero ritenerlo superfluo e quindi che sia meglio mantenere i due progetti separati.

Una volta presa questa decisione si è dovuto pensare se fosse opportuno gestire questa dipendenza come tutte le altre dipendenze locali; questo avrebbe richiesto diversi passaggi, fra cui la creazione del file .jar di *resource_management*. La cosa che si è ritenuta più comoda è stata, in questo caso, richiedere di installare il progetto *resource_management* in locale con Maven, prima di poter utilizzare *policy_validation*.

I due progetti sono quindi pensati per essere distribuiti insieme, ma il primo è completamente indipendente dal secondo, che può essere ignorato o addirittura cancellato se non si intende usarlo. Dato che invece il secondo dipende dal primo e si vuole rendere facile l'applicazione delle modifiche apportate a *resource_management* di modo che abbiano effetto sul funzionamento di *policy_validation*, l'unica scelta adatta risulta essere quella presa.

3.12 TESTING

Tutte le classi presenti nei due progetti discussi in questi paragrafi presentano delle opportune classi di test. Il framework di test utilizzato è *JUnit5* [6] che è stato scelto principalmente per la sua larga diffusione e semplicità di utilizzo.

I test per il progetto *resource_management* sono stati scritti in modo basilare senza utilizzare alcun tipo di framework per la gestione dei Mock. I Mock sono stati comunque necessari date le molte integrazioni con le API di OpenNebula ma si è fatto uso di classi create ad-hoc per il mocking, cosa che ci ha permesso anche di renderci conto più facilmente della testabilità del nostro codice dato che non si è mai fatto ricorso alla

reflection di Java.

Un esempio di questo è la classe CreateVM, mostrata nel listing 8, la cui classe di test CreateVMTest si presenta nel seguente modo:

Listing 23: CreateVMTest

```

1 Public class CreateVMTest {
2
3     private StringBuilderLogHandler mockLogHandler;
4     private Logger mockLogger;
5
6     @BeforeEach
7     void setUp() throws ClientConfigurationException {
8         mockLogger = Logger.getLogger(CreateVMTest.class.getName());
9         mockLogHandler = new StringBuilderLogHandler();
10        mockLogger.addHandler(mockLogHandler);
11    }
12
13    @Test
14    public void testEvalSuccess() throws ClientConfigurationException {
15        MockClientTrue mockClient = new MockClientTrue("150");
16        MockOpenNebulaActionContext mockContext = new
17            MockOpenNebulaActionContext(mockClient, mockLogger);
18
19        CreateVM createVMAction = new CreateVM(mockContext);
20
21        List<Object> args = Arrays.asList(0, "VMName", 1);
22        createVMAction.eval(args);
23
24        assertTrue(mockLogHandler.getLogBuilder().contains("INFO: Starting VM:
25            [1, VMName]\n"));
26        assertTrue(mockLogHandler.getLogBuilder().contains("INFO: 150\n"));
27    }
28
29    @Test
30    public void testEvalFailure() throws ClientConfigurationException {
31        MockClientFalse mockClient = new MockClientFalse("150");
32        MockOpenNebulaActionContext mockContext = new
33            MockOpenNebulaActionContext(mockClient, mockLogger);
34
35        CreateVM createVMAction = new CreateVM(mockContext);
36
37        List<Object> args = Arrays.asList(0, "VMName", 1);
38        createVMAction.eval(args);
39
40        assertTrue(mockLogHandler.getLogBuilder().contains("INFO: Starting VM:
41            [1, VMName]\n"));
42        assertTrue(mockLogHandler.getLogBuilder().contains("SEVERE: 150\n"));
43    }
44 }

```

Notiamo l'utilizzo di tre classi di Mock: StringBuilderLogHandler, MockClientTrue e MockClientFalse. La prima è una classe creata appositamente per il testing che permette di salvare i log in una stringa, le

altre due sono classi create per il testing che estendono la classe `Client` di `OpenNebula` e che permettono di simulare il comportamento di un client vero e proprio. Queste classi permettono di testare la classe `CreateVM` in modo indipendente dalla classe `Client` di `OpenNebula`, verificando soltanto i side effects delle operazioni, ovvero in questo caso verificando che le stringhe di logging siano coerenti con il comportamento atteso.

Per quanto riguarda *policy_validation* i test sono stati scritti in modo abbastanza diverso. Infatti dato che il progetto si basa sul framework `Spring Boot` si è reso necessario l'utilizzo delle annotazioni specifiche per i test fornite da `Spring` oltre che il framework `Mockito` [13] per la gestione dei Mock più complessi dove non era necessario avviare l'applicazione `Spring Boot`. `Spring` fornisce infatti molte opzioni per il testing, alcune delle quali avviano l'intera app, altre che permettano di inizializzare soltanto il livello web e altre che non hanno proprio bisogno di avviare l'applicazione, il tutto per rendere i test più efficienti e indipendenti dai layer che non sono in fase di test in specifico¹¹. Anche in questo caso si evidenzia come queste potenzialità siano state esplorate soltanto superficialmente data la semplicità della web-app in esame.

Un esempio di test per il progetto *policy_validation* sono i test per la classe `PolicyController`, mostrata nel listing 19, che si presentano nel seguente modo:

Listing 24: `PolicyControllerTest`

```
1 @WebMvcTest(PolicyController.class)
2 class PolicyControllerTest {
3
4     @Autowired
5     private MockMvc mockMvc;
6
7     @MockBean
8     private PolicyService policyService;
9
10    @Test
11    void getPolicies_ShouldReturnPolicyFileContent() throws Exception {
12        String policyContent = "Mock Policy Content";
13        given(policyService.getPolicies())
14            .willReturn(ResponseEntity.ok(Arrays.asList(policyContent)));
15
16        mockMvc.perform(get("/policies"))
17            .andExpect(status().isOk())
18            .andExpect(content().contentType(MediaType.APPLICATION_JSON))
19            .andExpect(content().json("[\"Mock Policy Content\"]"));
20    }
21
22    @Test
```

¹¹ <https://docs.spring.io/spring-framework/reference/testing.html>

```

23 void getPolicies_WhenFileNotFound_ShouldReturnNotFound() throws Exception {
24     given(policyService.getPolicies())
25         .willReturn(ResponseEntity.status(HttpStatus.NOT_FOUND)
26             .body(Collections.singletonList("Policies file not found.")));
27
28     mockMvc.perform(get("/policies")
29         .accept(MediaType.APPLICATION_JSON))
30         .andExpect(status().isNotFound())
31         .andExpect(content().contentType(MediaType.APPLICATION_JSON))
32         .andExpect(content().json("[\"Policies file not found.\"]"));
33 }
34
35 @Test
36 void validatePolicies_ShouldReturnOk() throws Exception {
37     String policyContent = "Mock Policy Content";
38     given(policyService.validatePolicies(Arrays.asList(policyContent)))
39         .willReturn(ResponseEntity.ok("Validation successful"));
40
41     mockMvc.perform(post("/policies/validate")
42         .contentType(MediaType.APPLICATION_JSON)
43         .content("[\"Mock Policy Content\"]"))
44         .andExpect(status().isOk())
45         .andExpect(content().string("Validation successful"));
46 }
47 ...
48 }

```

In questo caso si può notare l'utilizzo delle annotazioni `@WebMvcTest` che permette di caricare soltanto la parte da noi in testing dell'applicazione. Notiamo inoltre l'utilizzo di `@AutoWired` per iniettare il mock dell'Mvc (*model-view-controller*) di Spring e di `@MockBean` per iniettare il mock della classe `PolicyService`. Questo permette di testare la classe `PolicyController` in modo indipendente dalla classe `PolicyService` e di verificare che le risposte siano coerenti con il comportamento atteso.

Soltanto alcuni dei metodi di test sono stati riportati, questo perchè seguono tutti lo stesso schema e differiscono soltanto per i parametri passati e per le risposte attese. Vengono utilizzati `given` e `willReturn` per definire il comportamento del mock e successivamente vengono chiamati i metodi `perform` e `andExpect` per effettuare la chiamata e verificare che la risposta sia coerente con il comportamento atteso.

3.13 RILASCIO DEL PROGETTO

Come discusso in precedenza, i due progetti sono distribuiti insieme e disponibili alla pagina github [FACPL_OpenNebula](#) con un file README che spiega come installarli e utilizzarli. Questi progetti sono pensati per

essere utilizzati da un utente di OpenNebula, quindi l'installazione locale tramite Maven dei progetti deve essere fatta con un utente di questo tipo. Se si cerca di installarlo con un utente che non appartiene a questo gruppo alcuni test falliranno, questo perchè non si riesce ad instanziare nessun oggetto di classe `Client`. Il comportamento è voluto e serve a garantire che l'utente che installa il progetto abbia i permessi necessari per utilizzarlo. Nel file README è comunque brevemente indicato un modo per aggirare il problema che consiste nel saltare la fase di test, installare *resource_management* e avviare quindi tranquillamente *policy_validation*, tuttavia affinché i comandi sulle virtual machine funzionino bisogna per forza avere almeno delle credenziali valide da inserire nella classe `ContextStub_Default` e quindi questo metodo è sconsigliato perchè è molto più consono avviare il programma con l'utente corretto.

Chiaramente è richiesta all'utente l'installazione di Maven e la presenza di Java 8. L'IDE consigliato per lo sviluppo è Eclipse. Non sono stati testati altri IDE ma una volta installati i progetti nel proprio Maven locale, questi sono sicuramente accessibili anche da altri IDE che supportano progetti Maven, includendo le dipendenze nel file `pom.xml`. Il progetto è testato e funzionante su Ubuntu 20.04, Ubuntu 22.04 e Arch Linux.

UTILIZZO DEI PROGETTI IN UN SERVER REALE

Per fornire un esempio di utilizzo dei progetti presentati nei capitoli precedenti, si è deciso di installare OpenNebula su un computer di prova. Il computer scelto è un desktop posizionato all'interno di una stanza ad uso degli studenti di informatica. Il computer non è raggiungibile dall'esterno ma è accessibile tramite rete cablata e wifi interna alla stanza. Questo permette di avere un ambiente di test semplice ma vicino alla realtà.

Le specifiche principali del desktop sono:

- processore: Intel Core i7-13700 (24 core);
- scheda grafica integrata: Intel UHD 770;
- RAM: 32 GB ddr5.

Questo computer servirà sia come gestore degli host che come uno degli host stessi. L'operazione è tranquillamente consentita in OpenNebula; da ora in avanti questo computer sarà riferito come *server*. In questo esempio è stato considerato anche un altro host, un computer portatile presente nella stessa rete del *server* OpenNebula con le seguenti specifiche:

- processore: Intel Core i5-8350U (8 core);
- scheda grafica integrata: Intel UHD 620;
- RAM: 8 GB ddr4.

Questo computer sarà riferito da qui in avanti come *laptop*.

Le policy scelte sono state quelle per il load balancing. Questa scelta è data dal fatto che il *server* è molto più potente del *laptop* e quindi sarà facile vedere se il bilanciamento viene fatto correttamente.

4.1 INSTALLAZIONE DEI DUE PROGETTI

In questo capitolo non sarà presa in considerazione l'installazione di OpenNebula; tutte le procedure descritte in questa sezione saranno riferite ad un sistema in cui OpenNebula è già installato e funzionante (per informazioni sulle procedure seguite fare riferimento alla documentazione ufficiale¹). Stessa considerazione è fatta anche per i nodi².

Per eseguire correttamente il codice descritto nel capitolo 3 è stato necessario innanzitutto aprire un terminale ad autenticarsi come utente del gruppo *oneadmin*, da cui si è dovuto clonare la repository github per poi seguire le procedure descritte nel file `README.md`. Si evidenzia come sia fondamentale che l'utente che esegue i comandi sul terminale da qui in avanti abbia diritti di lettura, scrittura e esecuzione sui file appropriati, di conseguenza è consigliato direttamente clonare la repository con l'utente corretto; i permessi si possono fornire in un secondo momento se lo si preferisce, ma se non risultano del tutto corretti i risultati potrebbero essere inaspettati. In questo caso si vuole testare il funzionamento di *policy_manager* di conseguenza i passaggi da seguire sono:

- Lato *server*:
 - entrare nella cartella *resource_management* ed eseguire il comando `mvn install` assicurandosi che tutti i test che sono eseguiti abbiano successo. Questo comando installerà il progetto all'interno della repository locale di Maven, passaggio fondamentale per eseguire con successo il comando successivo;
 - entrare nella cartella *policy_manager* ed eseguire il comando `mvn test` per assicurarsi che i test vadano a buon fine e tutto possa funzionare correttamente;
 - ancora all'interno della cartella *policy_manager* eseguire il comando `mvn spring-boot:run` per far partire il server gestito con Spring-Boot.
 - entrare nella cartella *policy_manager/config.properties* e modificare i campi in modo consono con la propria installazione di OpenNebula. In questo caso abbiamo considerato i due host di

1 https://docs.opennebula.io/6.8/installation_and_configuration/frontend_installation/install.html

2 https://docs.opennebula.io/6.8/open_cluster_deployment/kvm_node/kvm_node_installation.html

ID 0 e 2, e i due template di ID 0 e 1. Il file di configurazione avrà quindi la seguente forma:

Listing 25: config.properties

```
1 hyper1.host.id=0
2 hyper2.host.id=2
3 type1.template.id=0
4 type2.template.id=1
5 context.file.location=opennebula_context_actions/
```

- lato client *da eseguire una volta*:
 - connettersi all'ip del server sulla porta 8080;
 - validare le policy che si vogliono utilizzare, in questo caso abbiamo scelto quelle di load balancing presenti nel progetto;
 - modificare le policy per aderire al caso reale in questione. In particolare quindi inserire i corretti ID degli host e dei template. Ovviamente questo passaggio può essere saltato se si scrivono le policy direttamente per il proprio sistema senza usarne di generiche già scritte;
- lato client *da eseguire ogni volta*:
 - connettersi all'ip del server sulla porta 8080;
 - creare una richiesta con il menù a tendina disponibile e valdarla;
 - inviare la richiesta;

Nel nostro esempio di test tutti i passaggi hanno funzionato senza alcun problema, anche se in una precedente installazione su un altro server si era evidenziata la necessità di inserire alcune dipendenze nel file `pom.xml` rispetto a quelle che erano necessarie sul computer di test originale. Si evidenzia inoltre come non sia stato necessario modificare alcun file `.java`.

4.2 SPECIFICHE DELL'ESEMPIO

Le valutazioni fatte da FACPL sono visibili direttamente nel terminale da cui è stato eseguito il comando `mvn spring-boot run`. Le informazioni sull'applicazione delle policy e l'invio delle richieste, oltre che le informazioni sulle azioni svolte sulle virtual machine, sono visibili nei file di log posti nella cartella `logs`.

L'utente può connettersi a OpenNebula Sunstone, accessibile di default sulla porta 9869 del server per vedere la situazione delle virtual machine e degli host. Per farlo deve però essere in possesso delle credenziali di un account di OpenNebula.

Nel nostro caso di esempio abbiamo considerato degli utenti che si connettono alla web-app di *policy_manager* attraverso la rete wifi. Nessun utente ha la possibilità di modificare le policy, però tutti possono vederle e quindi formattare le loro richieste di conseguenza. Alcuni utenti appartengono al gruppo P_2 mentre altri al gruppo P_1, quindi è possibile verificare il cambiamento del sistema all'invio di specifiche richieste da entrambi i gruppi.

4.3 RISULTATI DELL'ESECUZIONE

Di seguito sono riportate le prime richieste e i risultati del loro invio sulla dashboard di OpenNebula:

<input type="checkbox"/>	ID	Name	Owner	Group	Status	Host	IPs
<input checked="" type="checkbox"/>	32	c1105b73-c9fc-46a0-988b-d3e78b9f801e	oneadmin	oneadmin	RUNNING	localhost	--

Figura 3: Dashboard di OpenNebula con una sola virtual machine

Generated Request

```
Request: { Online_Generated
  (action/action-id, "CREATE")
  (subject/profile-id, "P_1")
  (resource/vm-type, "1")
}
```

✓ Submit Request

Generated Request

```
Request: { Online_Generated
  (action/action-id, "CREATE")
  (subject/profile-id, "P_2")
  (resource/vm-type, "1")
}
```

✓ Submit Request

Figura 4: Richieste di P_1 e P_2

Come si può vedere nonostante siano state eseguite due richieste è stata creata soltanto una macchina virtuale. Questa logica è corretta perchè l'utente P_1 non è autorizzato a creare virtual machine con il template 1. Le valutazioni del sistema FACPL sono quindi:

Request: Online_Generated	Request: Online_Generated
PDP Decision= Decision: DENY Obligations:	PDP Decision= Decision: PERMIT Obligations: PERMIT M create([0, c1105b73-c9fc-46a0-988b-d3e78b9f801e, 1])
PEP Decision= DENY	PEP Decision= PERMIT

Figura 5: Valutazioni di FACPL sulle due richieste

A questo punto sono state inviate ulteriori richieste da utenti con ID P_2 sia per virtual machine di tipo 1 che per virtual machine di tipo 0. Si nota che il sistema continua ad inserire tutte le virtual machine sul *server* fino a che non si raggiunge un bilanciamento di risorse rimaste tra *server* e *laptop*:

<input type="checkbox"/>	ID	Name	Cluster	RVMs	Allocated CPU	Allocated MEM	Status
<input type="checkbox"/>	2	192.168.10.6	0	0	0 / 800 (0%)	0KB / 15.5GB (0%)	ON
<input type="checkbox"/>	0	localhost	0	10	1600 / 2400 (67%)	7.5GB / 31GB (24%)	ON

Figura 6: Dashboard di OpenNebula con i due host bilanciati

Il load balancing sembra quindi funzionare correttamente. Per mostrarlo ulteriormente si continuano a creare delle virtual machine di modo da verificare che comincino ad essere aggiunte anche all'interno del *laptop*:









<input type="checkbox"/>	ID	Name	Owner	Group	Status	Host	IPs	
<input type="checkbox"/>	48	4668729e-dc82-4fc6-8591-262587e33624	oneadmin	oneadmin	RUNNING	192.168.10.6	--	 
<input type="checkbox"/>	47	60be2986-b370-49ba-87ee-35e66b701323	oneadmin	oneadmin	RUNNING	localhost	--	 
<input type="checkbox"/>	46	fae092f1-1fc5-4eec-a263-157808027f8b	oneadmin	oneadmin	RUNNING	192.168.10.6	--	 
<input type="checkbox"/>	45	cfd13ec4-521f-4f8c-a9bd-f63ffb57620	oneadmin	oneadmin	RUNNING	localhost	--	 

Figura 7: Dashboard con le virtual machine alternate fra i due host

Continuando in questo modo si raggiunge una situazione in cui entrambi gli host hanno tutte le CPU utilizzate al 100%. A questo punto secondo le policy scelte il sistema dovrebbe controllare se ci sono alcune

virtual machine di tipo 1 da freezzare in favore della creazione di virtual machine di tipo 2. Questo infatti è quello che succede:

```
-----
Request: Online_Generated

PDP Decision=
Decision: PERMIT Obligations: PERMIT M freezeMultiple([0, 2, 0])PERMIT M create([0, 52d87ca7-b0c0-42dd-803c-946b67fb2470, 1])

PEP Decision=
PERMIT
-----
```

Figura 8: Valutazione di FACPL

<input type="checkbox"/>	ID	Name	Cluster	RVMs	Allocated CPU	Allocated MEM	Status
<input checked="" type="checkbox"/>	2	192.168.10.6	0	4	800 / 800 (100%)	3GB / 15.5GB (19%)	ON
<input checked="" type="checkbox"/>	0	localhost	0	14	2400 / 2400 (100%)	10.5GB / 31GB (34%)	ON

Figura 9: Dashboard con entrambi gli host al 100% di utilizzo

<input type="checkbox"/>	39	38c95a8f-77ee-456d-9d9d-88c2a0eb70db	oneadmin	oneadmin	STOPPED	--	--
<input type="checkbox"/>	38	38fdfeea-4970-43c8-9564-76407b39eb36	oneadmin	oneadmin	STOPPED	--	--

Figura 10: virtual machine stoppate automaticamente

Le due virtual machine freezzate erano appartenenti al *server*, anche perchè era l'unico dei due host ad avere macchine di tipo 1 attive e quindi freezzabili. Possiamo notare questo anche nella figura 9, dato che il numero delle virtual machine sul *server* è 14. Questo numero è dato dalla presenza di 10 virtual machine che richiedono 2 processori l'una e 4 virtual machine che richiedono 1 processore l'una. Sul *laptop* ci sono invece soltanto 4 virtual machine che richiedono 2 processori l'una.

Se si tenta di creare ulteriori virtual machine di tipo 1 il sistema FACPL ritorna correttamente una valutazione di *DENY* e quindi la richiesta non viene neanche inviata a OpenNebula come si può vedere nella seguente immagine:

```
-----
Request: Online_Generated

PDP Decision=
Decision: DENY Obligations: DENY 0 log([Not enough available resources for TYPE_1 VMs])

PEP Decision=
DENY
-----
```

Questo è un comportamento corretto dato che in realtà OpenNebula permette di sovraccaricare il sistema ma non è quello che vogliamo nel nostro caso.

A questo punto l'unico passo rimasto è testare se le macchine vengono correttamente rilasciate anche con l'utilizzo diretto delle richieste di tipo *RELEASE*. Per fare questo si è resettato il sistema, si è poi creato una virtual machine e si è verificato che FACPL valutasse correttamente la richiesta di rilascio della stessa. Come si può vedere nelle seguenti figure l'esecuzione è andata a buon fine:



Figura 11: Richiesta di rilascio

```
Request: Online_Generated
PDP Decision=
Decision: PERMIT Obligations: PERMIT M release([0, 86c46de8-e1dd-4286-9356-33118deaa9de])
PEP Decision=
PERMIT
```

Figura 12: valutazione di FACPL

<input type="checkbox"/>	ID	Name	Owner	Group	Status	Host	IPs
<input checked="" type="checkbox"/>	86	86c46de8-e1dd-4286-9356-33118deaa9de	oneadmin	oneadmin	STOPPED	--	--

Figura 13: Virtual machine stoppata

CONCLUSIONI E SVILUPPI FUTURI

In questo capitolo verranno analizzati i punti positivi e le criticità riscontrabili nella modalità di utilizzo di FACPL per un progetto come quello in esame. Ci si concentrerà quindi sui pregi e difetti che caratterizzano la libreria Java con cui viene distribuito. Saranno inoltre proposti alcuni sviluppi futuri per migliorare la libreria e anche per sviluppare ulteriormente i progetti ideati in questa tesi.

5.1 CRITICITÀ E PUNTI DI FORZA DELLA LIBRERIA JAVA DI FACPL

Per questo progetto le prime criticità sono state quelle riguardanti la poca diffusione di FACPL e la conseguente mancanza di documentazione dettagliata. Capire il funzionamento della libreria non è stato immediato dato che anche la documentazione ufficiale non è completa. Non è spiegato all'effettivo come implementare le componenti del sistema che devono per forza essere modificate a mano come mostrato nella sezione 2.4.

È anche vero però che una volta compreso come interagiscono fra loro le classi della libreria, inserire le funzionalità già sviluppate è stato molto semplice. Questo ci porta a considerare che in effetti la libreria è ben sviluppata ma manca soltanto di una documentazione un po' più specifica che indichi come integrare il software in un progetto più ampio; in particolar modo servirebbe una guida su come gestire le variabili di sistema e le *PEPAction*.

Un'altra mancanza è una modalità semplice di conversione delle policy e delle richieste scritte in FACPL in codice Java. Sono documentati dei validatori e convertitori del linguaggio FACPL che però, come indicato più volte anche in questa tesi, sono utilizzabili solo tramite la UI di Eclipse. Nella libreria sono presenti delle classi di validazione e conversione di

file da FACPL verso diversi linguaggi fra cui anche Java, tuttavia non è presente alcun tipo di documentazione al riguardo; guardando la letteratura si fa sempre riferimento ai convertitori automatici accessibili da Eclipse. Una strada percorribile molto facilmente per migliorare questo aspetto, sarebbe quella di consigliare l'utilizzo dello *StandaloneGenerator* fornito come esempio.

Un altro aspetto migliorabile è la gestione delle dipendenze. Usare un gestore come Maven renderebbe la libreria più appetibile per un utilizzatore e permetterebbe di rendere la libreria più mantenibile e più semplice da aggiornare.

L'ultima problematica riguarda la dipendenza da Java 8; in realtà questo problema è facilmente risolvibile. Ad esempio durante il nostro sviluppo per un periodo si è compilato il codice con Java 11 senza problemi con qualche piccolo accorgimento. Tuttavia per evitare di dover svolgere test estensivi su parti di codice già testate con Java 8 si è dovuto ritornare ad utilizzare questa versione.

Parlando degli aspetti positivi troviamo sicuramente la facilità di integrazione con un progetto già esistente. Quando si utilizza FACPL un approccio che funziona bene è quello di pensare alla logica della propria implementazione in modo del tutto distaccato da FACPL e poi integrare le due parti. Per farlo basta scrivere del codice che fornisca due cose:

- dei metodi in grado di ritornare delle informazioni sullo stato del sistema;
- delle classi che permettano di eseguire delle azioni sul sistema e che devono aderire all'interfaccia *IPepAction*.

A quel punto cambiando le classi *ContextStub_Default* e *PepAction*, in poche righe il proprio codice è utilizzabile dal sistema FACPL.

L'altro aspetto molto positivo è la gestione del logging. Infatti questo è già molto espressivo e permette di capire in modo dettagliato le scelte che vengono fatte dal sistema. Questo torna utile in fase di test del proprio codice, oltre che per un eventuale utilizzatore finale.

5.2 SVILUPPI FUTURI

Per quanto riguarda la libreria Java di FACPL uno sviluppo futuro potrebbe riguardare l'aggiornamento ad una versione di Java LTS più recente

come la 11 o la 17. Questo permetterebbe innanzitutto di rimuovere alcune dipendenze da librerie esterne, che sono entrate a far parte delle librerie standard nelle versioni più recenti di Java. Inoltre tale aggiornamento renderebbe la libreria più adatta ad essere integrata con codice più recente.

L'altro aspetto già largamente discusso sarebbe una nuova modalità di gestione delle dipendenze, che renderebbe sicuramente più facile anche eseguire il passo descritto sopra.

Per concludere, una volta aggiornata la libreria, sarebbe utile migliorare la documentazione e mostrare un workflow consigliato da seguire in dei progetti più grandi come quello considerato in questa tesi.

Per quanto riguarda invece i due progetti ideati in questa tesi, sicuramente ci sono diverse possibilità di ampliamento. Si potrebbe pensare di integrare molte più *PEPAction* riguardanti OpenNebula. Questo passo non sarebbe difficile da fare dato che è già presente la classe astratta che può essere estesa e da cui partire per scrivere nuovi comandi. Alcuni comandi interessanti che potrebbero essere aggiunti sono il cambiamento di rete per una virtual machine o il cambiamento delle impostazioni di una virtual machine a runtime, entrambe azioni facilmente implementabili con le API di OpenNebula.

Una miglioria che si potrebbe portare avanti riguarda il front-end e l'integrazione di un sistema di autenticazione. Questa parte nella logica attuale è rimandata alla persona che intende inserire il codice nella sua infrastruttura. Il sistema di autenticazione potrebbe tornare utile per un utilizzatore finale che vuole partire da zero. Si considera però che nella maggior parte dei sistemi la nostra web-app sarà gestita tramite un sistema di autenticazione generale, che racchiude anche altre web-app. Il front-end è sicuramente migliorabile anche in molti altri modi, espressi nel capitolo 3.

RINGRAZIAMENTI

BIBLIOGRAFIA

- [1] Joshua Bloch. *Effective Java*. Addison-Wesley, 3rd edition, 2018.
- [2] OpenNebula Community. Opennebula. <https://opennebula.io/>.
- [3] Eclipse Foundation. Eclipse xtext: Language development framework. <https://eclipse.dev/Xtext/>, 2023.
- [4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994. Professional eBook edition.
- [5] highlight.js contributors. highlight.js - javascript syntax highlighter. <https://highlightjs.org/>, 2024.
- [6] JUnit Team. Junit 5. <https://junit.org/junit5/>, 2024.
- [7] Andrea Margheri. *Progetto e realizzazione di un linguaggio formale per il controllo degli accessi basato su politiche*. PhD thesis, Università degli Studi di Firenze, Italy, 2012.
- [8] Andrea Margheri. Facpl documentation. <https://facpl.readthedocs.io/en/latest/index.html>, 2024.
- [9] Andrea Margheri. Facpl: Flexible and adaptive control programming language. <https://github.com/andreamargheri/FACPL>, 2024.
- [10] Andrea Margheri, Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. Developing and enforcing policies for access control, resource usage, and adaptation. In Emilio Tuosto and Chun Ouyang, editors, *Web Services and Formal Methods*, pages 85–105, Cham, 2014. Springer International Publishing.
- [11] Spring.io. Spring Boot Project. <https://spring.io/projects/spring-boot>, 2024.
- [12] Spring.io. Spring Framework. <https://spring.io/>, 2024.
- [13] Mockito Team. Mockito. <https://site.mockito.org/>, 2024.
- [14] The Apache Software Foundation. Maven Project. <https://maven.apache.org/>.

- [15] The Apache Software Foundation. Apache commons. <https://commons.apache.org/>, 2024.
- [16] Dave Winer. Xml-rpc specification. <https://xmlrpc.com/>, 2024.