Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Un plugin Maven per l'automatizzazione della pubblicazione di documentazione software

Tesi di laurea triennale

Relatore
Prof.Paolo Baldan

Laureanda
Laura Cameran

Anno Accademico 2018-2019

"I made a discovery today. I found a computer.
Wait a second, this is cool. It does what I want it to.
If it makes a mistake, it's because I screwed it up. Not because it doesn't like me."

— The Mentor

Sommario

Il documento corrente descrive il lavoro svolto durante il periodo di stage, della durata di trecentoventi ore, dalla laureanda Laura Cameran presso l'azienda Finantix Pro Unipersonale S.r.l.
L'obiettivo principale da raggiungere era lo sviluppo di un plugin Maven al fine di automatizzare la pubblicazione di documentazione di software sul sistema documentale Atlassian Confluence. Per realizzare tale compito era richiesto inoltre lo studio di API RESTful, mezzo con cui interagire con Confluence.

Ringraziamenti

Ringrazio il mio tutor aziendale Gian Maria Romanato e il mio tutor interno Paolo Baldan per avermi aiutato e dato consiglio durante questa esperienza di stage.

Ringrazio inoltre tutte le persone che mi sono state vicine nel corso di questi anni di studio all'università.

Padova, Settembre 2019 Laura Cameran

Indice

1 Introduzione **1**
- 1.1 Il progetto . . . 1
- 1.2 Principali problematiche . . . 2
- 1.3 Strumenti utilizzati . . . 2
  - 1.3.1 Maven . . . 3
  - 1.3.2 Confluence . . . 4
  - 1.3.3 Riepilogo degli strumenti . . . 5
- 1.4 Il prodotto ottenuto . . . 7
- 1.5 Organizzazione del testo . . . 7

2 Analisi dei requisiti **9**
- 2.1 Premessa . . . 9
- 2.2 Descrizione del prodotto . . . 9
  - 2.2.1 Il goal *publish* . . . 10
  - 2.2.2 Il goal *cleanup* . . . 12
- 2.3 Raccolta dei requisiti . . . 13
- 2.4 Requisiti . . . 14
  - 2.4.1 Requisiti di funzionalità . . . 14
  - 2.4.2 Requisiti di qualità . . . 16
  - 2.4.3 Requisiti di vincolo . . . 17
  - 2.4.4 Riepilogo dei requisiti . . . 18
- 2.5 Tracciamento dei requisiti . . . 19

3 Progettazione e realizzazione **23**
- 3.1 Procedura di lavoro . . . 23
- 3.2 Tecnologie e librerie utilizzate . . . 23
  - 3.2.1 Javax . . . 23
  - 3.2.2 Codehaus Plexus . . . 24
  - 3.2.3 Maven . . . 24
  - 3.2.4 Jersey . . . 24
- 3.3 Diagramma dei package . . . 25
- 3.4 Diagrammi delle classi . . . 26
  - 3.4.1 Diagramma dei mojo . . . 26
  - 3.4.2 Diagramma del client . . . 29
  - 3.4.3 Diagramma delle componenti del plugin Docs . . . 31
  - 3.4.4 Riepilogo delle classi . . . 32
- 3.5 Diagrammi di sequenza . . . 33
  - 3.5.1 Diagramma del goal *publish* . . . 33

3.5.2 Diagramma del goal *cleanup* 34
3.6 Configurazione 36
3.6.1 Proprietà 38
3.7 Esecuzione 39
3.8 Documentazione 39
3.8.1 Manuale utente 39
3.8.2 Manuale sviluppatore 41

4 Verifica e validazione **43**
4.1 Analisi statica 43
4.2 Test di unità 44
4.2.1 JUnit 44
4.2.2 Mockito 45
4.3 Test di validazione 46

5 Conclusioni **47**
5.1 Risultato ottenuto 47
5.2 Analisi critica del prodotto e del lavoro di stage 47
5.2.1 Raggiungimento degli obiettivi 48
5.2.2 Conoscenze possedute e acquisite 49
5.2.3 Utilizzazione del prodotto 49
5.2.4 Valutazione degli strumenti utilizzati 49
5.2.5 Possibili estensioni del prodotto 51

Bibliografia **53**

Elenco delle figure

1.1 Esempio di Docs Plug-in 5
1.2 Schema riassuntivo del prodotto 7

2.1 Esempi di titoli di pagine doc 11
2.2 Esempio di Docs nel caso di file “index.html” mancante 12
2.3 Esempio di Docs prima e dopo l’esecuzione del goal *cleanup* 12

3.1 Diagramma dei package 25
3.2 Diagramma delle classi relativo alla gerarchia principale del plugin . . 26
3.3 Diagramma delle classi relativo al client 29
3.4 Diagramma delle classi relativo ai dettagli di ogni componente del plugin Confluence 31
3.5 Diagramma di sequenza relativo al *goal publish* 33
3.6 Diagramma di sequenza relativo al *goal cleanup* (1) 34
3.7 Diagramma di sequenza relativo al *goal cleanup* (2) 35
3.8 Diagramma di sequenza relativo al *goal cleanup* (3) 35
3.9 Esempio di un file in formato APT 40
3.10 Esempio di pagina di Usage Maven 40

4.1 Esempio di SonarQube MAJOR issue 44

Elenco delle tabelle

1.1 Obiettivi del progetto 2
1.2 Tecnologie utilizzate durante il progetto e loro scopo. 6

2.1 Elenco dei requisiti di funzionalità relativi alla configurazione 15

2.2 Elenco dei requisiti di funzionalità relativi ai messaggi di errore . . . 16
2.3 Elenco dei requisiti di funzionalità del sistema 16
2.4 Elenco dei requisiti di qualità (1) 16
2.5 Elenco dei requisiti di qualità (2) 17
2.6 Elenco dei requisiti di vincolo (1) 17
2.7 Elenco dei requisiti di vincolo (2) 18
2.8 Riepilogo dei requisiti 18
2.9 Requisiti in rapporto alle fonte “Azienda” (1) 19
2.10 Requisiti in rapporto alla fonte “Azienda” (2) 20
2.11 Requisiti in rapporto alla fonte “Interno” 21

3.1 Parametri configurabili dall’utente 27
3.2 Metodi di DocsPluginClient che compiono chiamate REST 30
3.3 Elenco delle classi 32

5.1 Obiettivi dello stage raggiunti con relativa spiegazione 48

Capitolo 1

Introduzione

1.1 Il progetto

Finantix è un'azienda di informatica che vende un prodotto software principale. Questo prodotto, per la realizzazione di applicazioni finanziarie, è suddiviso in moduli. Ognuno di questi moduli prevede una propria documentazione delle API Java (un archivio zip contente documentazione in formato JavaDoc) e la documentazione della API RESTful (un archivio zip contenente documentazione in formato Open API). Questa documentazione viene manualmente caricata sulla piattaforma Confluence, ove cui è consultata dagli sviluppatori dell'azienda.

Il plugin Maven nasce dalla necessità di automatizzare la pubblicazione di questa documentazione sulla piattaforma, in modo da semplificare e velocizzare notevolmente questo processo. Infatti, una volta configurato correttamente il plugin in tutti i progetti relativi ai moduli software, il caricamento della documentazione avviene direttamente durante la build del progetto, senza richiedere ulteriore intervento umano.

Il progetto ha avuto inizio il 10 giugno 2019 ed ha terminato il 2 agosto 2019, per un totale di 320 ore.

Durante queste 8 settimane, i prodotti attesi erano:

1. **plug-in Maven**: progetto Java il cui risultato è il plugin sopra descritto;
2. **manuale dell'utilizzatore**: documentazione del plugin Maven che illustra come attivarlo e configurarlo al fine di pubblicare la documentazione generata durante il processo di build sul sistema documentale dell'azienda;
3. **manuale del programmatore**: descrizione tecnica del plugin Maven, vale a dire, le classi principali dell'implementazione con le relative responsabilità ed il relativo funzionamento.

Per raggiungere questo scopo, sono stati fissati alcuni fondamentali obiettivi:

- studiare in maniera autonoma lo strumento Maven per poterne capire appieno l'utilità ed il funzionamento;
- comprendere come realizzare un plugin Maven, oggetto centrale del progetto;
- comprendere il paradigma RESTful, per poter trasmettere dati al server documentale;

- implementare il plugin Maven, una volta raggiunte le competenze sufficienti;
- realizzare la documentazione utente, ovvero il manuale dell'utilizzatore sopra citato, per semplificare la comprensione dell'utilizzo del plugin all'utente finale. Non è strettamente necessario all'adempimento del plugin Maven, ma auspicabile;
- realizzare la documentazione dello sviluppatore, ovvero il manuale del programmatore sopra citato, per aiutare il programmatore che dovrà fare manutenzione nella cognizione del codice. Anch'esso, come il punto precedente, non è necessario ma auspicabile;
- impiegare la tecnologia Jenkins che è usata in azienda per ogni progetto per la continuous integration. Per questo motivo, è possibile farne un minimo utilizzo, ma non è strettamente richiesto.

Tutti gli obiettivi possono essere tracciati da un codice univoco che li identifica. Ognuno di essi ha inoltre una determinata priorità scelta in base all'importanza e necessità di raggiungerli. La tabella 1.1 li riassume.

Codice	Descrizione	Priorità
O01	Acquisizione competenze su Maven	Obbligatorio
O02	Acquisizione competenze sull'implementazione di plugin Maven	Obbligatorio
O03	Familiarità con il paradigma RESTFul	Obbligatorio
O04	Implementazione di un plugin Maven	Obbligatorio
D01	Documentazione per l'utilizzatore	Desiderabile
D02	Documentazione per il programmatore	Desiderabile
F01	Utilizzo di base di Jenkins	Facoltativo

Tabella 1.1: Obiettivi del progetto

1.2 Principali problematiche

Durante il corso dello stage non sono stati riscontrati rilevanti problemi che hanno particolarmente influito sull'attività. Nonostante ciò, un problema non banale che è stato affrontato riguarda la documentazione di Maven. Molte pagine relative alla documentazione di plugin Maven infatti, risultano obsolete perché poco aggiornate. Per far fronte a questo problema, un confronto diretto e costante con gli sviluppatori senior del team DevOps, esperti della tecnologia, è stato il metodo di risoluzione determinante per il proseguimento del progetto.

1.3 Strumenti utilizzati

Gli strumenti adottati per la creazione del plugin Maven sono molteplici. Alcuni di questi sono abitualmente adoperati da tutti gli sviluppatori dell'azienda, motivo per

cui sono stati utilizzati anche per questo progetto, mentre altri sono stati liberamente scelti dalla candidata. Tra essi, selezionati per i vari motivi sotto elencati, troviamo:

- **GitKraken**: client di Git che presenta un'interfaccia grafica molto intuitiva e interattiva, oltre che semplice da usare;
- **Visual Studio Code**: editor di codice che supporta molti linguaggi, tra cui JSON e HTML [**site:visual-studio-code**];
- **SequenceDiagram.org**: strumento online che permette la creazione di diagrammi di sequenza in modo semplice e veloce grazie ad una sintassi propria;
- **ObjectAid UML Explorer**: estensione di Eclipse che permette la creazione automatica di diagrammi delle classi a partire dal codice Java;
- **Meecrowave**: framework, tra i vari consigliat dall'azienda, che permette la creazione di server velocemente [**site:meecrowave**].

1.3.1 Maven

Maven è un software per la gestione di progetti, basato sul concetto di un "project object model" (POM), che gli permette di gestire la build, il report e la documentazione di un progetto Java da un unico pezzo di informazione centrale [**site:maven-introduzione**].

Maven è basato sul concetto di un ciclo di vita della build . Ciò significa che il processo per la build e la distribuzione di un particolare artefatto (progetto) è chiaramente definito. Per la persona che realizza un progetto, ciò significa che è solamente necessario imparare un piccolo set di comandi per eseguire la build di un progetto e il POM assicurerà l'ottenimento dei risultati desiderati [**site:maven-lifecycle**]. Ci sono diversi cicli di vita possibili ed ogni ciclo di vita è suddiviso in fasi ben determinate. Per esempio, una fase di particolare rilevanza per il progetto del plugin Maven è la fase di *package*, che si occupa di prendere il codice compilato e impacchettarlo nel suo formato distribuibile, come per esempio un JAR.

Ciò che è possibile gestire con questo POM è, per esempio, la lista di dipendeze e i report dei test d'unità, incluso il coverage. Il suo aspetto è quello di un file XML che contiene le informazioni riguardanti ul progetto e la sua configurazione, usati da Maven per farne la build.

Cos'è un plugin Maven

Un plugin è un programma non autonomo che interagisce con un programma per ampliarne o estenderne le funzionalità originarie. In questo caso specifico, il programma in questione è Maven.

Essendo il progetto incentrato sulla realizzazione di un plugin Maven, esso deve avere un *goal*. Un *goal* rappresenta un compito specifico (è più fine di una fase) che contribuisce alla gestione del progetto. Può essere legato a nessuna, una o più fasi della build. Un *goal* non legato ad alcuna fase può essere eseguito al di fuori del ciclo di vita, tramite un'invocazione diretta [**site:maven-plugin**].

Quando viene eseguito un goal, Maven cerca il file "pom.xml" (il POM) nella cartella corrente, lo legge, ottiene le informazioni della configurazione e dopo di che esegue il goal.

Alcune di queste informazioni di configurazione che si possono specificare nel POM possono essere: dipendenze, plugin o goal che possono essere eseguiti, versione e

descrizione del progetto, ecc [**site:maven-pom**]. A continuazione, ne viene riportato un esempio.

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>com.thedigitalstack.maven.plugins</groupId>
        <artifactId>maven-plugins</artifactId>
        <version>1.3.1-SNAPSHOT</version>
    </parent>
    <artifactId>tds-docs-publisher-plugin</artifactId>
    <packaging>maven-plugin</packaging>

    <name>Maven Docs Publisher Plugin</name>
    <description>Maven plug-in to publish generated documentation to
    Confluence so that official documentation is always up to date with the
    binary release and easily accessible to authorized users.</description>
    ...
</project>
```

1.3.2 Confluence

Confluence è un software di collaborazione sviluppato dalla compagnia australiana Atlassian [**site:confluence**]. Esso è il principale sistema documentale dell'azienda, infatti viene usato da ogni dipendente per la consultazione di vario materiale: dalle norme aziendali, alla documentazione del codice. Ciò che è più importante, motivo per cui viene utilizzato, è il fatto che permette di raggruppare pagine correlate in uno spazio dedicato accessibile a tutti o a gruppi ristretti di persone.

Recentemente è stato introdotto a Confluence un plugin di terze parti, chiamato *Docs*. Per poter comprendere appieno il progetto del plugin Maven, è prima necessario fare luce su questa estensione di Confluence.

Il plugin Docs

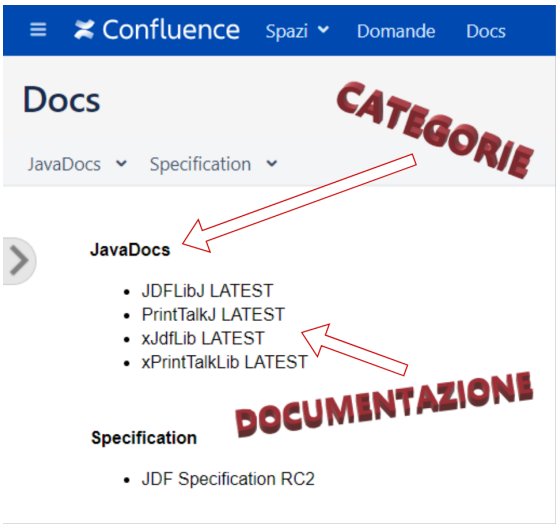

Figura 1.1: Esempio di Docs Plug-in

Come è possibile vedere dall'immagine 1.2, *Docs* è suddiviso in categorie (in questo caso "JavaDocs" e "Specification"). Le categorie presentano un nome, consentono di raggruppare la documentazione e possono essere collegate agli spazi Confluence esistenti, in modo da permettere la visione di questa documentazione solo a determinate persone. Ogni categoria quindi contiene al suo interno delle pagine web (chiamate anche *doc*) che includono la documentazione in formato HTML [**site:docs-plugin**] (come per esempio "JDF Specification RC2" per la categoria "Specification").

1.3.3 Riepilogo degli strumenti

La tabella 1.2 riportata in seguito, riassume tutti gli strumenti utilizzati e a quale scopo.

Strumento	Scopo
Java	Linguaggio di programmazione
Eclipse	Ambiente di sviluppo
Maven	Build automation per la gestione di progetti
Confluence	Pubblicazione, creazione e consultazione di documentazione
Jira	Issue tracking system
Jenkins	Continuous integration
SonarQube	Analisi statica del codice
Bitbucket e GitKraken	Controllo di versione
JUnit	Test di unità
Visual Studio Code	Editor di codice
SequenceDiagram.org	Creazione dei diagrammi di sequenza
ObjectAid UML Explorer	Creazione dei diagrammi delle classi
Meecrowave	Creazione di server

Tabella 1.2: Tecnologie utilizzate durante il progetto e loro scopo.

1.4 Il prodotto ottenuto

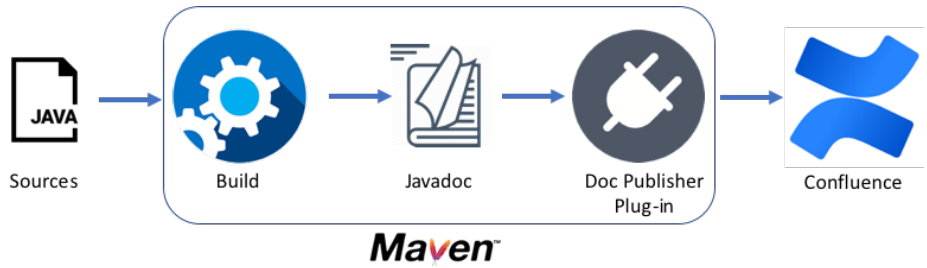

Figura 1.2: Schema riassuntivo del prodotto

Il plugin Maven riesce a realizzare il caricamento di documentazione grazie ad un plugin di terze parti su Confluence, chiamato *Docs*, descritto nella sezione §1.3.2.

Ciò che fa il plugin Maven è caricare su *Docs* in maniera automatica la documentazione nella corretta categoria, realizzando un nuovo *doc* o aggiornandone uno esistente.

Nel caso in cui il titolo fornito per la pagina *doc* non sia presente in *Docs*, verrà creata una nuova pagina, altrimenti il *doc* già esistente con quel nome verrà aggiornato con la documentazione data.

1.5 Organizzazione del testo

Il secondo capitolo comprende l'analisi dettagliata del prodotto, elencandone successivamente i relativi requisiti individuati.

Il terzo capitolo spiega la progettazione e la realizzazione del software, descrivendo le tecnologie utilizzate e l'organizzazione del codice tramite diagrammi.

Il quarto capitolo approfondisce come è stata effettuata l'attività di verifica e validazione, soffermandosi in particolare sull'analisi statica del codice e il testing.

Il quinto capitolo corrisponde al capitolo conclusivo. Esso riassume il risultato finale ottenuto e la valutazione critica del prodotto attuata.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni:

- i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*;
- tutti i concetti che possono essere riassunti, vengono raccolti in una tabella o in un elenco puntato;
- citazioni e riferimenti, provenienti da siti o libri, vengono accompagnati dal loro numero identificativo tra parentesi quadre, ad esempio [**site:junit-annotation**].

Capitolo 2

Analisi dei requisiti

Tale capitolo ha l'obiettivo di esporre e analizzare i requisiti espliciti e impliciti per la realizzazione del plugin Maven per la pubblicazione di documentazione software. L'attività di analisi ha funto da base per la fase di progettazione del software, in modo che il prodotto fosse conforme alle richieste dell'azienda.

2.1 Premessa

Il prodotto realizzato è un plugin Maven e possiede come nome ufficiale: *Maven documentation publisher plug-in*.

L'obiettivo all'inizio dello stage era il semplice caricamento di documentazione archiviata su *Docs*, per questo motivo è stato scelto di creare il goal denominato *publish*. Successivamente, nel corso dello stage, sono state aggiunte delle nuove funzionalità e un nuovo goal, dato che le tempistiche pianificate erano ottimistiche e hanno permesso sufficiente tempo per ampliare il prodotto.

2.2 Descrizione del prodotto

Maven documentation publisher plug-in supporta la pubblicazione di documentazione in formato HTML. Ha due goal:

- **publish**: che pubblica la documentazione;
- **cleanup**: che elimina la documentazione contenente SNAPSHOT nel nome.

Il principale è *publish* e si occupa della pubblicazione su *Docs* Confluence della documentazione del codice di un qualunque progetto Maven su cui è configurato il plugin. Questo è possibile perché il plugin Docs di Confluence accetta archivi, ovvero file con estensione .zip o .jar. La documentazione in questo formato può essere per esempio la documentazione Javadoc (documentazione del codice sorgente scritto in linguaggio Java) o Open API (specifica per file di interfaccia leggibili dalle macchine per descrivere servizi web RESTful, conosciuta anche come specifica Swagger [**site:specifica-openapi**]). Entrambe Javadoc e Open API sono il tipo di documentazione di maggior interesse per l'azienda da pubblicare su Docs.
Ogni archivio caricato contribuisce alla creazione di una pagina *doc* ed ogni *doc* viene identificato univocamente all'interno di una categoria, per questo motivo, il titolo

deve essere unico. Una pagina viene creata se il titolo della documentazione è nuovo, altrimenti la pagina già esistente viene semplicemente aggiornata.

2.2.1 Il goal *publish*

Maven documentation publisher plug-in è altamente configurabile, in modo da soddisfare qualunque esigenza dello sviluppatore. Innanzitutto esso consente all'utente di inserire:

- la documentazione;
- le credenziali per accedere a Confluence;
- il nome della categoria in cui allocare la documentazione.

La documentazione che fornisce l'utente può essere di tre tipi:

1. archivio (ZIP o JAR);
2. cartella (contenente più file HTML);
3. singolo file (HTML).

Nei casi 2. e 3. il plugin si occupa anche dell'archiviazione per permettere a Docs di mostrare correttamente la documentazione su Confluence.
I possibili modi per fornire le credenziali sono molteplici:

- username e password vengono date direttamente nei campi della configurazione a loro destinati;
- username e password vengono prese dalla sezione server del file ".m2/settings.xml" in cui sono salvate criptate. [1]

Non è necessario che l'utente provveda ad entrambi, ma almeno uno ci deve sempre essere. Se ci sono entrambi, il plugin utilizza il prelevamento dal file "settings.xml".

Il sistema richiede obbligatoriamente che l'utente fornisca le informazioni sopra descritte. Oltre a questi parametri però, esistono altri parametri per cui il sistema prevede una configurazione di default, ma che possono essere facoltativamente cambiati dall'utente.

Parametri opzionali

L'utente può scegliere nome e versione della documentazione. Il titolo della pagina *doc* viene costruito dall'unione di nome e versione (ad esempio dati il nome "Docs Maven Plugin" e la versione "2019", il titolo apparirà come: "Docs Maven Plugin 2019").

Nel caso in cui l'utente non fornisca nome o versione, o nessuno dei due, il sistema prevede il seguente comportamento per i due parametri:

- **nome**: preso dal nome del progetto o alternativamente dall'*artifactId* (identificativo di un progetto Maven);
- **versione**: preso dalla versione del progetto.

[1] A questo scopo, l'utente deve aver prima proceduto con la criptazione delle proprie credenziali come spiegato alla pagina `https://maven.apache.org/guides/mini/guide-encryption.html`

Vengono riportati qui di seguito alcuni esempi (in grassetto vengono evidenziati i valori dati dall'utente per la documentazione, i restanti sono altri elementi configurabili nel POM di un generico progetto Maven):

1. **nome documentazione**: Quickstart Doc, **versione documentazione**: 2019, nome progetto: Quickstart project, versione progetto: 2.1.1-SNAPSHOT, artifactId: quickstart
2. nome progetto: Quickstart project, versione progetto: 2.1.1-SNAPSHOT, artifactId: quickstart
3. **versione documentazione**: 2018, versione progetto: 2.1.1-SNAPSHOT, artifactId: quickstart

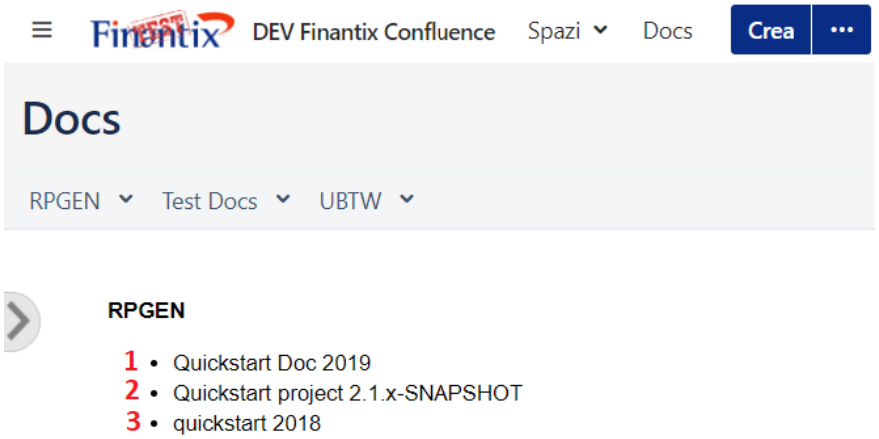

Figura 2.1: Esempi di titoli di pagine doc

Come è possibile vedere dall'immagine 2.1, nel caso 1. vengono dati dall'utente sia nome che versione della documentazione, per questo il titolo viene costruito con questi valori. Nel caso 2. invece non viene fornito nessuno dei due e per questo vengono utilizzati il nome e la versione del progetto. Nel caso 3. viene data solo la versione della documentazione, che infatti viene preferita alla versione del progetto, ma non il nome della documentazione. Poiché nemmeno il nome del progetto è configurato nel POM del progetto, viene scelto di utilizzare l'ultima opzione rimanente: l'artifactId.

L'utente può inoltre decidere di impostare:

- lo skip (salto) dell'esecuzione del plugin;
- che il plugin non fallisca nel caso in cui accada qualche errore del client;
- i tipi di progetto supportati dal plugin e quelli a cui il plugin non deve dare dei warning (messaggi di avvertimento);
- che il plugin non fallisca nel caso in cui il percorso all'archivio dato dall'utente come documentazione non esista.

Per di più, nel caso in cui la documentazione non sia di tipo archivio, bensì il percorso ad una cartella, è possibile dare delle ulteriori istruzioni:

- dove salvare l'archivio creato;
- quali tipi di file includere ed escludere dall'archivio.

Ogni archivio caricato su *Docs* plugin di Confluence deve contenere al suo interno una pagina denominata *main entrance page*, la quale essenzialmente è la prima pagina che viene visualizzata quando si entra in un *doc*. Questa è impostata di default nel plugin come "index.html". Per questo motivo è stato scelto di permettere a *Maven documentation publisher plug-in* di creare questo file qualora mancasse. Esso indirizza automaticamente al file principale della documentazione: un file scelto dall'utente tramite l'inserimento del nome in fase di configurazione.

La figura 2.2 mostra Docs nel caso di main entrance page mancante.

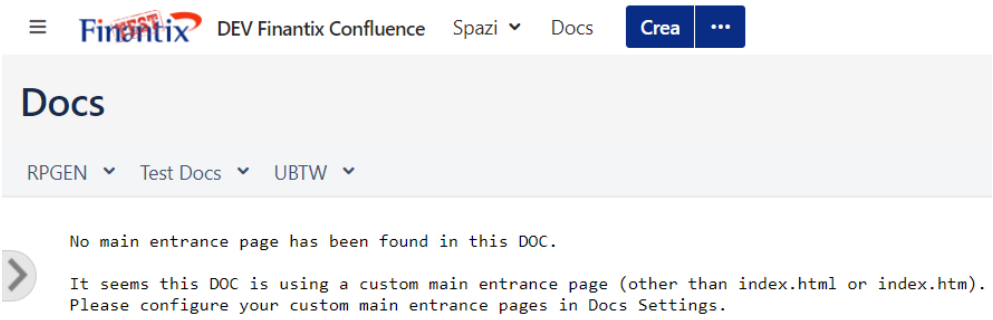

Figura 2.2: Esempio di Docs nel caso di file "index.html" mancante

2.2.2 Il goal *cleanup*

Il secondo goal, *cleanup*, è nato dalla necessità di eliminare la documentazione relativa ad un prodotto che non è stato rilasciato. Questo tipo di prodotti presentano "SNAPSHOT" nella versione e per questo motivo, anche il titolo della pagina *doc* lo contiene. Si ha quindi qui a che vedere con la pulizia totale dal plugin *Docs* di tutte queste pagine. A questo scopo non è necessario aggiungere nulla alla configurazione del plugin: sono sufficienti le credenziali dell'utente.

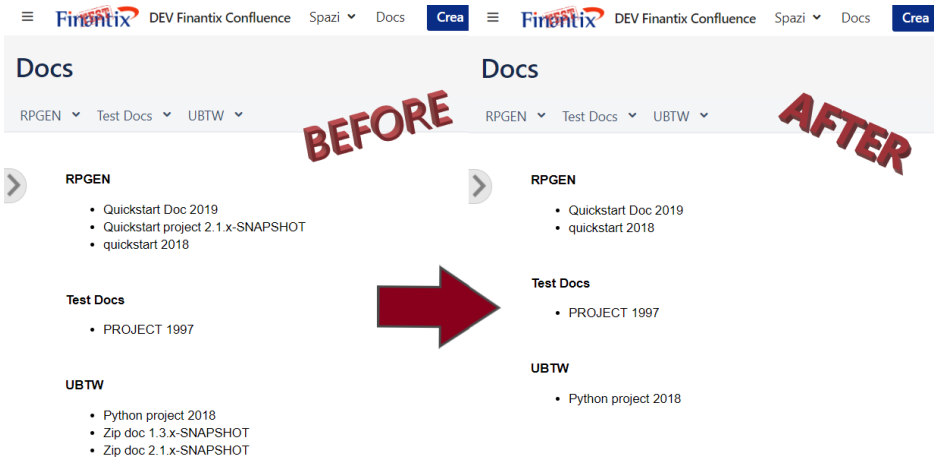

Figura 2.3: Esempio di Docs prima e dopo l'esecuzione del goal *cleanup*

2.3 Raccolta dei requisiti

Il momento dell'analisi e raccolta dei requisiti è avvenuto in congiunzione con il tutor aziendale. In principio è stato discusso dei prodotti principali da realizzare, quali plugin e documentazione. Dopodiché si è proseguiti con il dettaglio delle loro caratteristiche, come per esempio gli artefatti compresi nella documentazione e i precisi compiti del plugin. Tutti i requisiti identificati sono stati riportati in specifici ticket in Jira, lo strumento di monitoraggio delle attività di riferimento per tutti gli sviluppatori dell'azienda. Ogni ticket possiede un identificativo univoco e specifica un compito da svolgere; citiamo per esempio:

- creazione di un "hello world" (prototipo semplice) plugin Maven;
- implementazione del plugin Maven per la pubblicazione di documentazione;
- creazione manuale dello sviluppatore;
- creazione manuale per l'utente.

Ognuno di questi ticket ha un proprio flusso di lavoro predefinito o personalizzabile in base alla modalità di lavoro [**site:jira**]. In questo caso, essendo il progetto portato avanti da una persona singola sequenzialmente, il flusso comprendeva gli stati di base:

1. **TO DO**: attività da svolgere;
2. **IN PROGRESS**: attività iniziata da completare;
3. **DONE**: attività completata.

Ogni ticket viene associato ad una pagina di Confluence, appositamente creata in parallelo, che possiede una struttura ben delineata. Essa spiega nel modo più specifico possibile il ticket a cui è collegato: quale necessità ne ha spinta la creazione, che cosa ne consegue, ecc. Viene innanzitutto riportata la *user story*, ovvero la descrizione del linguaggio naturale e informale di una o più funzionalità di un sistema software [**site:user-story**], comprendente le informazioni riguardanti a: la persona che richiede che l'attività venga svolta, qual è esattamente il compito e per quale motivo deve essere svolto. Per esempio, nel caso della pagina Confluence legata al ticket della creazione del manuale utente, questa era:
As a developer
I want a guide for the Maven documentation publisher plug-in
so that can easily configure it for my Java project.
Successivamente alla user story, nella pagina Confluence gli artefatti tecnici, ovvero il dettaglio delle caratteristiche del ticket. Nel caso dell'implementazione del plugin Maven, questi erano:

- informazioni sulla repository di Bitbucket in cui deve risiedere il codice sorgente;
- informazioni sull'identificazione del plugin;
- configurazione desiderata;
- comportamento desiderato;
- scenari di test;
- goal extra;
- configurazioni extra.

2.4 Requisiti

Ad ogni requisito viene assegnato il codice identificativo univoco:

`R[Numero][Tipo][Priorità]`

in cui ogni parte ha un significato preciso:

- **R**: requisito.
- **Numero**: numero progressivo che segue una struttura gerarchica.
- **Tipo**: la la tipologia di requisito che può essere di:
  - ◇ **F**: funzionalità.
  - ◇ **Q**: qualità.
  - ◇ **V**: vincolo.
- **Priorità**: indica il grado di urgenza di un requisito di essere soddisfatto, come:
  - ◇ **0**: opzionale.
  - ◇ **1**: desiderabile.
  - ◇ **2**: obbligatorio.

Esempio: `R2Q1` indica il secondo requisito di qualità ed è desiderabile.

I requisiti individuati sono riassunti in tabelle secondo la loro tipologia. Inoltre, tutti i requisiti individuati e scelti dalla candidata, presentano nella tabella come fonte "Interno" e come grado di urgenza "0" poiché il loro soddisfacimento non è richiesto.

2.4.1 Requisiti di funzionalità

I requisiti di funzionalità sono ulteriormente suddivisi in tabelle secondo:

- requisiti relativi a ciò che il sistema permette all'utente di configurare;
- requisiti relativi a messaggi di errore che il sistema deve essere in grado di lanciare;
- requisiti relativi a generiche funzionalità del sistema.

Per ogni tabella, i primi requisiti, segnalati come obbligatori, sono i requisiti identificati all'inizio del progetto, essenziali per il funzionamento del plugin. I requisiti elencati successivamente come desiderabili sono invece le funzionalità aggiunte in un secondo momento, nel corso dello stage, come quanto accennato nella premessa all'inizio di questo capitolo. All'interno della descrizione del requisito si utilizzano alcune abbreviazioni per semplificare la lettura della tabella:

- "Inserimento" per "L'utente nella configurazione deve poter inserire";
- "Errore se" per "Il sistema deve dare un messaggio di errore qualora".

Codice	Descrizione	Fonte
R1F2	Inserimento documentazione	Azienda
R1.1F2	Inserimento archivio (ZIP o JAR)	Azienda
R1.2F1	Inserimento file HTML	Azienda
R1.3F1	Inserimento cartella	Azienda
R2F2	Inserimento credenziali	Azienda
R2.1F2	Inserimento username	Azienda
R2.2F2	Inserimento password	Azienda
R2.3F2	Inserimento identificativo server	Azienda
R3F2	Inserimento nome della categoria Confluence in cui allocare la documentazione	Azienda
R4F2	Inserimento nome della documentazione	Azienda
R5F2	Inserimento versione della documentazione	Azienda
R6F2	L'utente deve poter configurare il plugin in modo che esso ne salti la propria esecuzione	Azienda
R7F1	Inserimento luogo in cui l'archivio viene salvato all'interno del progetto	Azienda
R8F1	Inserimento tipologie di file attinenti alla documentazione	Azienda
R9F1	Inserimento tipologie di file da includere nell'archivio	Azienda
R10F1	Inserimento tipologie di file da escludere dall'archivio	Azienda
R11F1	Inserimento nome del file principale della documentazione	Azienda
R12F1	L'utente deve poter configurare il plugin in modo che esso non fallisca se avvengono errori del client	Azienda
R13F1	Inserimento tipi di progetto supportati dal plugin	Azienda
R14F1	Inserimento tipi di progetto a cui il plugin non deve dare messaggi di warning	Azienda
R15F1	L'utente deve poter configurare il plugin in modo che esso non fallisca se l'archivio dato non esiste	Azienda

Tabella 2.1: Elenco dei requisiti di funzionalità relativi alla configurazione

Codice	Descrizione	Fonte
R16F2	Errore se l'utente non fornisce nessuna documentazione	Azienda
R17F2	Errore se l'archivio dato non esiste	Azienda
R18F2	Errore se l'utente non fornisce le proprie credenziali	Azienda
R19F2	Errore se l'utente non fornisce il nome della categoria	Azienda
R20F1	Errore se il file indicato dall'utente, come file principale della cartella, non esiste	Azienda

Tabella 2.2: Elenco dei requisiti di funzionalità relativi ai messaggi di errore

Codice	Descrizione	Fonte
R21F2	Il sistema deve fornire delle proprietà per tutti gli elementi configurabili dall'utente	Azienda
R22F2	Il sistema deve essere in grado di costruire il titolo della pagina contenente la documentazione, a partire da nome e versione della documentazione	Azienda
R23F1	Il sistema deve permettere lo skip dell'esecuzione del plugin, nel caso in cui il progetto compilato non sia tra i tipi supportati	Azienda
R24F1	L'utente deve poter eliminare tutta la documentazione con versione "SNAPSHOT" presente	Azienda

Tabella 2.3: Elenco dei requisiti di funzionalità del sistema

2.4.2 Requisiti di qualità

Codice	Descrizione	Fonte
R1Q2	Le norme presenti sulla wiki aziendale devono essere rispettate	Azienda
R1.1Q2	Il nome di ogni variabile, classe e metodo nel codice deve essere significativo	Azienda
R1.2Q2	I commenti nel codice devono essere facilmente comprensibili	Azienda
R1.3Q2	Il codice non deve contenere violazioni di SonarQube con alta severità	Azienda
R1.4Q2	La copertura dei test deve essere almeno pari al 70% del codice	Azienda

Tabella 2.4: Elenco dei requisiti di qualità (1)

Codice	Descrizione	Fonte
R2Q2	Devono essere forniti test realizzati con JUnit che verifichino la copertura di tutti i parametri disponibili nella configurazione	Azienda
R3Q2	Deve essere redatto un manuale utente	Azienda
R3.1Q2	Deve essere redatta una pagina Confluence che descriva come configurare il plugin	Azienda
R3.2Q1	Deve essere redatta una pagina di utilizzo Maven “Usage” che descriva tutti i possibili utilizzi del plugin	Azienda
R4Q2	Deve essere redatto un manuale sviluppatore	Azienda
R4.1Q2	Deve essere redatta una pagina Confluence che descriva la progettazione del plugin tramite diagrammi	Azienda
R4.2Q2	Deve essere redatta e generata la documentazione Javadoc del plugin	Azienda
R5Q0	Il nome di ogni test realizzato con JUnit deve iniziare con “test”	Interno
R6Q0	Il nome di ogni test realizzato con JUnit deve rispettare la notazione camel case	Interno
R7Q0	Ogni messaggio di errore del plugin deve essere sufficientemente esplicativo	Interno

Tabella 2.5: Elenco dei requisiti di qualità (2)

2.4.3 Requisiti di vincolo

Codice	Descrizione	Fonte
R1V2	Il plugin deve essere sviluppato nel linguaggio di programmazione Java	Azienda
R2V2	Il plugin deve essere testato tramite JUnit	Azienda
R3V2	Come ambiente di sviluppo è necessario utilizzare Eclipse	Azienda
R4V2	Per la build dei progetti è necessario utilizzare Maven	Azienda
R5V2	Per la pubblicazione di documentazione è necessario utilizzare Confluence	Azienda

Tabella 2.6: Elenco dei requisiti di vincolo (1)

Codice	Descrizione	Fonte
R6V2	I requisiti identificati devono essere tracciati su Jira	Azienda
R6.1V2	Lo stato di ogni requisito presente su Jira deve sempre essere opportunamente aggiornato	Azienda
R7V2	Come strumento di continuous integration è necessario utilizzare Jenkins	Azienda
R8V2	Per l'analisi statica del codice è necessario utilizzare SonarQube	Azienda
R9V2	Per il controllo di versione del codice è necessario utilizzare Bitbucket	Azienda
R10V2	Utilizzare JUnit per realizzare test di unità	Azienda
R11V0	Utilizzare GitKraken come client di Git	Interno
R12V0	Utilizzare Visual Studio Code come editor per il codice	Interno
R13V0	Utilizzare SequenceDiagram.org per la creazione dei diagrammi di sequenza	Interno
R14V0	Utilizzare ObjectAid UML Explorer per la creazione dei diagrammi delle classi	Interno
R15V0	Utilizzare Meecrowave per la creazione di un semplice server	Interno

Tabella 2.7: Elenco dei requisiti di vincolo (2)

2.4.4 Riepilogo dei requisiti

Tipologia	Obbligatori	Desiderabili	Opzionali
Di funzionalità	16	14	0
Di qualità	11	1	3
Di vincolo	11	0	5

Tabella 2.8: Riepilogo dei requisiti

2.5 Tracciamento dei requisiti

Codice	Fonte
R1F2	Azienda
R1.1F2	Azienda
R1.2F1	Azienda
R1.3F1	Azienda
R2F2	Azienda
R2.1F2	Azienda
R2.2F2	Azienda
R2.3F2	Azienda
R3F2	Azienda
R4F2	Azienda
R5F2	Azienda
R6F2	Azienda
R7F1	Azienda
R8F1	Azienda
R9F1	Azienda
R10F1	Azienda
R11F1	Azienda
R12F1	Azienda
R13F1	Azienda
R14F1	Azienda
R15F1	Azienda
R16F2	Azienda
R17F2	Azienda
R18F2	Azienda
R19F2	Azienda
R20F1	Azienda
R21F2	Azienda

Tabella 2.9: Requisiti in rapporto alle fonte “Azienda” (1)

Codice	Fonte
R22F2	Azienda
R23F1	Azienda
R24F1	Azienda
R1Q2	Azienda
R1.1Q2	Azienda
R1.2Q2	Azienda
R1.3Q2	Azienda
R1.4Q2	Azienda
R2Q2	Azienda
R3Q2	Azienda
R3.1Q2	Azienda
R3.2Q1	Azienda
R4Q2	Azienda
R4.1Q2	Azienda
R4.2Q2	Azienda
R1V2	Azienda
R2V2	Azienda
R3V2	Azienda
R4V2	Azienda
R5V2	Azienda
R6V2	Azienda
R6.1V2	Azienda
R7V2	Azienda
R8V2	Azienda
R9V2	Azienda
R10V2	Azienda

Tabella 2.10: Requisiti in rapporto alla fonte “Azienda” (2)

Codice	Fonte
R5Q0	Interno
R6Q0	Interno
R7Q0	Interno
R11V0	Interno
R12V0	Interno
R13V0	Interno
R14V0	Interno
R15V0	Interno

Tabella 2.11: Requisiti in rapporto alla fonte “Interno”

Capitolo 3

Progettazione e realizzazione

Il capitolo corrente ha lo scopo di illustrare l'architettura del plugin nel dettaglio con il supporto di diagrammi, le sue possibili configurazioni, la sua esecuzione e la relativa documentazione.

3.1 Procedura di lavoro

Inizialmente, per capire il funzionamento dei plugin Maven e delle API RESTful del server documentale (Confluence), è stato dedicato del tempo allo studio autonomo.
Successivamente è stato sviluppato un Proof of Concept al fine di mettere in pratica quanto appreso dalla teoria. Il prototipo consisteva in un semplice plugin Maven (un "hello world") che effettuava delle stampe e delle chiamate secondo il paradigma RESTful ad un server creato al momento con Meecrowave. Questo prototipo è successivamente cresciuto ed è stato ampliato e modificato per poter interagire con il plugin Docs di Confluence, anziché il server Meecrowave. Ciò ha permesso di comprendere il caricamento di materiale su Docs ed ha consentito di effettuare la scelta delle librerie Java più adatte per il prodotto finale.
Nel corso dello svolgimento delle attività, veniva regolarmente aggiornato lo stato dei ticket Jira coinvolti. Al termine dell'implementazione del Proof of Concept per esempio, il ticket ad esso corrispondente è stato aggiornato a "DONE".

3.2 Tecnologie e librerie utilizzate

In questa sezione viene data una panoramica delle tecnologie e librerie principali utilizzate. Esse sono state scelte dalla candidata in concomitanza con il tutor aziendale e gli sviluppatori DevOps senior dell'azienda.

3.2.1 Javax

Javax è un package di estensioni standard per il linguaggio Java [**site:javax**]. Le estensioni che include sono numerose; quelle usate per la realizzazione del prodotto sono:

- **javax.annotation**: ovvero *Java Null annotation*, per le annotazioni `Nonnull` e `Nullable`, in modo da segnalare gli elementi che possono o meno essere nulli;

- **javax.ws.rs.core**: ovvero *JAX-RS*, per la creazione di risorse relative ai servizi RESTful [**site:jax-rs**], utili per il client al momento della comunicazione con Confluence;
- **javax.xml.bind.annotation**: ovvero *JAXB*, per la trasformazione automatica di JSON in oggetti Java [**site:jaxb**], utile per convertire i messaggi mandati da Confluence in oggetti facilmente manipolabili dal plugin.

3.2.2 Codehaus Plexus

Codehaus Plexus è una collezione di componenti usata da Apache Maven. Le librerie adottate per il progetto sono:

- **org.codehaus.plexus.archiver**: per l'archiviazione della documentazione;
- **org.codehaus.plexus.util**: per utilità varie, adatte per la scrittura su file.

3.2.3 Maven

Maven è la tecnologia centrale del prodotto. Di essa sono state utilizzate numerose classi, ma i package principali sono:

- **org.apache.maven.plugins.annotations**: per le annotazioni relative ai plugin Maven, quali per esempio `Mojo` per identificare un *goal*, `Parameter` per segnalare un parametro della configurazione, ecc;
- **org.apache.maven.plugin**: per le eccezioni che può lanciare un plugin Maven;
- **org.apache.maven.project**: per accedere alle informazioni del progetto (quali nome e versione);
- **org.apache.maven.settings**: per decriptare le credenziali provenienti dal file "settings.xml".

3.2.4 Jersey

Jersey è un framework opensource per lo sviluppo di servizi web RESTful in Java [**site:jersey**]. All'interno del progetto è stata una parte focale perché utilizzato per la creazione del client:

- **com.sun.jersey.api.client**: per il client che effettua le chiamate verso Confluence;
- **com.sun.jersey.api.client.config**: per la configurazione iniziale del client.

3.3 Diagramma dei package

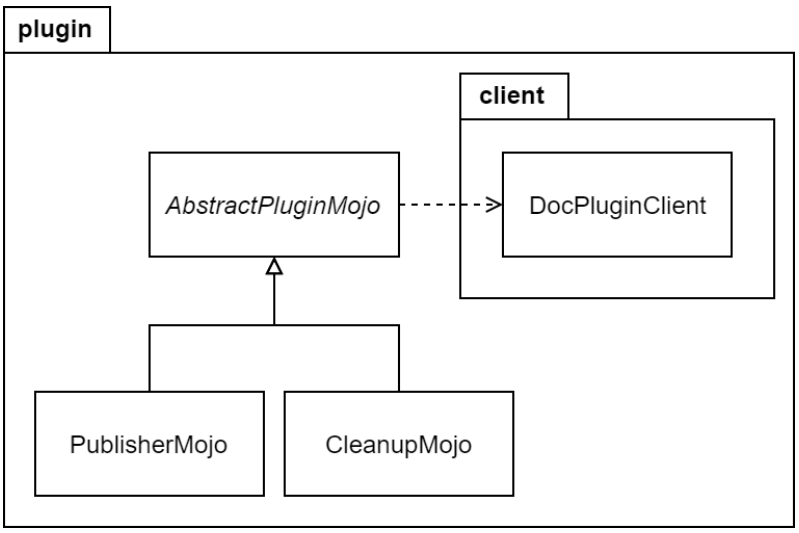

Figura 3.1: Diagramma dei package

Le classi principali di *Maven documentation publisher plug-in* sono i *mojos* di Maven e il client. Un *mojo* è un *goal* eseguibile in Maven, ovvero la classe che concretamente realizza lo scopo prefissato [**site:maven-mojo**].
I mojos appartengono al package Java `com.thedigitalstack.maven.docs.publisher.plugin` e il client al sub-package `com.thedigitalstack.maven.docs.publisher.plugin.client`. Tra loro è possibile identificare due classi fondamentali: PublisherMojo e CleanupMojo. Entrambe sono mojos di Maven e perciò rappresentano *goal* differenti.
Alcuni metodi che riguardano il client e le impostazioni del server sono uguali, per questo motivo esiste una classe base e un riferimento a DocPluginClient. DocPluginClient svolge le operazioni lato client ed è l'unico oggetto che comunica direttamente con Confluence.

3.4 Diagrammi delle classi

3.4.1 Diagramma dei mojo

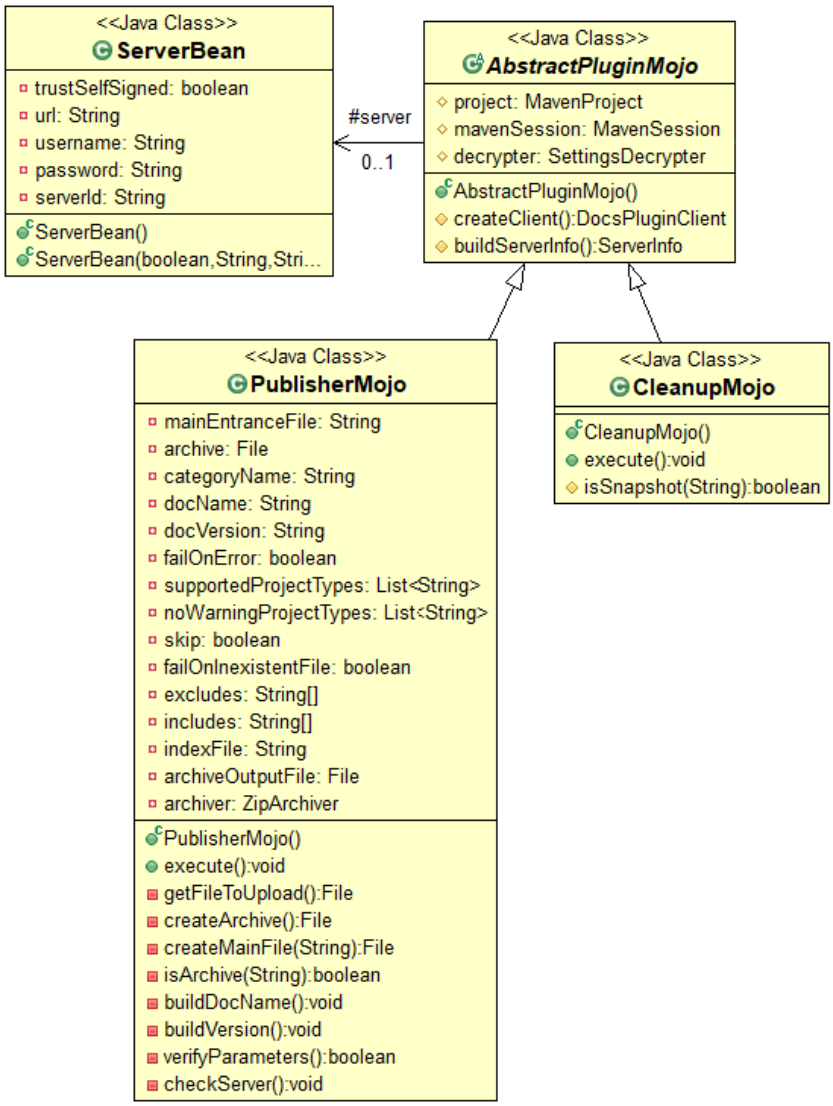

Figura 3.2: Diagramma delle classi relativo alla gerarchia principale del plugin

`PublisherMojo` realizza la pubblicazione della documentazione su Confluence, per questo motivo, il suo goal è nominato *publish* e la fase del ciclo di vita Maven a cui è legato è *package* di default.
Nella tabella 3.1 è possibile vedere tutti i parametri configurabili dall'utente che, come è possibile notare, coincidono con i campi dati di PublisherMojo visibili nella figura 3.2. Ognuno di essi è seguito da una descrizione che li definisce e dal loro valore di default.

Parametro	Descrizione	Valore di default
`archive`	Documentazione da pubblicare	-
`server`	Java bean con le informazioni del server	`trustSelfSigned`=false `url`=https://jira-dev.fx.lan/confluence/ `serverId`=my.server
`categoryName`	Nome della categoria scelta come posizione della documentazione	-
`docName`	Nome della documentazione	Nome del progetto, altrimenti l'artifactId del progetto
`docVersion`	Versione della documentazione	Versione del progetto
`failOnError`	Fallimento dell'esecuzione se si verifica qualche errore nel client	true
`supportedProjectTypes`	Lista dei tipi di progetto supportati dal plugin	jar, war, maven-plugin, eclipse-plugin
`noWarningProjectTypes`	Lista dei tipi di progetto a cui il plugin non deve sollevare warning se l'esecuzione viene saltata	pom
`skip`	Salta l'esecuzione del plugin	false
`failOnInexistentFile`	Fallisce l'esecuzione del plugin se `archive` non esiste, altrimenti salta l'esecuzione del plugin	true
`indexFile`	Nome del file HTML principale dell'archivio	index.html
`archiveOutputFile`	Percorso in cui salvare il nuovo archivio creato	${project.build.directory}/ docpublisher/archive.zip
`includes`	Lista dei file da includere nel nuovo archivio	**/**
`excludes`	Lista dei file da escludere dal nuovo archivio	**/*.git, **/*.svn, **/*.gitignore

Tabella 3.1: Parametri configurabili dall'utente

Come spiegato nella sezione §2.2.1, i primi tre parametri della tabella 3.1 sono necessariamente richiesti all'utente per l'esecuzione del goal publish, mentre i successivi sono opzionalmente configurabili e per questo motivo, presentano tutti dei valori di default utilizzabili dal sistema qualora l'utente non li specificasse.

`CleanupMojo` si occupa dell'eliminazione completa delle pagine *doc* contenenti SNAPSHOT. Ciò significa tutta la documentazione la cui versione comprende il qualificatore "-SNAPSHOT". Per questo motivo, il goal relativo si chiama *cleanup* e non è specificata nessuna fase del ciclo di vita di un progetto Maven. Esso non richiede altri parametri in uso dall'utente: fa semplicemente affidamento sul metodo `isSnapshot()` per comprendere se il titolo valutato è un "-SNAPSHOT".

PublisherMojo e CleanupMojo estendono `AbrasctPluginMojo`. Questa classe astratta estende `AbrasctMojo` (la classe astratta base di qualunque mojo Maven) e definisce i metodi in comune ad entrambi, come per esempio `createClient()` per l'inizializzazione del client, lasciando implementare il metodo `execute()`, che permette l'esecuzione del goal, alle sottoclassi.

AbstractPluginMojo fa uso di un oggetto di tipo `ServerBean`. Un Java Bean è una classe utilizzata per incapsulare più oggetti in un oggetto singolo, cosicché tali oggetti possano essere passati come un singolo oggetto bean invece che come multipli oggetti individuali [**site:java-bean**]. ServerBean infatti contiene altri oggetti relativi alle informazioni richieste per connettersi al server Confluence, come per esempio la URL e le credenziali dell'utente. Esso richiede inoltre il booleano `trustSelfSigned` per determinare se i certificati SSL sono accettati, e una stringa `serverId` nel caso l'utente volesse permettere di ricavare le credenziali dal file "settings.xml".

3.4.2 Diagramma del client

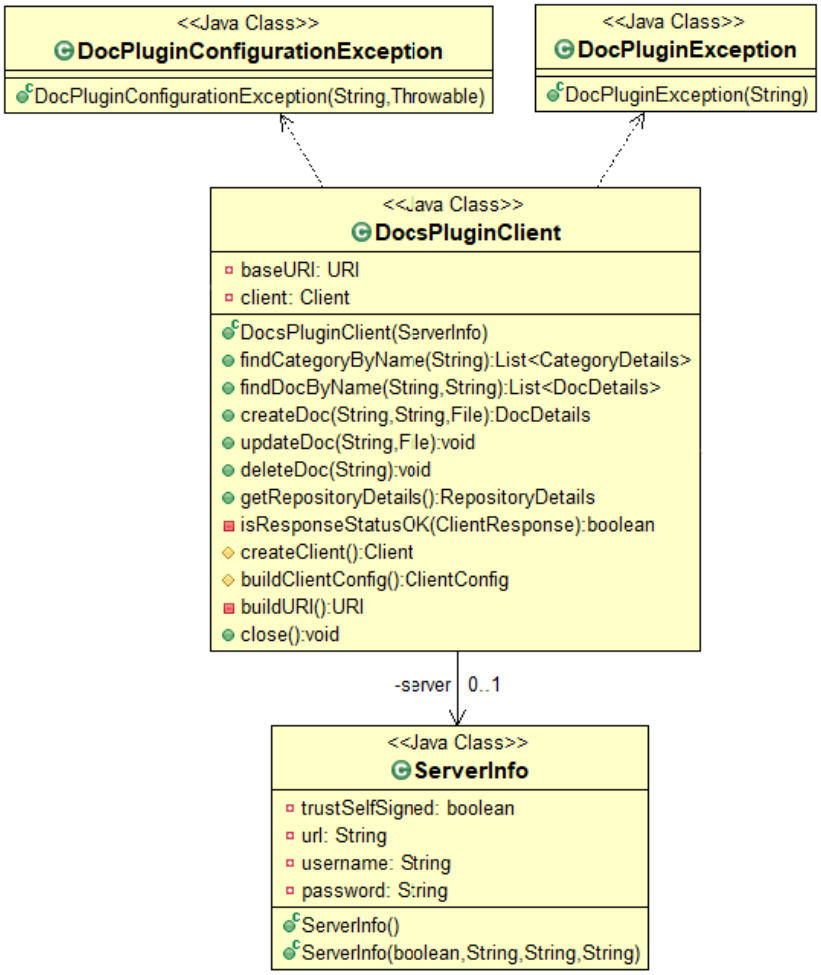

Figura 3.3: Diagramma delle classi relativo al client

Il client è creato da AbstractPluginMojo ed è un oggetto di tipo `DocsPluginClient`. Questa classe fornisce tutto il necessario per creare ed usare un'istanza del client Jersey: riceve le informazioni per la configurazione da `ServerInfo` (un semplice Java bean simile a ServerBean) e produce la directory di base richiesta per svolgere qualunque tipo di chiamata REST.

I metodi che compiono delle chiamate REST [**site:rest-docs**] sono elencati nella tabella 3.2.

Nome	Richiesta	Descrizione
`findCategoryByName(String categoryName)`	**GET**	Ritorna una lista di categorie esistenti, il cui nome coincide con la stringa data
`findDocByName(String categoryId, String docName)`	**GET**	Ritorna una lista di doc esistenti all'interno di una categoria i cui nomi coincidono con la stringa data
`createDoc(String categoryId, String docName, File docArchive)`	**PUT**	Crea la pagina doc all'interno di una categoria esistente, con l'archivio e il nome dato
`updateDoc(String docKey, File docArchive)`	**POST**	Aggiorna la pagina doc identificata dalla `docKey` data, con l'archivio dato
`getRepositoryDetails()`	**GET**	Ritorna tutti i dettagli relativi alla repository: tutte le categorie e i doc esistenti
`deleteDoc(String docKey)`	**DELETE**	Elimina la pagina doc relativa alla `docKey` data

Tabella 3.2: Metodi di DocsPluginClient che compiono chiamate REST

Molti di questi metodi possono tirare un'eccezione di tipo `DocPluginException` quando il client riceve una risposta inaspettata da Confluence in fase di comunicazione.

Un'altra eccezione che DocsPluginClient può tirare è `DocPluginConfigurationException`: un'eccezione RuntimeException che può avvenire durante la creazione del client.

3.4.3 Diagramma delle componenti del plugin Docs

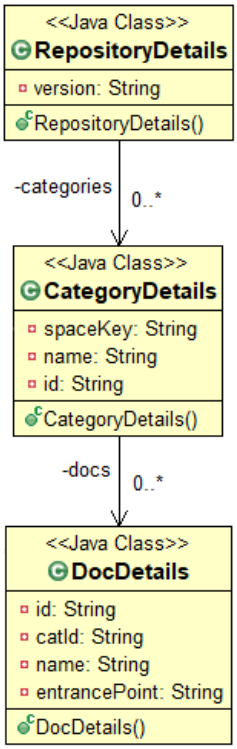

Figura 3.4: Diagramma delle classi relativo ai dettagli di ogni componente del plugin Confluence

È importante specificare che una chiamata REST, come per esempio una di quelle di tipo GET, ottiene le specifiche di una categoria o un *doc* tramite un file JSON. Questi JSON sono trasformati da JAXB in oggetti Java. A questo scopo, sono state create le seguenti classi:

- `RepositoryDetails`: informazioni sulla repository (comprende a sua volta tutte le informazioni sulle categorie e i loro *doc*);
- `CategoryDetails`: informazioni sulla categoria (comprende a sua volta tutti i *doc* contenuti);
- `DocDetails`: informazioni sulla pagina *doc*.

3.4.4 Riepilogo delle classi

Nome	Breve descrizione
`AbstractPluginMojo`	Classe astratta dei mojo del plugin
`CleanupMojo`	Classe mojo coincidente con il *goal cleanup*: elimina la documentazione "SNAPSHOT"
`PublisherMojo`	Classe mojo coincidente con il *goal publish*: pubblica la documentazione software
`DocsPluginClient`	Client del plugin che realizza chiamate REST
`RepositoryDetails`	Oggetto Java che corrisponde al JSON riguardante i dettagli della repository
`CategoryDetails`	Oggetto Java che corrisponde al JSON riguardante i dettagli di una categoria
`DocDetails`	Oggetto Java che corrisponde al JSON riguardante i dettagli di un *doc*
`ServerBean`	Java bean contenente le informazioni del sever
`DocsPluginException`	Eccezione sollevata da `DocsPluginClient` per problemi durante la comunicazione con il server
`DocsPluginConfigurationException`	RunTimeException sollevata da `DocsPluginClient` per problemi di configurazione

Tabella 3.3: Elenco delle classi

3.5 Diagrammi di sequenza

3.5.1 Diagramma del goal *publish*

Ecco come funziona la pubblicazione della documentazione:

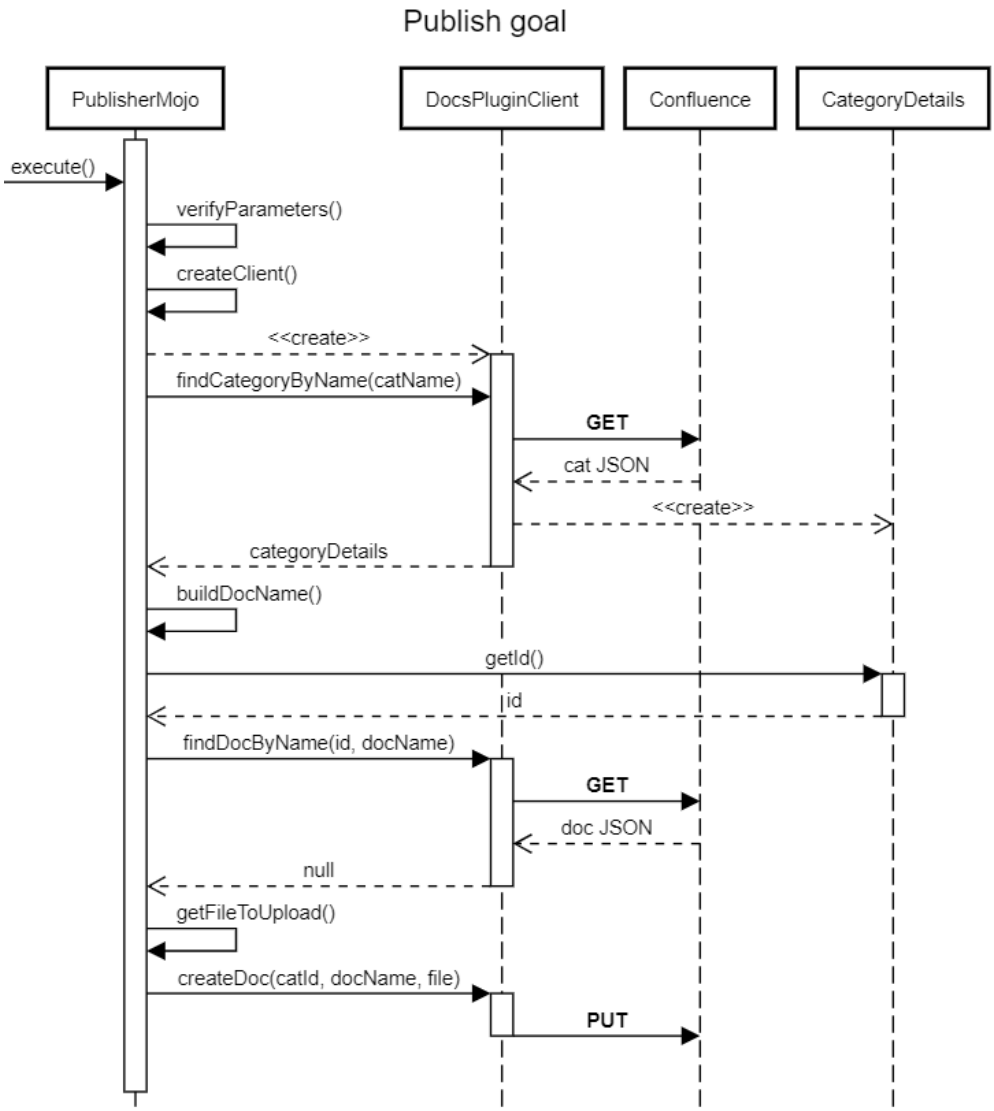

Figura 3.5: Diagramma di sequenza relativo al *goal publish*

Quando PublisherMojo viene eseguito, esso verifica la correttezza dei parametri dati e sospende l'esecuzione se necessario (per esempio se `skip` è `true`, ecc).

Dopo di che un'istanza di DocsPluginClient viene creata in modo da permettere le operazioni del client. PublisherMojo usa DocsPluginClient per ricevere i dettagli della categoria appartenente al nome dato in fase di configurazione dall'utente. Questo accade perché DocsPluginClient comunica con Confluence. Il JSON che esso riceve da

Confluence è trasformato nel relativo oggetto Java CategoryDetails.

Successivamente PublisherMojo costruisce il titolo della pagina *doc* tramite nome e versione della documentazione. In questo modo è possibile trovare il *doc* con quel determinato titolo. PublisherMojo ottiene l'id della categoria dall'oggetto precedentemente creato e richiede al client di trovare il *doc* all'interno di quella categoria.

In questo caso, il JSON ritornato sarà vuoto perché la pagina *doc* ancora non esiste. A questo punto PublisherMojo ottiene l'archivio da pubblicare, che sarà l'archivio dato dall'utente o un nuovo archivio generato.

Infine la pagina *doc* può essere creata all'interno della categoria con il nome e l'archivio selezionati.

L'aggiornamento di una pagina *doc* funziona in maniera molto similare. Le uniche differenze dallo scenario precedente si trovano dal JSON riguradante il *doc* ritornato da Confluence in poi. Esso non sarà vuoto, bensì conterrà le informazioni del *doc* esistente. A questo punto verrebbe chiamato un oggetto di tipo DocDeatails e il metodo `udpateDoc()` verrebbe chiamato al posto di `createDoc()`.

3.5.2 Diagramma del goal *cleanup*

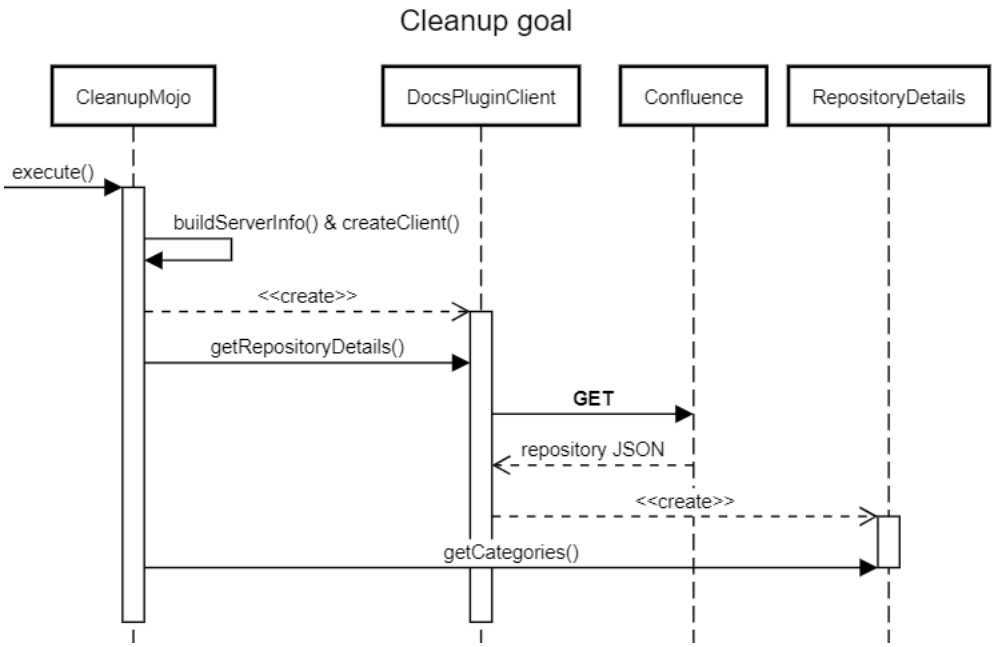

Figura 3.6: Diagramma di sequenza relativo al *goal cleanup* (1)

Prima di tutto, CleanupMojo agisce come PublisherMojo: crea un DocsPluginClient. Grazie ad esso ottiene i dettagli relativi alla repository attraverso una chiamata GET a Confluence. Il JSON ricevuto come risposta viene trasformato in un oggetto Java RepositoryDetails. CleanupMojo ricava la lista di categorie dal RepositoryDetails precedentemente creato e itera su di esso.

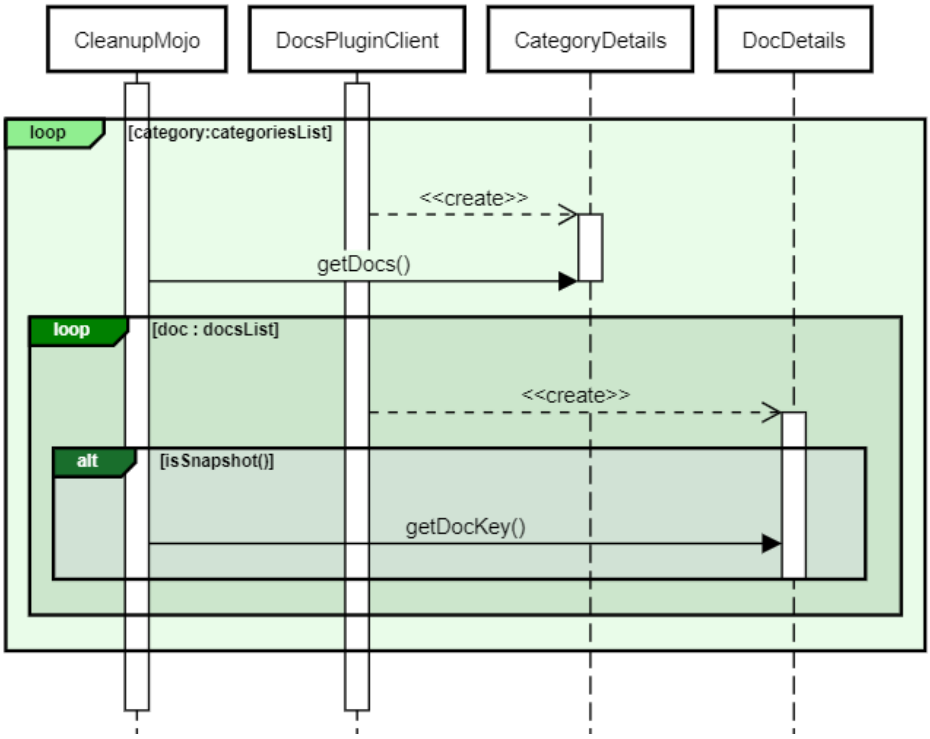

Figura 3.7: Diagramma di sequenza relativo al *goal cleanup* (2)

A questo punto richiede ad ogni CategoryDetails la sua lista di *doc*. Itera su ogni *doc* controllandone il titolo per verificare se contiene "SNAPSHOT". Se questa condizione è verificata, viene presa la chiave identificativa del *doc* (`docKey`).

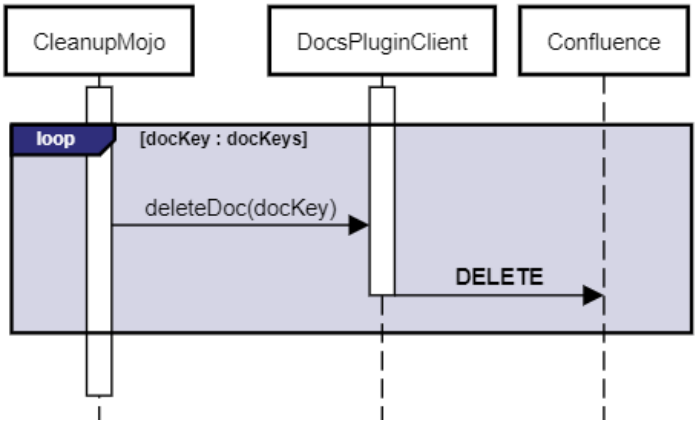

Figura 3.8: Diagramma di sequenza relativo al *goal cleanup* (3)

Infine CleanupMojo elimina ogni pagina *doc* all'interno del plugin Confluence *Docs* di cui possiede l'identificativo.

3.6 Configurazione

La configurazione è una parte predominante di Maven documentation publisher plug-in, poiché è l'unico momento un cui l'utente ha potere di decisione. Come già precedentemente spiegato nella sezione §1.3.1, tutte le informazioni di configurazione del progetto risiedono nel file "pom.xml" [**site:maven-plugin-configurazione**]. Per questo motivo, bisogna innanzitutto aggiungere il plugin al proprio progetto inserendo le informazioni che lo identificano:

```
<project>
    ...
    <build>
        ...
        <plugins>
            <plugin>
                <groupId>com.thedigitalstack.maven.plugins</groupId>
                <artifactId>tds-docs-publisher-plugin</artifactId>
                <version>1.3.1-SNAPSHOT</version>
            </plugin>
        </plugins>
    <build>
</project>
```

Dopo di che, la configurazione ideale per il plugin varia in base alla preferenza dell'utente.[1] Se il goal che si vuole eseguire è *cleanup*, è sufficiente aggiungere le informazioni del server con le proprie credenziali.

```
<plugin>
    <groupId>com.thedigitalstack.maven.plugins</groupId>
    <artifactId>tds-docs-publisher-plugin</artifactId>
    <version>1.3.1-SNAPSHOT</version>
    <configuration>
        <server>
            <trustSelfSigned>false</trustSelfSigned>
            <url>https://jira-dev.fx.lan/confluence/</url>
            <username>${env.USERNAME}</username>
            <password>${env.PASSWORD}</password>
            <serverId>my.server</serverId>
        </server>
    </configuration>
</plugin>
```

Nell'esempio sopra, sono stati dati entrambi i modi per ottenere le credenziali dell'utente: sia tramite `serverId` che tramite username e password grazie a delle variabili d'ambiente.

Se il goal che si vuole eseguire è invece *publish*, la configurazione richiede anche l'archivio della documentazione da pubblicare e il nome della categoria.

[1] Tutti gli elementi mostrati negli esempi a seguire, sono i parametri della tabella 3.1 presente nella sezione §3.4.1.

```
<plugin>
    <groupId>com.thedigitalstack.maven.plugins</groupId>
    <artifactId>tds-docs-publisher-plugin</artifactId>
    <version>1.3.1-SNAPSHOT</version>
    <configuration>
        <server>
            <trustSelfSigned>false</trustSelfSigned>
            <url>https://jira-dev.fx.lan/confluence/</url>
            <username>${env.USERNAME}</username>
            <password>${env.PASSWORD}</password>
            <serverId>my.server</serverId>
        </server>
        <archive>${project.basedir}/target/${project.artifactId}-
        ${project.version}-javadoc.jar</archive>
        <categoryName>Category</categoryName>
    </configuration>
</plugin>
```

La configurazione sopra mostrata va bene solo nel caso il cui la documentazione sia già un archivio (come in questo caso, un JAR) o un file HTML (in quel caso l'elemento `archive` conterrebbe il percorso ad il file HTML).

Nel caso in cui la documentazione fosse una cartella, è necessaria l'archiviazione, quindi la configurazione (scegliendo di tenere i valori di default dei parametri) sarebbe per esempio:

```
<configuration>
    <server>
        <trustSelfSigned>false</trustSelfSigned>
        <url>https://jira-dev.fx.lan/confluence/</url>
        <username>${env.USERNAME}</username>
        <password>${env.PASSWORD}</password>
        <serverId>my.server</serverId>
    </server>
    <categoryName>Category</categoryName>
    <archive>C:\Path\...\doc</archive>
    <archiveOutputFile>${project.build.directory}/docpublisher/
    archive.zip</archiveOutputFile>
    <includes>
        <include>**/**</include>
    </includes>
    <excludes>
        <exclude>**/*.git</exclude>
        <exclude>**/*.svn</exclude>
        <exclude>**/*.gitignore</exclude>
    </excludes>
</configuration>
```

Questo perché l'archivio verrà salvato nel percorso e col nome specificato da `archiveOutputFile`. Inoltre, gli elementi `includes` e `excludes` definiscono i file della cartella da inserire o meno nel nuovo archivio.

Nel caso in cui la documentazione data non contenesse il file "index.html", la *main entrance page* richiesta dal plugin Docs di Confluence, la configurazione sarebbe uguale

alla precedente, con l'aggiunta dell'elemento `indexFile`. Questo elemento richiede come valore il nome del file che l'utente sceglie come file principale della documentazione data, ad esempio `<indexFile>home.html</indexFile>`.

La configurazione per il titolo della pagina doc avviene tramite i parametri `docName` e `docVersion`, come:

```
<configuration>
    ...
    <docName>Docs Maven Plugin</docName>
    <docVersion>2019</docVersion>
</configuration>
```

per otteren il titolo "Docs Maven Plugin 2019".

Per quel che riguarda gli altri parametri del plugin inerenti allo skip dell'esecuzione, la configurazione (seguendo sempre i valori di default) sarebbe:

```
<configuration>
    ...
    <failOnError>true</failOnError>
    <skip>false</skip>
    <supportedProjectTypes>
        <supportedProjectType>jar</supportedProjectType>
        <supportedProjectType>war</supportedProjectType>
        <supportedProjectType>maven-plugin</supportedProjectType>
        <supportedProjectType>eclipse-plugin</supportedProjectType>
    </supportedProjectTypes>
    <noWarningProjectTypes>
        <noWarningProjectType>pom</noWarningProjectType>
    </noWarningProjectTypes>
    <failOnInexistentFile>true</failOnInexistentFile>
</configuration>
```

3.6.1 Proprietà

Ognuno di questi elementi configurabili può essere personalizzato dall'utente anche tramite l'utilizzo delle proprietà. Una proprietà è composta da `tds.docpublisher.` e il nome dell'elemento. Essa va aggiunta all'elemento `properties` del POM, come ad esempio:

```
    <project>
        ...
        <properties>
            <tds.docpublisher.skip>true</tds.docpublisher.skip>
        </properties>
        ...
    </project>
```

3.7 Esecuzione

Per eseguire il goal *publish* può essere lanciato sul progetto Maven qualunque comando che comprenda la fase di *package*, come per esempio:

```
mvn install
```

Questo perché *install* è la fase del ciclo di vita del progetto in cui Maven installa il pacchetto (JAR) nella repository locale, per poterlo usare come dipendeza in altri progetti locali [**site:maven-lifecycle**]. Lanciando questo comando quindi, vengono eseguite anche tutte la fasi precedenti.

Per il goal *cleanup* invece, non essendo legato a nessuna fase, è necessario invocare esplicitamente il goal del plugin, ovvero lanciare:

```
mvn tds-docs-publisher-plugin:cleanup
```

3.8 Documentazione

Oltre al plugin Maven, altri due prodotti sono stati realizzati per questo progetto: il manuale utente e il manuale dello sviluppatore.

3.8.1 Manuale utente

Il manuale dell'utilizzatore ha il compito di descrivere le possibili configurazioni del plugin per semplificarne l'utilizzo allo sviluppatore. Esso è suddiviso in due artefatti:

- una pagina di documentazione Confluence;
- una pagina di utilizzo Maven *Usage*.

Entrambe hanno essenzialmente lo stesso contenuto, ma la pagina Confluence è simile ad un documento Word con testo e immagini, mentre la pagina Maven di Usage presenta una struttura diversa.

Usage

Per realizzare una pagina di Usage è necessario comprendere la sintassi del linguaggio APT. APT sta per "Almost Plain Text" ed è un linguaggio di markup che è stato creato con l'obiettivo di semplificare la scrittura e la struttura della documentazione Maven [**site:apt**]. La sua sintassi assomiglia al plain-text (testo non formattato) e permette la generazione automatica di una pagina web, come mostrato dalle prossime immagini.

Il pezzo di file APT mostrato in figura 3.9 [**site:apt-file**], correttamente eseguito durante il ciclo di vita Maven denominato *site*, crea la parte di pagina HTML di Usage della figura 3.10 [**site:maven-usage**].

```
30
31  Usage
32
33   Below are the different goals and the minimalist configurations of the Help Plugin.
34
35  * The <<<help:active-profiles>>> Goal
36
37    The <<<{{{./active-profiles-mojo.html}active-profiles}}>>> goal is used to discover w
38    applied to the projects currently being built. For each project in the build session,
39    profiles which have been applied to that project, along with the source of the profil
40    or <<<profiles.xml>>>).
41
42    You can execute this goal using the following command:
43
44  +-----+
45  # mvn help:active-profiles
46  +-----+
47
48   <<Note>>: you could also use the <<<output>>> parameter to redirect output to a file.
49
50  * The <<<help:all-profiles>>> Goal
```

Figura 3.9: Esempio di un file in formato APT

Usage

Below are the different goals and the minimalist configurations of the Help Plugin.

The `help:active-profiles` Goal

The `active-profiles` goal is used to discover which profiles have been applied to the projects currently being built. For each project in the build session, it will output a list of profiles which have been applied to that project, along with the source of the profile (POM, `settings.xml` or `profiles.xml`).

You can execute this goal using the following command:

```
1. # mvn help:active-profiles
```

Note: you could also use the `output` parameter to redirect output to a file.

The `help:all-profiles` Goal

Figura 3.10: Esempio di pagina di Usage Maven

La pagina di Usage realizzata per questo progetto è simile all'esempio sopra riportato: spiega i due goal del plugin, *publish* e *cleanup*, con le stesse informazioni presenti in questo documento alla sezione §2.2, e con le istruzioni per la configurazione e l'esecuzione come esposti nelle sezioni §3.6 e §3.7.

3.8.2 Manuale sviluppatore

Il manuale del programmatore garantisce una spiegazione dettagliata delle classi create per la realizzazione del plugin Maven. Anch'esso ha richiesto la creazione di due artefatti tecnici:

- una pagina di documentazione Confluence;
- documentazione JavaDoc.

La pagina Confluence comprende la descrizione di tutte le classi con il supporto di diagrammi dei package, delle classi e di sequenza, come le sezioni §3.3, §3.4 e §3.5 di questo documento. La specifica JavaDoc consta di tutti i dettagli rilevanti di ogni frammento di codice (metodi e campi dati), con i classici tag [**site:javadoc**] quali per esempio:

- `@author`: per inserire il nome dello sviluppatore;
- `@param`: per definire i parametri di un metodo;
- `@return`: per indicare il valore di ritorno di un metodo;
- `@exception`: per indicare l'eccezione che il metodo può lanciare.

Sono state inoltre utilizzare le annotazioni di `Nonnull` per segnalare campi che non posso avere valore nullo e `Nullable` per indicare invece quali possono averlo.

Capitolo 4

Verifica e validazione

4.1 Analisi statica

L'analisi statica è stata realizzata grazie all'utilizzo di SonarQube [**site:sonarqube**] e veniva eseguita ad ogni occorrenza di una build di Jenkins. I problemi che SonarQube può segnalare, possono essere di tre tipi:

- **bug**: un errore nel codice che richiede di essere corretto il prima possibile;
- **vulnerabilità**: un punto nel codice che è aperto agli attacchi;
- **codesmell**: un problema che rende il codice confuso e difficile da manutenere.

Ognuno dei quali può avere un grado di severità differente, ordinate dalla più alla meno importante:

- **BLOCKER**
- **CRITICAL**
- **MAJOR**
- **MINOR**
- **INFO**.

La maggior parte dei problemi segnalati da SonarQube durante il progetto, erano codesmell con severità MINOR o MAJOR. La figura 4.1 ne riporta un esempio.

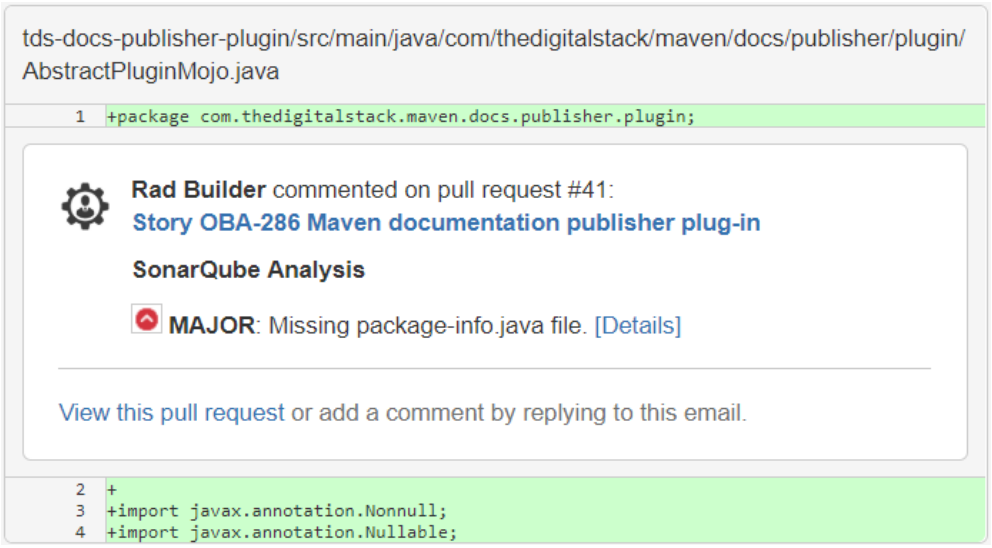

Figura 4.1: Esempio di SonarQube MAJOR issue

Per il termine del progetto, la totalità di quelli etichettati come MAJOR sono stati risolti, come richiesto dalle norme aziendali relative al codice, e anche molti dei MINOR.

4.2 Test di unità

L'attività di testing per quel che riguarda i test di unità è stata svolta utilizzando principalmente due framework: JUnit e Mockito.

4.2.1 JUnit

JUnit è il framework ideale per sviluppare i test di unità in Java. Esso fornisce notazioni [**site:junit-annotation**] essenziali quali:

- `Test` per indicare che il metodo rappresenta un test [**site:junit-test**];
- `Before` per creare gli oggetti comuni a più test prima della loro esecuzione;
- `Ignore` per ignorare temporaneamente un test.

Viene qui di seguito riportato un esempio:

```
@Test(expected = MojoExecutionException.class)
public void testCheckParametersThrowsCredentialException() throws
    Exception {
   mojo.getServer().setServerId("");
   mojo.getServer().setUsername("");

   mojo.execute();
}
```

La notazione Test viene qui usata in aggiunta al tipo di eccezione che il test si aspetta. Questo perché, in questo caso in particolare, il test verifica che venga tirata un'eccezione durante l'esecuzione del plugin, perché sia serverId che username sono stringhe vuote, quindi non c'è nessun modo per ottenere le credenziali corrette.

Sono stati utilizzati inoltre i classici metodi di `org.junit.Assert` come `assertEquals()`, `assertNull()`, `assertTrue()` ecc, utili per determinare lo stato di un test case.

4.2.2 Mockito

Mockito è un mock framework molto popolare che può essere usato in congiunzione a JUnit [**site:mockito**]. Ciò vuol dire che Mockito consente la creazione e configurazione di mock object. I mock object sono degli oggetti simulati che riproducono il comportamento degli oggetti reali in modo controllato [**site:mock**]. Un programmatore crea un oggetto mock per testare il comportamento di altri oggetti, reali, ma legati ad un oggetto inaccessibile o non implementato. Generalmente, all'interno del progetto, questi oggetti sono per esempio il client per il mojo o il server per il client. Realizzare un oggetto mock in questi casi era utile per evitare di trasformare i test di unità in test d'integrazione.

Ad esempio:

```
@Test
public void testCreateDoc() throws Exception {
    String categoryId = "c0000";
    String docName = "Documentation";
    File archive = File.createTempFile("file", "txt");
    DocDetails doc = new DocDetails();

    ClientResponse response = mock(ClientResponse.class);
    when(response.getStatus()).thenReturn(200);
    when(response.getEntity(DocDetails.class)).thenReturn(doc);
    when(builder.put(any(Class.class))).thenReturn(response);

    DocDetails returnedDoc = client
        .createDoc(categoryId, docName, archive);

    verify(httpClient).resource(new URI(
        serverInfo.getUrl() + "rest/docs/2.0/repository/" +
        categoryId + "/" + docName));
    assertNotNull(returnedDoc);
    verify(builder).put(any(Class.class));
}
```

Come suggerisce il nome, questo test verifica il comportamento del metodo `createDoc()` di DocsPluginClient. Grazie a Mockito è possibile determinare il comportamento desiderato di molti oggetti, per esempio della riposta proveniente da Confluence in seguito ad una richiesta (`ClientResponse`). È possibile infatti decidere lo stato della risposta: 200 in questo caso. In un altro test in cui si è scelti stato 401, il test dovrebbe verificare la ricezione di un errore o un'eccezione.

Oltre a decidere come un mock object si dovrebbe comportare, è possibile ad esempio verificare che un preciso metodo sia stato invocato. In questo caso, dopo che il metodo `createDoc()` è stato chiamato, viene controllato che il metodo `put()`

sia stato invocato (ultima riga del codice). Questo è di grande utilità per controllare che l'oggetto sotto test si comporti come dovrebbe, chiamando i corretti metodi degli oggetti mock.

4.3 Test di validazione

Al termine dell'implementazione del prodotto, insieme all'attività di testing, si è proceduto con la verifica di tutti i casi possibili, dal punto di vista di un qualunque sviluppatore che utilizza il plugin. Tra i casi limite, troviamo per esempio:

- pubblicazione di documentazione in una categoria in cui l'utente non ha permesso di accesso;
- pubblicazione di documentazione in una categoria inesistente;
- inserimento in configurazione del nome di un file inesistente all'interno della cartella data;
- omissione dell'utente di inserimento in configurazione delle credenziali.

In tutti questi tentativi, il prodotto si è comportato come atteso, dando dei messaggi di errore. Anche in tutte le altre circostanze previste, è stata verificata la corretta esecuzione del plugin.

Insieme agli sviluppatori senior DevOps dell'azienda e il tutor aziendale, ci si è accertati che tutte le funzionalità attese del prodotto, descritte nella sezione §2.4.1, sono state concretizzate.
In generale si è appurato che i requisiti, nella loro totalità, siano stati soddisfatti; non soltanto quindi i requisiti di funzionalità, ma anche i requisiti esplicitati nelle sezioni successive, quali requisiti di qualità e di vincolo, dagli obbligatori agli opzionali, come la stesura dei manuali e l'adozione di tutti gli strumenti elencati.

Capitolo 5

Conclusioni

5.1 Risultato ottenuto

Maven documentation publisher plug-in realizza la pubblicazione di documentazione Javadoc o Open API sul plugin Docs di Confluence grazie al goal *publish*.

Questa pubblicazione avviene in maniera automatica ogni qual volta un progetto Maven, in cui è stato configurato il plugin, attraversa la fase di *package* del suo ciclo di vita. Ciò comporta la creazione di una nuova pagina su Docs, nel caso in cui il titolo creato per la pagina a partire dai parametri dati in configurazione sia nuovo, altrimenti la pagina esistente con quel titolo viene aggiornata.

Maven documentation publisher plug-in permette inoltre di ripulire la repository dalla documentazione relativa a progetti che ancora sono sotto sviluppo e non sono stati rilasciati. Questi progetti sono marcati da Maven con il qualificatore "SNAPSHOT" e la loro documentazione può essere rimossa da Docs in qualunque momento tramite un'invocazione diretta del goal *cleanup*.

5.2 Analisi critica del prodotto e del lavoro di stage

Come prima esperienza lavorativa nell'ambito IT, il progetto di stage effettuato da Finantix è risultato ottimo e soddisfacente.
Nei primi giorni è stato un po' complicato abituarsi al lavoro in open space, perché con tante altre persone attorno è in un primo momento difficile trovare e mantenere la concentrazione. Nonostante questo, successivamente a questa fase, è emerso un ambiente di lavoro pacato e confortevole, che ha permesso il proseguimento del progetto senza problemi.
L'opportunità di gestirsi nel proprio lavoro di stage e confrontarsi con altri esperti del settore, non è mancata e anzi, si è rivelata molto interessante e stimolante.
Mettere in pratica il way of working aziendale, è stato di grande ispirazione e ha permesso la comprensione della realtà di un'azienda medio/grande come Finantix.
Una delle prove più difficili è stata pianificare il lavoro in modo da rispettare le tempistiche predefinite. Inizialmente infatti, il tempo da dedicare al progetto appariva scarso alla candidata, quasi insufficiente per tutte le nuove nozioni da apprendere. Mentre una volta ottenute la padronanza delle varie tecnologie coinvolte, è stato tutto molto più veloce di quanto appariva precedentemente. Il passo successivo e finale ha risultato nell'acquisizione della capacità di ragionare sul prodotto e valutarlo,

permettendo un raffinamento intelligente delle funzionalità con il tutor aziendale, in modo che il prodotto più si avvicinasse ad estinguere le esigenze dell'azienda.

5.2.1 Raggiungimento degli obiettivi

Nel corso del progetto, è stato inoltre appurato che gli obiettivi inizialmente fissati, visibili nella tabella 1.1 alla sezione §1.1, sono stati tutti progressivamente raggiunti. Una spiegazione più precisa che giustifica il successo di ognuno di essi è mostrato nella tabella 5.1.

Codice	Spiegazione
O01	Le competenze su Maven sono state acquisite durante il primo periodo di studio autonomo sulla documentazione di Maven
O02	Le competenze sull'implementazione di un plugin Maven sono sono state acquisite tramite studio autonomo e realizzazione del Proof of Concept
O03	La conoscenza del paradigma RESTful è stata ottenuta tramite studio autonomo, realizzazione del Proof of Concept e creazione del server con Meecrowave
O04	L'implementazione del plugin Maven ha avuto luogo in un secondo momento, successivo al Proof of Concept, grazie alla padronanza delle tecnologie ottenuta nel periodo precedente
D01	La documentazione utente è stata svolta al termine dell'implementazione del plugin con le configurazioni utilizzate nei progetti di prova che lo adoperavano
D02	La documentazione dello sviluppatore è stata svolta al termine dell'implementazione del plugin, utilizzando diagrammi UML e la specifica del codice
F01	Seppure facoltativo, è anche stato fatto un utilizzo base di Jenkins verso il termine del progetto, al momento della build finale

Tabella 5.1: Obiettivi dello stage raggiunti con relativa spiegazione

Uno strumento di grande aiuto a questo scopo è stato Jira poiché ha permesso il monitoraggio dell'andamento del progetto tramite il continuo aggiornamento dei task. Oltre a questo, anche l'iniziale sopravvalutazione della difficoltà di implementazione del plugin e della documentazione, ha contribuito al raggiungimento di tutti gli obiettivi prefissati, dando spazio anche a quelli di minor importanza: desiderabili e facoltativi.

5.2.2 Conoscenze possedute e acquisite

Le conoscenze possedute, antecedenti all'inizio del progetto, erano:

- buona conoscenza e uso intermedio di JUnit e annotazioni per i test;
- utilizzo base di Eclipse;
- conoscenza base di Maven;
- conoscenza base di Mockito;
- conoscenza base di SonarQube;
- conoscenza base della specifica JavaDoc.

Grazie a questa esperienza di stage, le conoscenze acquisite dalla candidata sono ora:

- ottima conoscenza di Maven e dei plugin Maven;
- ottima conoscenza di JUnit e miglioramento nella scrittura di test;
- ottima conoscenza della specifica JavaDoc e suo utilizzo;
- utilizzo intermedio di Eclipse;
- buona conoscenza del paradigma RESTful;
- buona conoscenza di Confluence;
- buona conoscenza di Meecrowave;
- buona conoscenza di SonarQube e risoluzione di segnalazioni;
- buona conoscenza di Mockito;
- buona conoscenza di JavaX, Codehaus Plexus e Jersey;
- conoscenza base di Jenkins;
- conoscenza base di Jira.

È quindi evidente che la candidata ha ottenuto delle nuove capacità e migliorato quelle precedentemente possedute.

5.2.3 Utilizzazione del prodotto

Maven documentation publisher plug-in non è ancora stato messo in produzione ma diventerà a breve il metodo ufficiale di pubblicazione della documentazione sul sistema aziendale Confluence.

5.2.4 Valutazione degli strumenti utilizzati

Java

Java si è rivelato il linguaggio di programmazione ideale per lo sviluppo del prodotto, in quanto fornisce molte librerie che sono state adatte alle varie esigenze. Per esempio, JAXB ha notevolmente semplificato la conversione di messaggi JSON in oggetti Java direttamente maneggiabili dai mojo.

Eclipse

Eclipse è stato un buon strumento come ambiente di sviluppo per il plugin grazie a tutte le sue integrazioni con gli altri strumenti, quali JUnit, Maven e SonarQube. Nonostante questo però, presenta alcuni bug che fanno preferire IntelliJ IDEA, usato dalla candidata in passato.

Maven

Maven è certamente eccellente per l'automazione della build di progetti e l'unico utilizzabile per lo sviluppo del prodotto. Oltre a questo, era di grande interesse per la candidata approfondirne le conoscenze.

Confluence e Jira

Confluence e Jira della suite Atlassian erano completamente sconosciuti alla candidata prima del progetto, ma la valutazione finale è positiva. Oltre che strumenti molto utili, hanno esposto delle funzionalità molto interessanti: per esempio la possibilità di collegare direttamente dei ticket (in Jira) con delle pagine di documentazione (in Confluence).

Jenkins

Di Jenkins è stato fatto solo un utilizzo base, ma si è mostrato di grande utilità al momento della build del progetto, per verificare che rispettasse tutte le varie norme aziendali.

SonarQube

SonarQube per l'analisi statica è sempre stato un ottimo strumento, anche per questo progetto, sebbene alcune segnalazioni fossero pressoché inutili o poco idonee.

BitBucket

BitBucket come strumento per il controllo di versione, non è sembrato particolarmente migliore di GitHub o Gitlab, già precedentemente utilizzati dalla candidata. Nonostante ciò, risulta coerente con la scelta dell'azienda di adottare interamente la suite di strumenti Atlassian (Confluence, Jira, ecc).

Strumenti scelti dalla candidata

Gli strumenti proposti dalla candidata, quali Visual Studio Code, GitKraken, ecc, si sono confermati comodi ed effettivamente vantaggiosi per lo scopo per la quale erano stati scelti.

5.2.5 Possibili estensioni del prodotto

Il prodotto realizzato può essere ulteriormente esteso in modo da ampliarne le funzionalità. A seguire, alcune proposte di estensione:

- **rinominazione titolo del doc**: aggiungere la possibilità di aggiornare la pagina doc della documentazione, oltre che con un nuovo archivio, rinominandone il titolo, qualvolta l'utente lo voglia, anziché creare necessariamente un nuovo doc. In questo caso, in configurazione basterebbe qualche semplice parametro in più;
- **pubblicazione di documentazione non HTML**: permettere all'utente di dare come documentazione una cartella contenente file in formato diverso dall'HTML. In questo caso, il plugin si occuperebbe non solo dell'archiviazione, ma anche della trasformazione dei file in HTML. Anche in questo caso, in configurazione potrebbe bastare l'inserimento di qualche altro nuovo parametro;
- **pulizia di altra documentazione**: estendere il goal cleanup o aggiungerne un altro che permetta l'eliminazione di documentazione secondo altri criteri (per esempio tutta la documentazione di una determinata categoria o tutta la documentazione con versione inferiore ad un certo anno).

Bibliografia