

UNIVERSITATEA „LUCIAN BLAGA” DIN SIBIU
FACULTATEA DE ȘTIINȚE

Specializarea: Informatică

LUCRARE DE LICENȚĂ

Coordonator științific
Lector univ. dr. Ralf FABIAN

Absolvent
Rodica-Lia GHIȚĂ

Sibiu
2021



UNIVERSITATEA „LUCIAN BLAGA” DIN SIBIU
FACULTATEA DE ȘTIINȚE

Specializarea: Informatică

**Reducerea timpului de implementare
prin automatizarea taskurilor generice
cu ajutorul dezvoltării de pluginuri și
extensii de IntelliJ**

Coordonator științific
Lector univ. dr. Ralf FABIAN

Absolvent
Rodica-Lia GHIȚĂ

Sibiu
2021



VIZAT

Conducător științific



Declarație pentru conformitate asupra originalității operei științifice

Subsemnatul / Subsemnata Ghiță Rodica-Lia

domiciliat/ă în localitatea Sibiu județul Sibiu

adresa Str. Plugarilor, nr. 4, Sc.C, Ap.M3,

legitimat/ă cu actul de identitate seria |S|B| nr. |7|0|6|2|0|6|, CNP |2|8|8|0|8|2|3|0|8|0|0|5|8|, înscris/ă pentru susținerea lucrării de licență / disertație cu titlul Reducerea timpului de implementare prin automatizarea taskurilor generice cu ajutorul dezvoltării de pluginuri și extensii de IntelliJ

declar următoarele:

- ideile prezentate în opera științifică sunt originale, iar sursele de informații care stau la baza emiterii unor teorii originale au fost corect citate și prezentate în opera științifică;
- opera științifică nu aparține altei persoane, instituții, entități cu care mă aflu în relații de muncă sau altă natură;
- opera științifică nu este contrară ordinii publice sau bunelor moravuri, iar prin aplicarea acesteia nu devine dăunătoare sănătății ori vieții persoanelor, animalelor sau plantelor;
- opera științifică nu a mai fost publicată de subsemnatul / subsemnata sau de o terță persoană fizică sau juridică, în țară sau în străinătate, anterior datei depunerii acesteia spre evaluare în scopul obținerii recunoașterii științifice în domeniu.

Data |1|4| / |0|6| / |2|0|2|1|

Numele și prenumele Ghiță Rodica-Lia

Semnătura 

Notă: Prezenta declarație va purta obligatoriu viza conducerului științific.

imc AG, Scheer Tower, Uni-Campus Nord, 66123 Saarbrücken

PRIVATE AND CONFIDENTIAL

TO WHOM IT MAY CONCERN

"Lucian Blaga" University of Sibiu
Computer Science Faculty
Str. Ion Rațiu, Nr.5-7,
Sibiu, 550012,
România

10th June 2021

Dear Sir/Madam

Re Lia Ghiță – Computer Science Bachelor Thesis

Our employee Lia Ghiță from imc Sibiu subsidiary is currently undertaking her bachelor thesis on the following topic:

Reducerea timpului de implementare prin automatizarea taskurilor generice cu ajutorul dezvoltării de pluginuri și extensii de IntelliJ

We hereby advise that we give our consent that our employee:

- Can use the imc logo and different screenshots from our imc learning suite product as part of her bachelor thesis to be completed in July 2021.
- Can develop the bachelor degree project tool in relation with our imc learning suite product as long as it does not include code which belongs to imc and is our intellectual property.

imc confirms that the tool will be used only internally for improving our development processes and will not be commercialized by imc, together with our learning management system.

imc
information multimedia communication AG
Scheer Tower, Uni-Campus Nord
66123 Saarbrücken
Deutschland/ Germany

Tel. +49 681 9476-0
Fax +49 681 9476-530
info@im-c.de
www.im-c.de

Vorstand/
Executive Board
Christian Wachter (Vorsitzender/ CEO)
Sven R. Becker
Dr. Wolfram Jost

Aufsichtsratsvorsitzender/
Chairman of the Supervisory Board
Prof. Dr. Dr. h.c. mult.
August-Wilhelm Scheer

Amtsgericht Saarbrücken/
Commercial Register Saarbrücken
HRB 13 338

USt-IdNr./
Vat ID no.
DE 812 187 208



imc AG, Scheer Tower, Uni-Campus Nord, 66123 Saarbrücken

Ms. Lia Ghiță will in addition, confirm on a separate document that the code she has written for her project does not include imc code that we use with our customers.

Should you have any questions, please do not hesitate to contact me on the details below:

E-mail : claire.raistrick@im-c.de

Tel +49 681 9476-315

Yours faithfully



Claire Raistrick
Snr International HR Manager



Nicolae Purcar
Managing Director, Romania

imc
information multimedia communication AG
Scheer Tower, Uni-Campus Nord
66123 Saarbrücken
Deutschland/ Germany

Tel. +49 681 9476-0
Fax +49 681 9476-530
info@im-c.de
www.im-c.de

Vorstand/
Executive Board
Christian Wachter (Vorsitzender/ CEO)
Sven R. Becker
Dr. Wolfram Jost

Aufsichtsratsvorsitzender/
Chairman of the Supervisory Board
Prof. Dr. Dr. h.c. mult.
August-Wilhelm Scheer

Amtsgericht Saarbrücken/
Commercial Register Saarbrücken
HRB 13 338

USt-IdNr./
Vat ID no.
DE 812 187 208



SELF DECLARATION

June 13th, 2021

To Whom it May Concern

I, Ghiță Rodica-Lia, hereby declare that I have not used as part of my bachelor thesis and implementation project imc code that that the company employs for commercial purposes, and which is under the company's intellectual property.

DECLARAȚIE PE PROPRIA RĂSPUNDERE

Subsemnata Ghiță Rodica-Lia, declar că nu am folosit ca parte din lucrarea și proiectul de licență cod aparținând firmei imc, folosit pentru produsele comercializate și pentru care imc deține dreptul de proprietate intelectuală.

Ghiță Rodica-Lia



Cuprins

INTRODUCERE.....	1
CAPITOLUL 1. PROBLEMA TIMPULUI DE IMPLEMENTARE.....	3
1.1. DESCRIEREA PROBLEMEI. MOTIVAREA ALEGERII TEMEI.....	3
1.1.1 Notiunea de timp.....	3
1.1.2. Metodologia Agile. Scrum.....	3
1.1.3. Notiunea de user story.....	3
1.1.4. Notiunea de estimare.....	4
1.1.4. Taskuri generice și repetitive	4
1.1.5. Diferenta dintre timpul ideal si timpul real.....	4
1.2. SCURTĂ PREZENTARE A ENTITĂILOR APLICAȚIEI PĂRINTE	5
1.2.1 Meta taguri	5
1.2.2 Configurări de sistem.....	5
1.3. ISTORIC AL INITIATIVELOR INTERNE	6
CAPITOLUL 2. DESCRIEREA SOLUȚIEI. DEZVOLTAREA UNUI PLUGIN DE INTELLIJ	7
2.1. DESCRIEREA SOLUȚIEI	7
2.2. DEZVOLTAREA UNUI PLUGIN DE INTELLIJ	8
CAPITOLUL 3. TEHNOLOGII FOLOSITE	9
3.1. JAVA	9
3.1.1 Java și Kotlin.....	9
3.1.2 Java Swing vs JavaFx	10
3.2. GRADLE	10
3.3. MYSQL	11
3.4. SERVICII WEB – REST API	11
3.5. GOOGLE GSON.....	12
CAPITOLUL 4. DESIGNUL APLICAȚIEI. FUNCȚIONALITATE	13
4.1. ACCESAREA APLICAȚIEI PRIN INTERMEDIUL MEDIULUI DE DEZVOLTARE INTELLIJ	13
4.2. MENIU.....	14
4.2.1 Meniul dedicat programatorului	14
4.2.2 Meniul dedicat administratorului	14
4.3. CREAREA UNUI META TAG.....	16
4.4. CREAREA UNEI CONFIGURĂRI DE SISTEM	18

CAPITOLUL 5. INTELLIJ PLUGIN. SETUP ȘI INSTALARE	19
CAPITOLUL 6. GENERARE DE SQL.....	24
CAPITOLUL 7. INTERACȚIUNEA CU CODUL APLICAȚIEI PĂRINTE	26
7.1. GENERAREA DE LINII DE COD ÎN CLASE JAVA	26
7.2. GENERAREA DE CLASE NOI.....	26
Probleme rezolvate pe parcursul implementării	28
CAPITOLUL 8. FOLOSIREA SERVICIILOR WEB DE REST API PENTRU INTEGRAREA CU DIVERSE APLICAȚII	30
8.1. INTEGRAREA CU JIRA PENTRU CREAREA DE SUB-TASKURI.....	33
8.2. INTEGRAREA CU APLICAȚIA DE MANAGEMENT AL LOCALIZĂRII	38
CAPITOLUL 9. ADMINISTRARE.....	42
9.1. ÎNCĂRCAREA VALORILOR PENTRU CONFIGURĂRI	43
9.2. AUTENTIFICARE	43
CONCLUZII ȘI DIRECȚII DE DEZVOLTARE	45
Concluzii	45
Direcții de dezvoltare	45
BIBLIOGRAFIE.....	46
ANEXE.....	47



Introducere

Lucrarea de față își propune să trateze tema eficientizării procesului de implementare a aplicațiilor software, prin explorarea de soluții legate de automatizarea anumitor taskuri generice și repetitive, care altfel ar consuma mult mai mult timp.

În cei peste 6 ani de experiență ca manager de produs pentru platforme de eLearning, am observat creșterea considerabilă în complexitate a aplicației principale pe care compania noastră o dezvoltă și m-am confruntat adesea cu nevoia de a specifica noi cerințe pentru produs, dar care se bazează pe un tipar de implementare ce solicită din partea programatorului anumite schimbări sau extinderi generice, repetitive, dar care solicită în mod constant aceeași cantitate de efort pentru implementare.

În contextul nevoii de dezvoltare mai rapidă și de focusare a eforturilor de implementare în inovație, mai degrabă decât în configurare sau în taskuri adiacente, am simțit nevoia de a cerceta în direcția unei soluții pentru a reduce acest timp necesar unor activități conexe, unele dintre ele încă manuale.

Nu în ultimul rând, o aplicație de complexitate crescută aduce provocări și în ceea ce privește inițierea programatorilor începători. O soluție de automatizare contribuie și la a ajuta un programator novice în a descoperi structura și modulele aplicației.

Bazată pe o arhitectură de tip monolit și cu o istorie de peste 20 de ani de dezvoltare (și aproximativ 2.000.000 de linii de cod), aplicația a devenit dificil de extins, în contextul nevoii de a fi menținută la un nivel ridicat de flexibilitate în ceea ce privește posibilitățile de customizare și adaptare la cerințele unei palete largi de clienți cu nevoi și procese diferite.

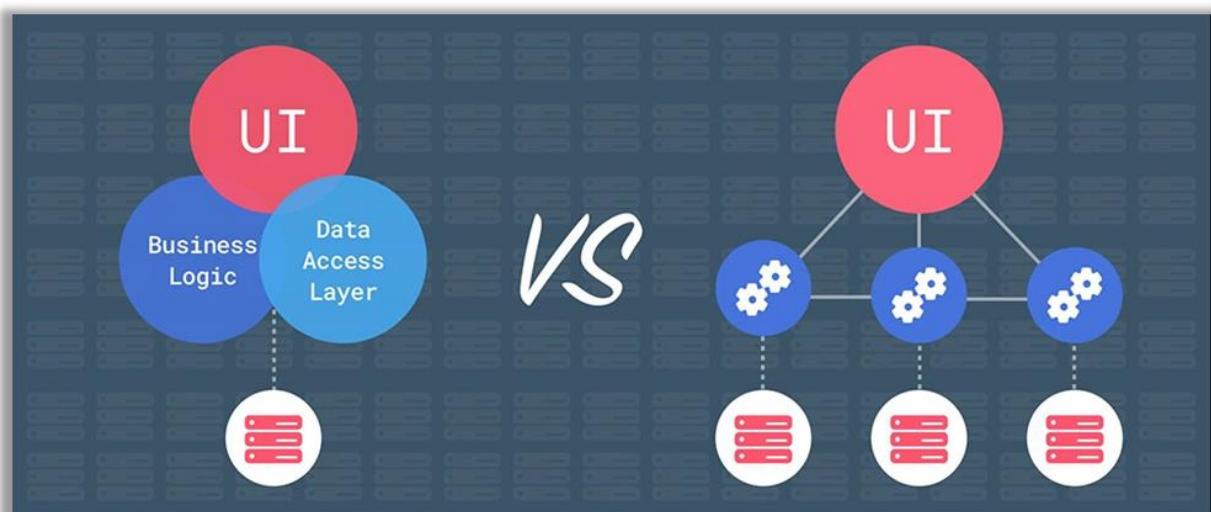


Fig. 1.1 Arhitectura monolitică vs. Arhitectura pe baza de microservicii [1]

Din punct de vedere arhitectural, s-a inițiat procesul de transformare a arhitecturii din una de tip monolit, în una bazată pe microservicii. Cu toate acestea, având în vedere că procesul de schimbare al arhitecturii este unul de durată, iar mare parte din extensiile curente se bazează încă pe core-ul existent, există nevoie unui proces de simplificare și automatizare a taskurilor generice și repetitive, pentru a putea reduce timpul necesar acestora și focusarea eforturilor către inovație.

Capitolul 1. Problema timpului de implementare

1.1. Descrierea problemei. Motivarea alegerii temei

1.1.1 Notiunea de timp

Noțiunea de timp este una fundamentală atât în domeniul fizicii, cât și a filosofiei. Fiind definit ca o măsură a duratei unui anumit eveniment [2], este un aspect pe care îl conștientizează fiecare individ ca fiind bunul cel mai valoros, ce trebuie investit cu înțelepciune, privit din prisma disponibilității acestuia.

Mai ales în ultimele secole, s-a putut observa o explozie a soluțiilor găsite pentru înlocuirea muncii manuale cu soluții automatizate, începând cu revoluția industrială din perioada dintre anii 1760 și 1840 [3] și continuând cu evoluția uimitoare a tehnologiei din ultimele câteva zeci de ani.

În ceea ce privește dezvoltarea de soluții software, se fac progrese constante în lansarea de tehnologii cât mai ușor de învățat, menite să permită un „time to market” cât mai scurt pentru noi funcționalități.

Când vorbim despre aplicații legacy, dezvoltate pe parcursul mai multor ani sau chiar zeci de ani, se ivește problema abilității acestora de a ține pasul cu noutățile de pe piață în care activează, dat fiind faptul că nu este ușor de realizat o combinație sustenabilă de tehnologii noi și vechi.

1.1.2. Metodologia Agile. Scrum

imc a adoptat metodologia Agile Scrum pentru managementul procesului de dezvoltare software pentru a îmbunătăți timpul de livrare de noi soluții, precum și pentru a crește calitatea soluțiilor oferite.

Conform Atlassian [4], Agile este o abordare iterativă a managementului de proiect și a dezvoltării de software, care ajută echipele să livreze valoare clienților lor mai repede.

1.1.3. Notiunea de user story

Când vine vorba de implementarea de noi funcționalități într-o aplicație, cerințele ce vor trebui îndeplinite prin noua funcționalitate pot fi structurate în entități numite „user story”.

O astfel de entitate este utilizată pentru a descrie funcționalitatea care va aduce valoare fie utilizatorului, fie clientului care va achiziționa soluția software. Un user story trebuie să se focuseze pe șase attribute, și anume să fie independent, negociabil, să aducă valoare utilizatorului sau clientului, să poată fi estimat, să fie de dimensiuni mici și să poată fi testabil. [5]

1.1.4. Noțiunea de estimare

Conform metodologiei Agile, există două tipuri principale de estimări: pe baza de „story points” sau pe baza de „ideal days”.

Un story point este o unitate de măsură folosită pentru exprimarea dimensiunii unui „user story”, unei noi funcționalități sau alt tip de diviziune a muncii. Estimarea prin story points constă în atribuirea unei valori fiecărui item. Numărul de story points asociat cu un anumit story reprezintă dimensiunea totală a acelui story. Nu există o formulă pentru definirea dimensiunii unui story. Un story-point este, mai degrabă, o estimare a efortului implicat în dezvoltarea funcționalității, complexitatea dezvoltării, riscul inherent, etc. Acest tip de estimare este strâns legat de noțiunea de „velocity”, care semnifică cuantificarea ratei de progres a unei echipe de dezvoltatori. [6]

Noțiunea de „timp ideal” se referă la timpul necesar îndeplinirii unei unități de muncă, atunci când eliminăm din calcul activitățile conexe, cum ar fi citirea unui e-mail, logarea în diverse aplicații, timpul de build, etc. Timpul ideal diferă de timpul real, care este timpul ce se va scurge pe ceas (sau calendar). [7]

1.1.4. Taskuri generice și repetitive

Deoarece din punct de vedere funcțional, introducerea unei configurații, fie sub formă de meta tag, fie sub formă de configurare de sistem (noțiuni descrise în continuarea acestui capitol), constituie o unitate de diviziune a muncii (task) care îndeplinește condițiile cerute de conceptul de user story, echipa de dezvoltare a avut ocazia să ofere estimări pentru o astfel de implementare în repetate ocazii.

Din datele rezultate în urma unei statistici interne, estimările au tendința să rămână constante, în ciuda faptului că pașii necesari realizării unei astfel de implementări rămân în mare măsură aceiași.

1.1.5. Diferența dintre timpul ideal și timpul real

În momentul în care diferența dintre timpul ideal și timpul real este foarte mare, în sensul în care nu implementarea soluției în sine necesită mult timp, ci activitățile conexe acesteia, se ridică întrebarea legată de optimizarea procesului de implementare și automatizarea în măsura posibilităților a activităților conexe.

1.2. Scurtă prezentare a entităților aplicației părinte

Pentru o mai bună înțelegere a utilității aplicației plugin ca sprijin pentru mediul de dezvoltare, acest subcapitol include o scurtă prezentare a entităților la a căror extindere contribuie aplicația plugin.

1.2.1 Meta taguri

Meta tagurile reprezintă tipuri de fielduri prin intermediul cărora se pot adăuga configurații sau elemente descriptive entităților precum cursuri sau programe de studiu. Aceste meta taguri se regăsesc într-un manager de unde se pot crea și astfel de elemente custom, manual. Totuși, meta taguri funcționale care impactează modul de afișare sau comportamentul entității trebuie implementate ca parte integrantă din sistemul standard.

O cerință des întâlnită din partea clientilor aplicației este introducerea unei configurații care să permită aplicarea unui anumit comportament doar anumitor cursuri. De exemplu se dorește introducerea unei bife care să ajute la a controla dacă respectivul curs permite rating din partea cursanților, sau unei configurații care să permită administratorului de curs să specifiche dacă cursul respectiv se desfășoară online sau are loc într-o sală de curs fizică. Aceste configurații vor fi introduse în sistem sub formă de meta taguri, care ulterior vor fi adăugate template-ului de curs pentru a permite apoi administratorilor să definească valoarea dorită pentru cursul creat.

1.2.2 Configurații de sistem

Fiind un sistem cu un nivel de flexibilitate crescut, aplicația necesită diverse configurații pentru a putea fi adaptată la cerințele fiecărui client sau use-case.

Astfel, un manager dedicat configurațiilor va conține secțiuni cu diverse tipuri de configurații, aplicabile diverselor tipuri de funcționalități suportate de aplicație.

Un exemplu de astfel de set de configurații poate fi cel referitor la autentificare și sesiune, unde se pot specifica atribute precum durata unei sesiuni, tipuri de autentificare permise, etc.

După cum se va vedea în capitolele ce urmează, pentru a crea un meta tag sau o astfel de secțiune de configurații, se urmează mereu un anumit set de pași sau se crează anumite tipuri de clase. Extinderea manuală a codului în clasele sau pachetele unde este necesară modificarea este relativ costisitoare în ceea ce privește timpul investit în localizarea fișierelor, modificarea lor, crearea de noi fișiere și urmarea tuturor pașilor necesari, asigurând că toate fișierele sunt adaptate corespunzător. Astfel că, prin identificarea aspectelor comune acestor activități, am creat prin intermediul aplicației plugin support o modalitate ușoară și rapidă de realizare a acestui tip de taskuri.

1.3. Iсторик al inițiativelor interne

În contextul descris anterior, ideea de automatizare nu este una tocmai nouă. Chiar și programatorii firmei s-au văzut motivați să vină cu soluții în vederea simplificării și automatizării procesului de development.

Din discuțiile cu aceștia, am identificat că au mai existat încercări de a automatiza procesul de creare a unui meta tag, însă scripturile create și folosite de programator nu au fost adoptate la nivelul departamentului, astfel că fiecare programator a încercat să își implementeze propria soluție.

Explicându-le propria-mi motivație, am încercat să identificăm împreună care ar putea fi o soluție cu șanse de a fi adoptată la nivel de departament și încorporată în procesul de development. Concluzia a fost că o astfel de soluție trebuie să fie una care să fie cât mai aproape de mediul de lucru al programatorului, ușor de accesat și de folosit și care să nu necesite instalarea sau folosirea altor programe decât cele cu care programatorul este deja obișnuit.

Capitolul 2. Descrierea soluției. Dezvoltarea unui plugin de IntelliJ

2.1. Descrierea soluției

O aplicație care să ajute în optimizarea procesului de development ar trebui să ofere, în mare, următoarele funcționalități:

- Să prezinte o modalitate de alegere a procesului pentru care programatorul necesită sprijin în implementare
- Să prezinte facilități de obținere a datelor necesare de către aplicație pentru a facilita automatizarea procesului
- Posibilitatea de output, fie de cod pre-generat, fie de părți de script și sprijinirea programatorului în a identifica modulele și fișierele din proiect unde sunt necesare modificări
- Să reducă numărul de clickuri/timpul necesar folosirii și interschimbării mai multor aplicații în procesul de implementare
- Posibilitatea de administrare pentru a putea configura aspecte ce țin de aplicații externe precum Jira și Propadmin, configurarea pachetelor destinație pentru clasele generate precum și a numelor claselor ce se cer a fi extinse în funcție de tipul de implementare ales.

Există mai multe modalități prin care aceste cerințe ar putea fi îndeplinite și, de asemenea, mai multe tipuri de aplicații.

Una dintre acestea ar putea fi dezvoltarea unei aplicații independente, fie o aplicație Windows (pornind de la premisa că majoritatea programatorilor folosesc acest sistem de operare), sau chiar și o aplicație web.

Pentru a veni în întâmpinarea programatorilor și a le ușura atât procesul de acomodare cu nivelul ridicat de complexitate a aplicației, dar și pentru a minimiza timpul necesar activităților conexe și generice cum ar fi introducerea unei noi configurații la diverse niveluri, introducerea unui nou meniu în navigația aplicației, sau alte astfel de mici schimbări necesare, este necesară o soluție cât mai apropiată de mediul de lucru al programatorului.

Având în vedere că aplicația „părinte” este o aplicație Java EE iar mediul de lucru al programatorilor, mai ales pentru partea de back-end a aplicației este IntelliJ Idea, am explorat posibilitatea unei soluții de formă unui plugin ce poate fi instalat în cadrul mediului de dezvoltare IntelliJ, aducând astfel aplicația „helper” ca parte integrantă din mediul de dezvoltare cu care programatorul este obișnuit să lucreze.

2.2. Dezvoltarea unui plugin de IntelliJ

În August 2020, JetBrains (compania care dezvoltă cel mai actual pachet de medii de dezvoltare, printre care și IntelliJ) a lansat un proiect template pentru crearea de pluginuri pentru IntelliJ, pe platforma GitHub.

Acest template a făcut accesibil și utilizatorilor începători procesul de dezvoltare a pluginurilor.

Parte din procesul care a determinat alegerea acestei modalități pentru a dezvolta aplicația dorită a constat în cercetarea capabilităților IntelliJ Platform SDK. Aici, cel mai interesant și util aspect este redat de Program Structure Interface, pe scurt PSI, care permite lucrul cu fișierele din cadrul unui proiect, modificarea acestora și crearea de fișiere noi. Având în vedere că ne dorim ca prin intermediul aplicației noastre să generăm cod, o soluție care să poată apenda fișiere existente (deschizându-le în mediul de dezvoltare) este una extrem de utilă, ajutând programatorului să observe în timp real fișierele modificate de către procesul automatizat, oferind în același timp și posibilitatea de a revizui schimbările și a le adapta.

Capitolul 3. Tehnologii folosite

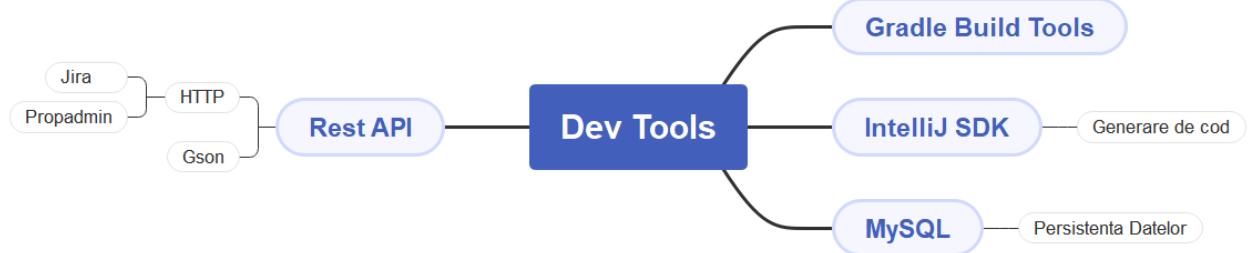


Fig. 3.1. Diagrama tehnologiilor folosite

3.1. Java

Un plugin pentru platforma IntelliJ poate fi dezvoltat în mai multe limbaje de programare, dintre care preferate sunt Java și Kotlin.

3.1.1 Java și Kotlin

Deși Kotlin aduce anumite avantaje în comparație cu Java, printre care referințele către valori nule [8], fiind un limbaj al cărui sistem de tipuri de date este axat pe eliminarea pericolului referințelor nule [9], Java este încă unul dintre cele mai utilizate limbaje de programare (Fig 3.1.)

Din acest considerent, dar și din prisma faptului că programatorii care vor lucra cu aplicația plugin sunt în mare parte programatori experimentați Java și, în consecință, cu abilitatea de a extinde ei însăși această aplicație, am considerat Java ca fiind cea mai bună tehnologie de implementare a părții de backend și logică a acestei soluții.

Pe de altă parte, există posibilitatea folosirii atât Java cât și Kotlin ca parte din același proiect, existând un nivel înalt de interoperabilitate.

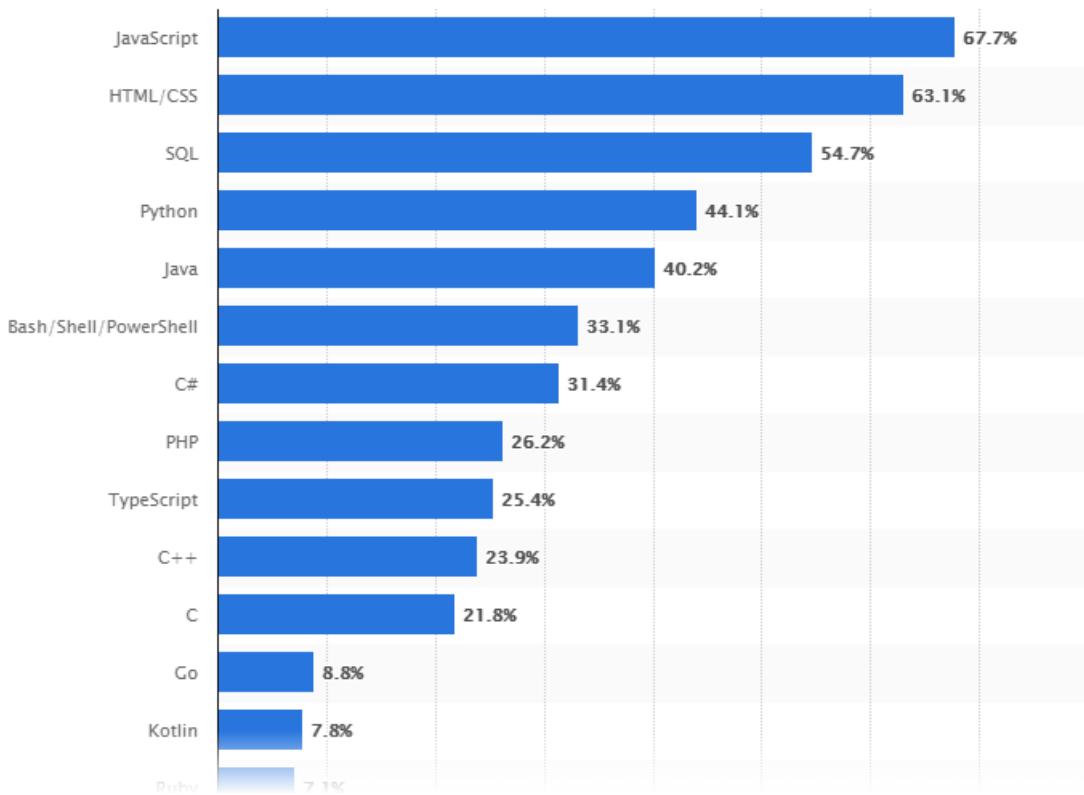


Fig. 3.1. Cele mai populare limbaje de programare [10]

3.1.2 Java Swing vs JavaFx

Când vine vorba de interfață grafică, am identificat două alternative pentru implementare, și anume Java Swing și JavaFx.

Componentele Java Swing fac parte din toolkit-ul standard Java, fiind o librărie matură și stând chiar la baza dezvoltării produselor JetBrains, inclusiv IntelliJ Idea.

JavaFx este o tehnologie care conține componente UI încă în evoluție, cu un *look and feel* mai avansat. Deși a făcut parte din Java SDK, în prezent necesită dependințe externe, iar în contextul dezvoltării unui plugin de IntelliJ s-a dovedit a fi dificil de folosit.

3.2. Gradle

Deși IntelliJ oferă posibilitatea creării unui proiect de tip Plugin fără utilizarea de aplicații dedicate pentru procesul de build, conform documentației oferite de JetBrains, aceștia recomandă crearea proiectelor de tip plugin pe baza de Gradle Build Tools. Deși pe parcursul implementării am folosit mai multe dependințe asociate direct proiectului, am constat că pentru dependința de mysqlconnector, necesară integrării cu o baza de date MySQL, singura soluție funcțională a fost declararea acestei dependințe ca și dependință Gradle.

În cadrul proiectului, folosind Gradle ca și Build Tools, fișierul de build.gradle unde sunt declarate dependințele este un fișier .kts astfel că proiectul va folosi Kotlin pentru procesul de build.

3.3. MySQL

Pentru a oferi posibilitatea stocării anumitor configurații ce țin de aspectul funcțional al aplicației plugin, am explorat diverse tipuri de bază de date.

Conform statista.com, printre cele mai populare motoare de baze de date se numără Oracle, MySQL și MSSQL.

Deși Oracle este cel mai popular tip de bază de date, având în vedere necesitatea unei baze de date comune tuturor utilizatorilor pluginului, am orientat alegerea către MySQL, dat fiind faptul că acest tip de bază de date este inclus de majoritatea platformelor care oferă hosting.

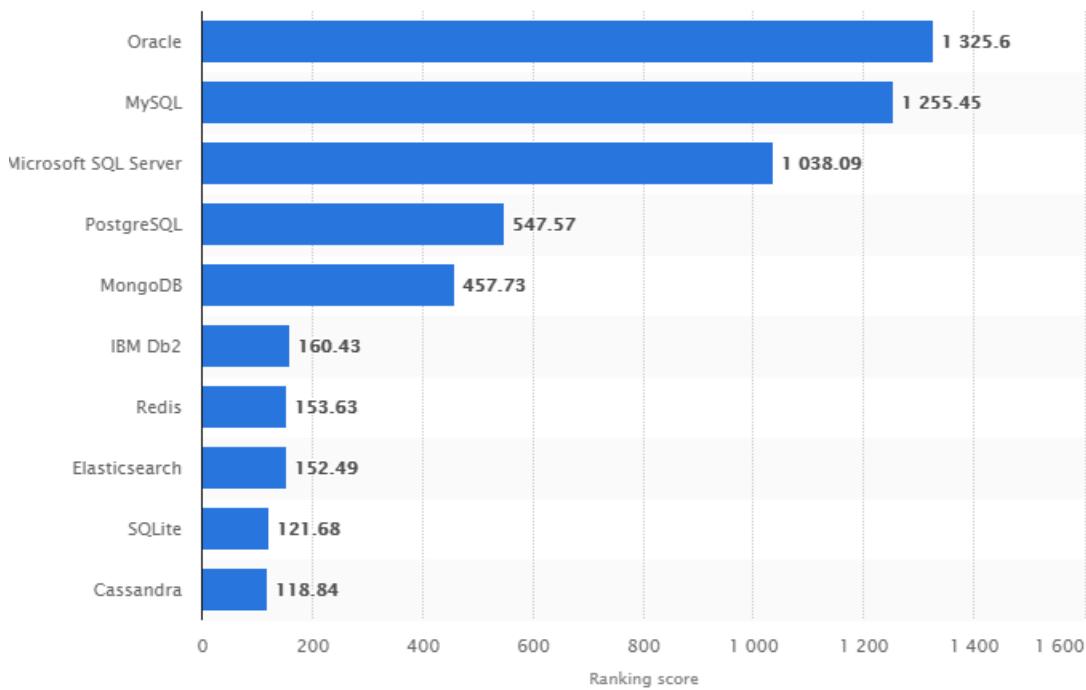


Fig. 3.2. Cele mai populare motoare de baze de date [11]

3.4. Servicii Web – REST API

Serviciile Web de tip REST presupun o comunicare între client și server pe baza unui protocol care permite transmiterea unui „payload” într-un format de tip HTML, XML sau JSON sau alt format și primirea unui răspuns care să confirme că statusul resursei s-a schimbat. [12]

Cel mai comun protocol pentru resurse REST este HTTP.

Deși Atlassian oferă Jira SDK pentru Java, din cauza dificultăților de definire a dependințelor și din dorința de a experimenta o implementare la un nivel mai jos a compunerii unui payload mai complex, am ales să implementez clasele necesare compunerii obiectului json necesar creării și assignării unui task.

3.5. Google Gson

Google Gson este o librărie Java open source care ajută la serializarea și deserializarea obiectelor Java în și din format JSON.[13]

Am utilizat această librărie pentru a putea compune payload-ul pentru cele două integrări cu Jira și Propadmin.

Capitolul 4. Designul aplicației. Funcționalitate

4.1. Accesarea aplicației prin intermediul mediului de dezvoltare IntelliJ

Un plugin de IntelliJ se poate instala fie din marketplace, fie de pe disk, acesta prezentându-se sub forma unei arhive .zip. Programatorul care dorește să îl folosească va avea astfel acces la funcționalitățile acestuia direct în mediul sau de programare.

În funcție de modul de configurare intern al pluginului, acesta se poate face disponibil direct în meniul principal al IDE-ului, sau într-un submenu.

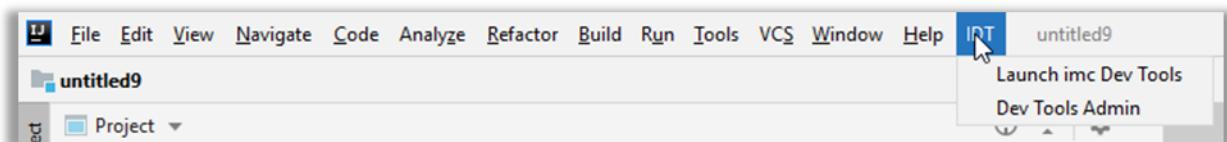


Figura 4.1. Modalitatea de accesare a aplicației

Aplicația va împrumuta tema mediului de dezvoltare pentru păstrarea consistenței.

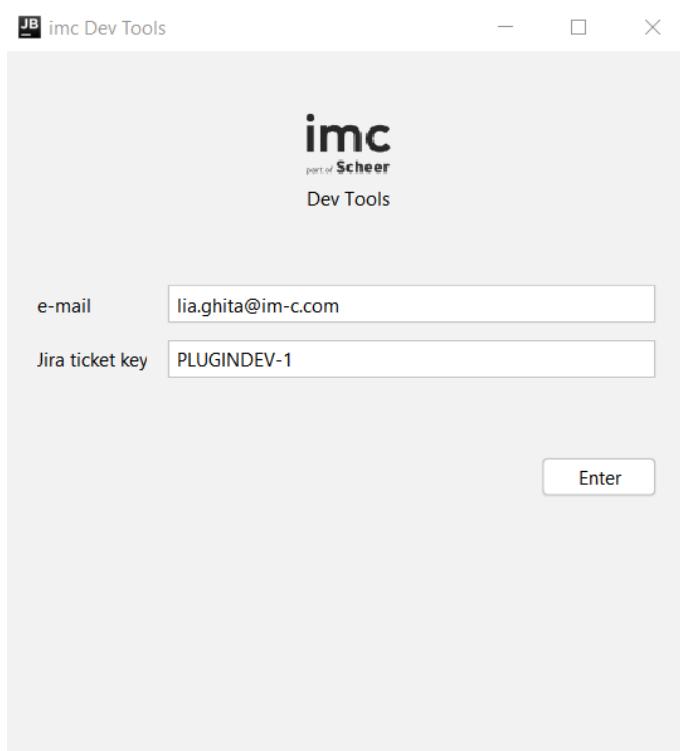


Figura 4.2. Pagina de pornire a aplicației

4.2. Meniu

Aplicația conține două meniuri principale, unul dedicat programatorului, iar celalat dedicat administratorului.

4.2.1 Meniul dedicat programatorului

Odată accesat, acest meniu va lansa pagina de intrare, unde programatorul va trebui să introducă adresa de e-mail și cheia taskului la care lucrează și pentru care necesită ajutor din partea pluginului.

De aici, utilizatorul va naviga la un meniu de unde are posibilitatea să aleagă wizard-ul pe care vrea să îl lanseze: fie cel de creare a unui nou meta tag, fie cel de creare a unei secțiuni de configurare.

De aici există de asemenea posibilitatea de autentificare ca și administrator pentru a avea acces la funcționalități de administrare.

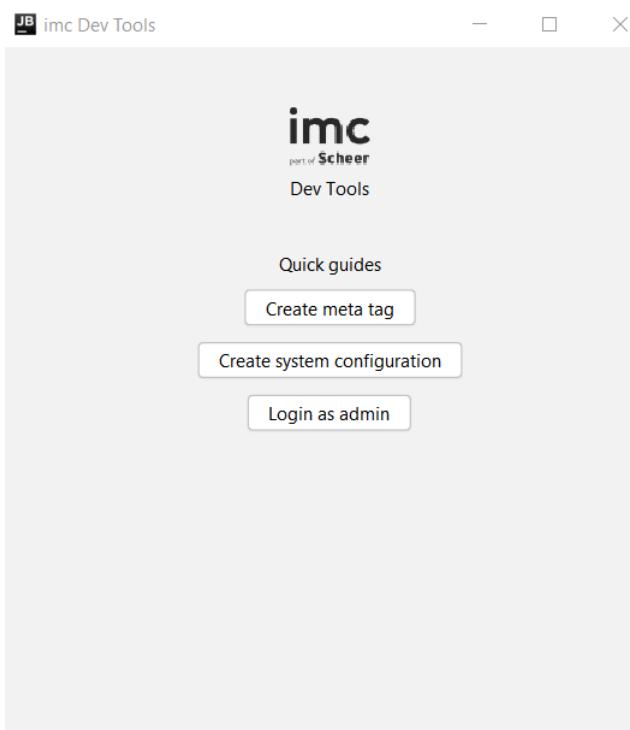


Fig. 4.3 Meniul de opțiuni disponibile pentru automatizare

4.2.2 Meniul dedicat administratorului

Deoarece aplicația necesită conexiuni cu diverse servicii web, dar și informații legate de anumite clase și pachete din aplicația părinte, este nevoie de un mechanism de configurare care permite specificarea și stocarea endpointurilor de Jira, Propadmin, denumirile claselor, etc.

Astfel, aplicația oferă un meniu de administrare care permite administratorului să se autentifice și să configureze aceste date, date care vor fi stocate într-o bază de date MySQL.

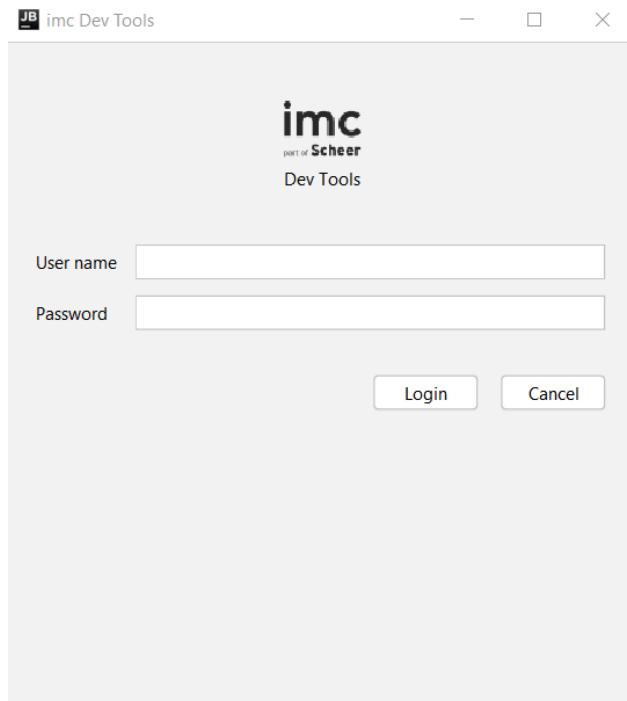


Fig. 4.4. Pagina de login a administratorului

A screenshot of the "imc Dev Tools" application window. The interface includes the "imc" logo at the top. Below it is a section for configuration parameters:

MetaTag Class	MetaTagId
Jira Rest Endpoint	https://xamplitude.atlassian.net/rest/api/2/issue/
Propadmin Rest Endpoint	?user=lia.ghita@im-c.com
Propadmin bundle endpoint	
Jira Project	PLUGINDEV
Jira sub-task id	10003

At the bottom right of the configuration area are two buttons: "Save" and "Restore".

Fig. 4.5. Administrarea configurațiilor

După autentificare, administratorul va avea acces la un formular care permite modificarea valorilor și salvarea noilor valori în baza de date. Toți utilizatorii aplicației plugin vor putea astfel beneficia imediat de schimbările aplicate de către acesta.

4.3. Crearea unui meta tag

În cadrul aplicației părinte, una dintre cele mai frecvente nevoi întâlnită în cursul implementării de funcționalități noi, este cea de a introduce o configurare care are rolul de a permite activarea sau dezactivarea acestei funcționalități.

Una dintre modalitățile prin care se poate introduce configurarea dorită este prin intermediul unui Meta tag.

În prezent, procesul implementării unui nou meta tag constă în următorii pași:

- Introducerea unui ID unic într-un fișier excel care are că și rol evidență tuturor id-urilor folosite pentru meta taguri, ținând cont de faptul că nu toate Id-urile sunt folosite în aplicația standard
- Crearea manuală a scriptului SQL de insert pentru un meta tag nou, proces destul de migălos, având în vedere mulțimea de coloane și atenția cu care trebuie mapate valorile, care în majoritate sunt de 0 și 1
- Crearea unui task sau assignarea taskului de lucru persoanei responsabile de mențenanță și optimizare a bazei de date pentru adăugarea scriptului la scriptul de inițializare al bazei de date
- Editarea unor fișiere din proiect și adăugarea unor linii de cod generice pentru extinderea listei existente de meta taguri.

În contextul acestei nevoi, aplicația plugin oferă posibilitatea creării unui astfel de meta tag în câteva click-uri. Prin ghidul rapid de creare a unui meta tag, programatorul va completa câteva detalii despre noul meta tag (Fig.3.4). În urma completării meta tagului, scriptul SQL de insert va fi generat automat, cu datele incluse în cadrul formularului. (Fig. 3.5)

Programatorul are posibilitatea să:

- Creeze, în aplicația care se ocupă de localizare și traducere a textelor, a textelor care aparțin noului meta tag: numele și descrierea acestuia
- Creeze un sub-task în Jira (Sistemul de monitorizare a taskurilor). Această opțiune va copia automat scriptul de insert în descrierea ticketului și îl va assigna persoanei responsabile
- Apendeze în mod automat fișierele relevante din proiect unde e necesară introducerea informațiilor despre noul meta tag.

JB imc Dev Tools

Back

Name	<input type="text"/>				
Description	<input type="text"/>				
Type	<input type="text" value="Text"/>				
Id	<input type="text"/>				
Is system item	<input type="text" value="Yes"/>	Use for courses	<input type="text" value="Yes"/>	Use for media	<input type="text" value="Yes"/>
Multi Language	<input type="text" value="Yes"/>	Use for learning paths	<input type="text" value="Yes"/>		

Figura 4.4. Formular de creare a unui meta tag

JB imc Dev Tools


 part of Scheer
 Dev Tools

Back

SQL query

```
insert into metatag (metatag_id, language_id, creator, creator_id, creationdate, name, active,
description, description2, formelementtype_id, required_tag, systemitem, useforcourse, useformedia,
useforcommunity, useforservice, useforprogram, useforresource, lastupdated, lastupdated_id,
useforexercise, useforexercisegroup, useforexercisesheet, useforexercise, multilang) values (10001dbsLANG,
'Learning Suite System', 0, '2021-06-01 00:00:00.000', 1, '', 1, 0, 1, 1, 0, 0, 0, 1, 0,
'2021-06-01 00:00:00.000', 0, 0, 0, 0, 1);
```

Figura 4.5. Opțiuni disponibile în contextul creării unui nou meta tag

4.4. Crearea unei configurări de sistem

Prin intermediul wizard-ului de creare a unei configurații de sistem, programatorul va defini numele secțiunii și configurațiile dorite.

Toate configurațiile adăugate deja vor fi afișate într-o listă, de unde programatorul are posibilitatea să steargă, dacă este nevoie, configurațiile adăugate din greșală.

Butonul „Add section” va deveni activ doar dacă există cel puțin o configurație adăugată în listă.

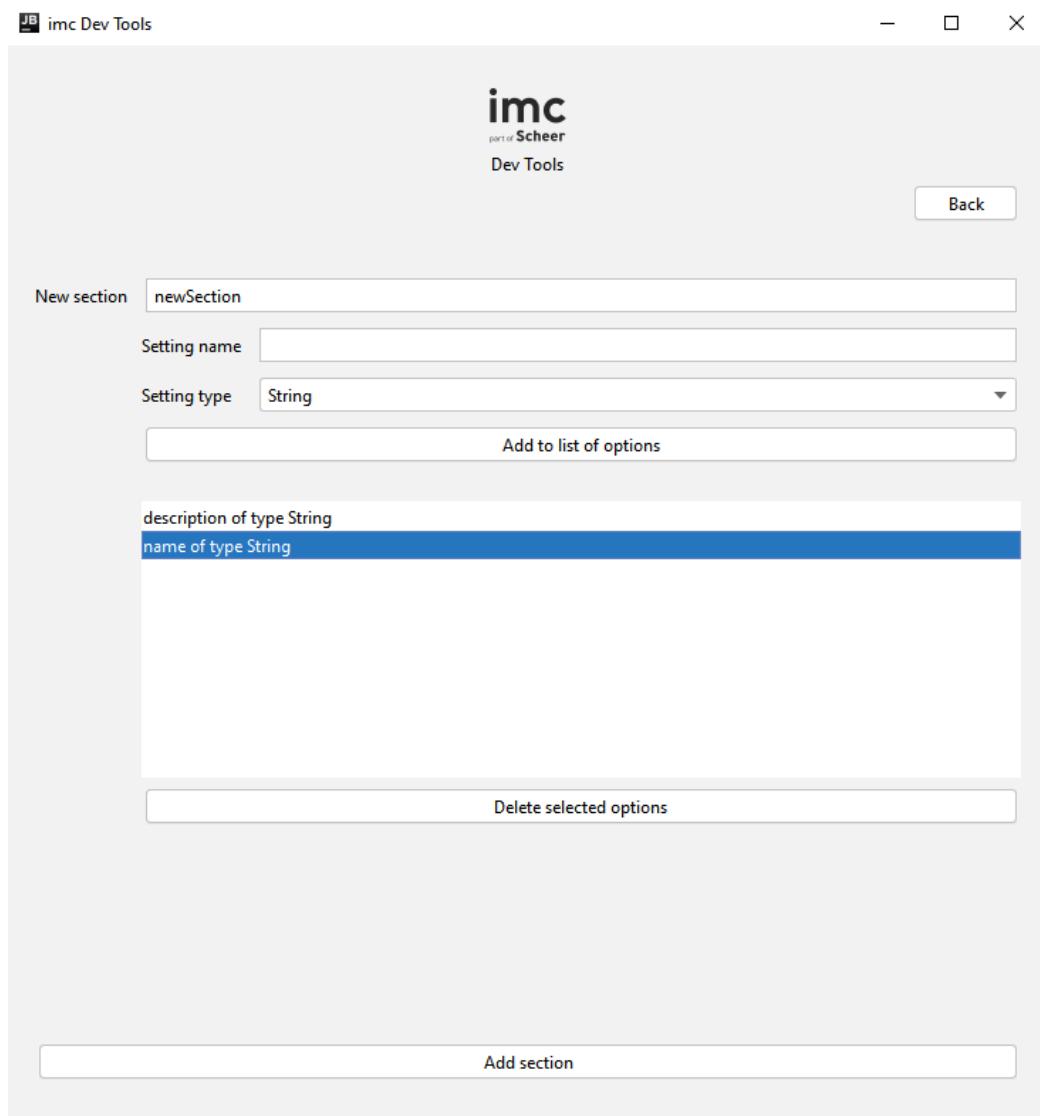


Fig 3.6 Meniul de creare a unei noi secțiuni de configurații

Prin selectarea opțiunii “Add selection”, aplicația va genera clasele necesare pentru a extinde lista de configurații existente.

Pentru a realiza acest lucru am folosit IntelliJ SDK și în principiu PSI (Program Structure interface), care oferă posibilități de navigare în proiectul curent și generare de clase și fișiere noi.

Capitolul 5. IntelliJ Plugin. Setup și instalare

Orice plugin pentru mediul de dezvoltare IntelliJ trebuie să conțină un fișier numit plugin.xml.

În acest fișier se definește modalitatea de acces a aplicației, precum și clasa de start.

```
<actions>
  <group id="PLUGIN>HelloAction" text="IDT" description="IDT">
    <add-to-group group-id="MainMenu" anchor="last" />
    <action class="LaunchDevTools" id="Plugin.Action.LaunchDevTools" text="Launch imc Dev Tools" />
    <action class="LaunchAdminMode" id="Plugin.Action.AdminPanel" text="Dev Tools Admin" />
  </group>
</actions>
```

Aici, group-id specifică id-ul grupului de meniuri unde dorim să adăugăm acțiunile noastre. „MainMenu” reprezintă meniul principal al IntelliJ.

Cele două secțiuni de tip „action class” ajută la specificarea numelui clasei, un id unic și textul afișat pentru acțiunile introduse.

Pentru prima acțiune, cea de lansare a aplicației destinată programatorului, clasa de pornire este LaunchDevTools, iar cea pentru secțiunea de administrare este LaunchAdminMode.

În plus, pe lângă aceste informații, fișierul plugin.xml trebuie să conțină numele și descrierea pluginului, precum și “vânzătorul”.

Rulând comanda Gradle de ‘assemble’, vom obține fișierul instalabil al pluginului.

Pentru a putea instala plugin-ul, navigăm la meniul Settings -> Plugins și selectăm opțiunea de Install Plugin from Disk.

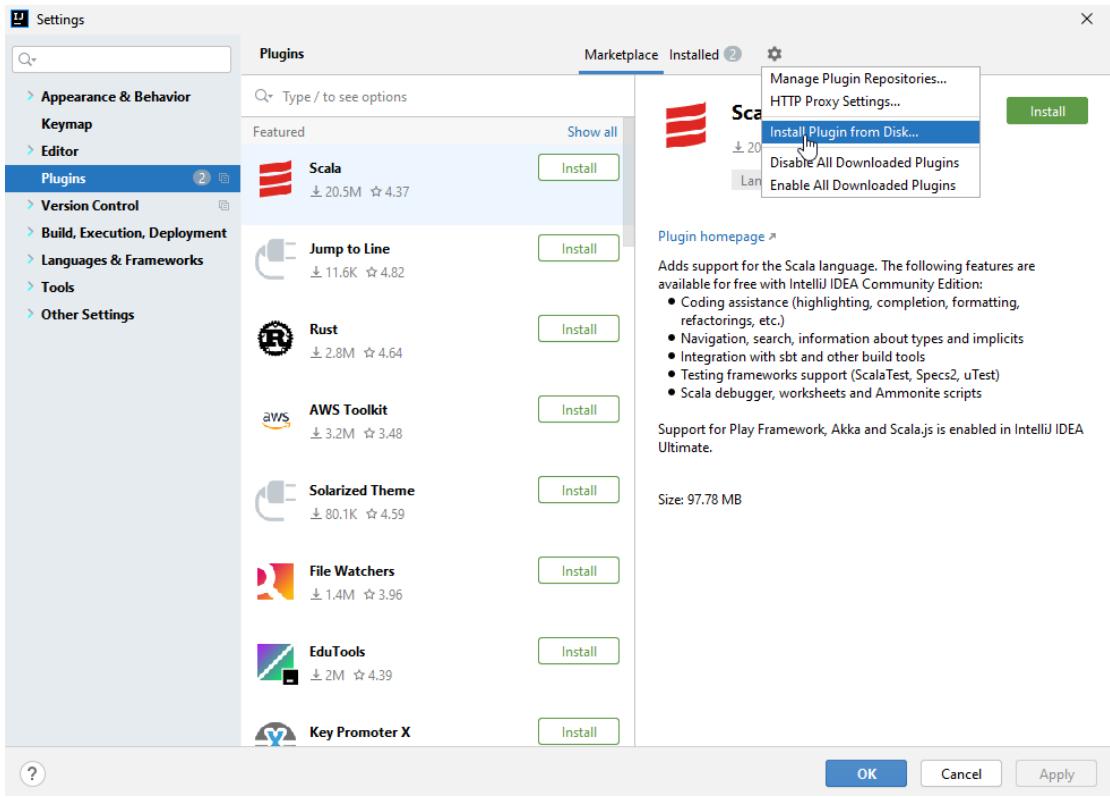


Fig. 5.1 Instalarea de pe disk a unui plugin în IntelliJ Idea

De aici, vom naviga către folderul proiectului, iar din folderul build -> distributions, vom selecta arhiva .zip generată.

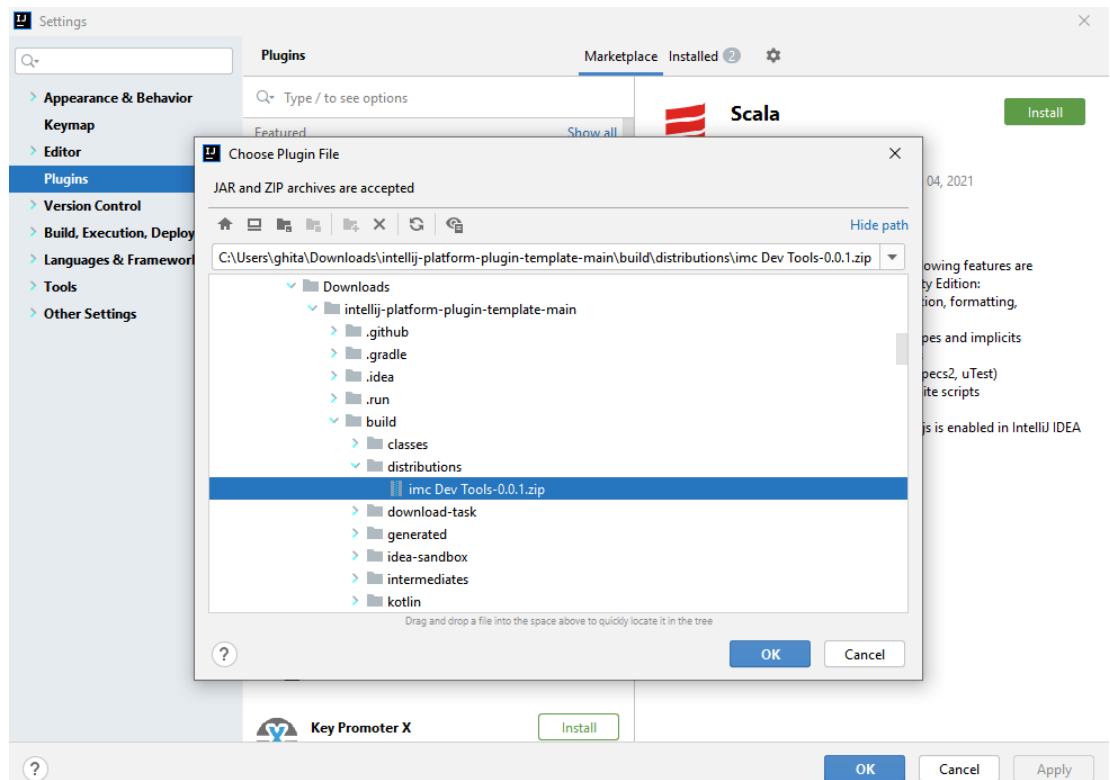


Fig. 5.2 Localizarea fișierului instalabil în folderul distributions al proiectului de dezvoltare

Odată instalat, pluginul va necesita un restart de IDE.

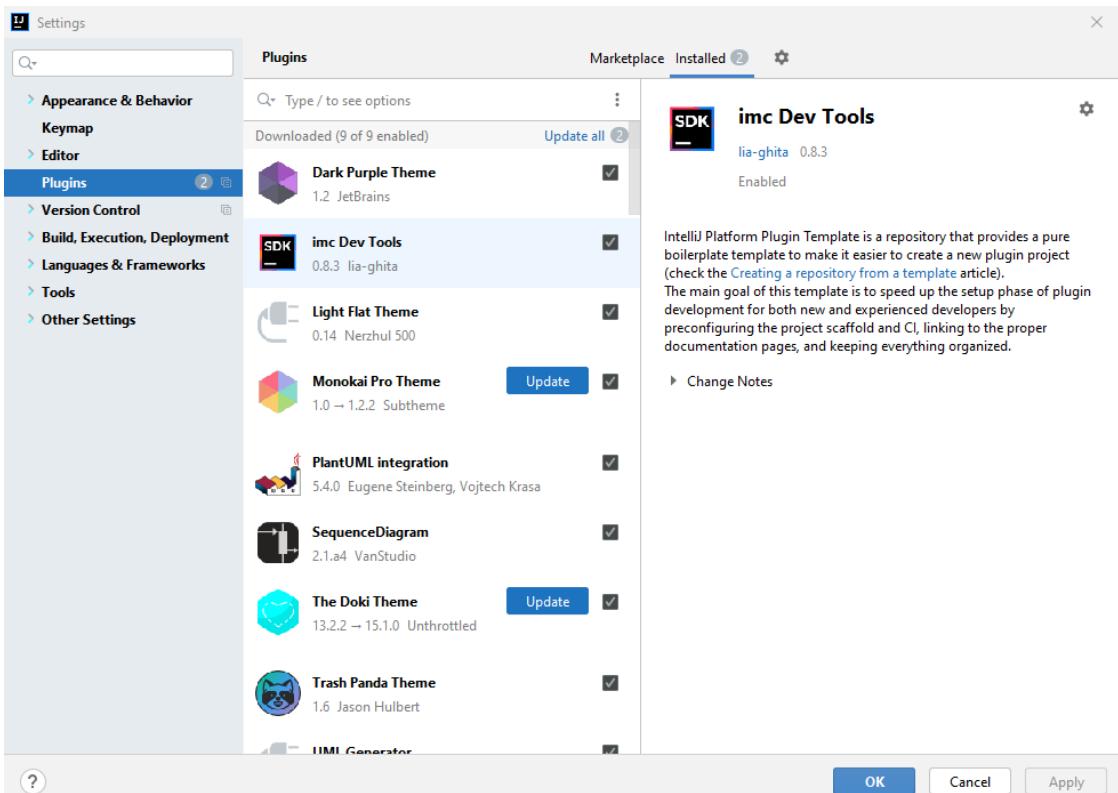


Fig. 5.3 Instalarea pluginului necesita un restart al IDE-ului

După restart, vom regăsi în meniul principal, noul buton care va lansa aplicația.

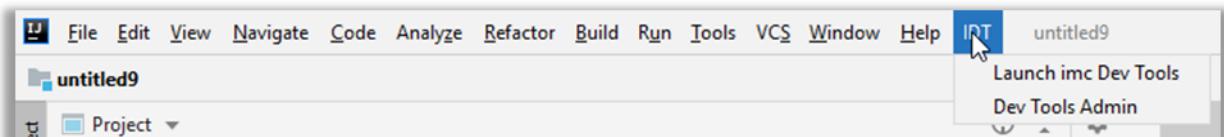


Fig. 5.4 Accesarea meniului introdus de plugin

Clasa de start trebuie să extindă obligatoriu clasa AnAction oferită de IntelliJ.

```
public class LaunchDevTools extends AnAction {
    @Override
    public void actionPerformed(@NotNull AnActionEvent anActionEvent) {
        new Thread(new PluginConfigurationStrings()).start();
        try {
            WelcomeScreenForm mm = new WelcomeScreenForm();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Am folosit clasa de start ca și clasă intermediaр, lansând de aici prima pagină afișată utilizatorului. În acest caz, pentru meniul destinat programatorului, aceasta va fi definită de clasa WelcomeScreen.

De asemenea, după cum vom explica în capitolele ce urmează, vom aduce, cu ajutorul unui nou thread, o serie de informații necesare bunei funcționări a aplicației din baza de date conectată cu aceasta.

WelcomeScreen va conține 2 fielduri, primul pentru a colecta adresa de e-mail a utilizatorului, iar cealaltă pentru a colecta id-ul ticketului de implementare la care programatorul lucrează în mod curent.

Aceste informații sunt necesare pentru utilizarea anumitor funcționalități ale aplicației, astfel că după colectarea acestora, le vom stoca static după cum urmează:

```
enterButton.addActionListener(e -> {
    CurrentUser.issueKey = taskKey.getText().toUpperCase();
    CurrentUser.email = txtName.getText();
    PluginConfigurationStrings.propadminBundle+=txtName.getText();
    jf.dispose();
    try {
        MainMenuForm mm = new MainMenuForm();
    } catch (ClassNotFoundException classNotFoundException) {
        classNotFoundException.printStackTrace();
    }
});
```

CurrentUser este o clasa cu proprietăți statice, folosită pentru a colecta date necesare pe parcursul folosirii aplicației.

```
public class CurrentUser {
    public static String issueKey = "";
    public static String email = "";
    public static String jiraPassword="";
    public static String localizationPassword="";
    public static int role=0;
}
```

După salvarea datelor, utilizatorul va fi redirecționat la meniul de opțiuni, de unde va putea alege tipul de wizard pe care dorește să îl folosească, în funcție de tipul de task pe care îl implementează.

Pentru implementarea formularelor și a elementelor de UI am folosit Swing UI Designer, facilitate oferită de mediul de dezvoltare IntelliJ, astfel că vom avea, pentru fiecare clasă care conține elemente de interfață grafică, un fișier .java care va conține codul java și un fișier .form care va fi interpretat de IDE, oferindu-ne posibilitatea de a defini și ajusta interfața grafică.

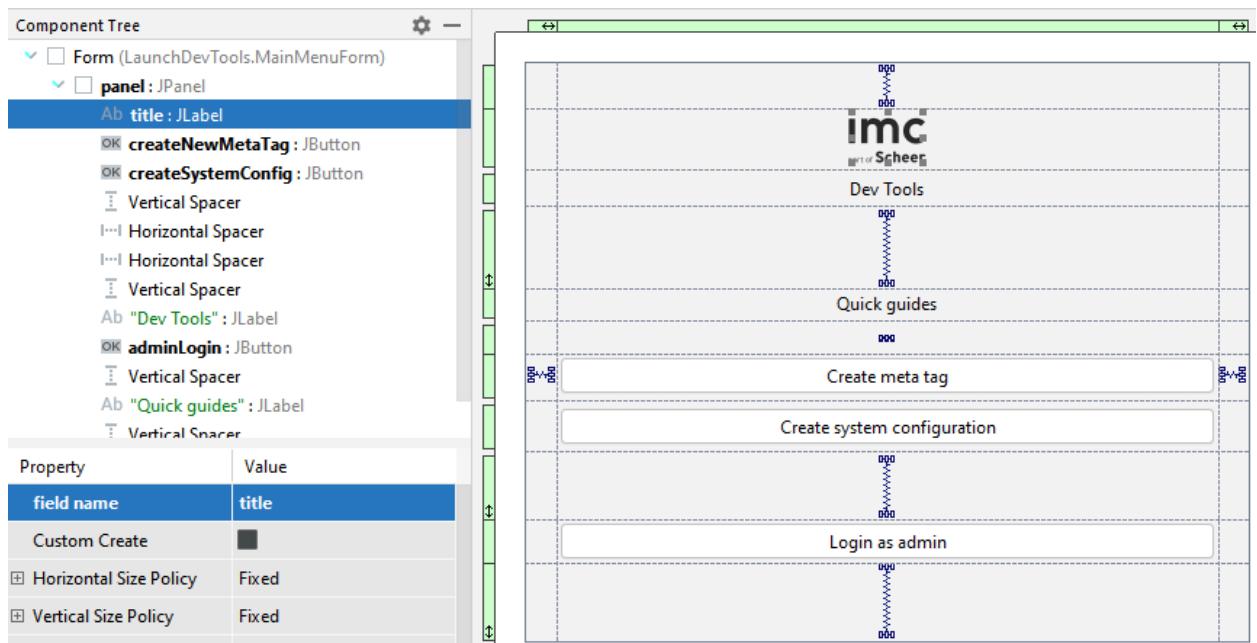


Fig 5.5 Designerul de Swing UI din cadrul IntelliJ Idea

Capitolul 6. Generare de SQL

Utilitatea acestei funcționalități reiese din faptul că, după cum am precizat și anterior, unele aspecte ce țin de extinderea scriptului de inițializare a bazei de date a aplicației părinte sunt tratate de altă persoană decât programatorul care e responsabil de compunerea acestui script.

Având în vedere că acest script nu se execută prin intermediul aplicației, ci doar se generează, nu vom folosi un PreparedStatement, ci vom compune stringul corespunzător scriptului cu ajutorul unui StringBuilder.

În clasa MetatagCreation form am definit un formular care permite colectarea de informații de la utilizator despre configurarea ce se dorește a fi creată. Aceste informații vor fi utilizate pentru a popula valorile corespunzătoare într-un statement SQL predefinit.

```
public String composeSQL(String id, String type, String isSystem, String useForCourses, String useForLP, String useForMedia, String multiLang){  
    String sql="";  
    StringBuilder builder = new StringBuilder();  
    builder.append("insert into metatag (metatag_id, language_id, creator, creator_id, creationdate, name, active,  
description, description2, formelementtype_id, required_tag, systemitem, useforcourse, useformedia,  
useforcommunity, useforservice, useforprogram, useforresource, lastupdated, lastupdater_id, useforexercisegroup,  
useforexercisesheet, useforexercise, multilang) values ("");  
    builder.append(id);  
    builder.append("dbsLANG, 'Learning Suite System', 0, "");  
    builder.append(dtf.format(now));  
    builder.append(" 00:00:00.000'");  
    builder.append(metaTagNameKey);  
    builder.append(", 1,");  
    builder.append(metaTagDescriptionKey);  
    builder.append(", ", ");  
    builder.append(type);  
    builder.append(", 0, ");  
    builder.append(isSystem);  
    builder.append(", ");  
    builder.append(useForCourses);  
    builder.append(", ");  
    builder.append(UseForMedia);  
    builder.append(", 0, 0, ");  
    builder.append(useForLP);  
    builder.append(", 0, "");  
    builder.append(dtf.format(now));  
    builder.append(" 00:00:00.000', 0, 0, 0, 0, 0,");  
    builder.append(multiLang);  
    builder.append(" );");  
    sql = builder.toString();  
    return sql;  
}
```

După cum se poate observa, metoda de generare a scriptului acceptă următorii parametri:

- Id – care corespunde id-ului meta tagului oferit de programator în cadrul formularului de creare a metatagului
- Type – care corespunde tipului de meta tag dorit. Aici vom prelua id-ul asociat fiecărei valori disponibile în lista de selecție
- Trei atrbute corespunzătoare opțiunilor ce determină dacă meta tagul dorit este relevant pentru cursuri, obiecte media, programe de studiu și respectiv dacă meta tagul nostru va fi identificat ca și meta tag de sistem.
- Un atrbut corespunzător valorii alese pentru multi language.

Valoarea corespunzătoare coloanelor „creationdate” și „lastupdated” vor fi generate programatic, corespunzând zilei curente, astfel:

```
DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy-MM-dd");
LocalDateTime now = LocalDateTime.now();
```

Deoarece există aspecte care nu se modifică frecvent în procesul de creare a unui meta tag nou, valorile acestora vor fi hardcodate, pentru a reduce din complexitatea formularului de creare a unui nou meta tag.

Capitolul 7. Interacțiunea cu codul aplicației părinte

7.1. Generarea de linii de cod în clase java

O altă funcționalitate oferită de aplicație este generarea de clase și de apendare a claselor existente când se dorește fie introducerea unei noi secțiuni de configurații în sistem, fie extinderea unei secțiuni existente.

Pentru a putea realiza astfel de operațiuni, ne folosim de partea de PSI din API-ul de IntelliJ. PSI (Program Structure Interface) permite acțiuni precum obținerea proiectului curent deschis în IDE, opținerea directorului principal, identificarea unui fișier după nume, accesarea fișierului virtual (VirtualFile) al unui fișier și modificarea conținutului acestuia.

Operațiuni mai avansate constau în lucrul cu PSIElement care permite localizarea unui anumit element din proiect sau fișier și manipularea acestuia.

Pentru toate utilitățile necesare, legate de manipularea de fișiere prin intermediul PSI, am creat o clasa Helper numită PSIHelper cu metode statice care pot fi apelate în cadrul proceselor care le necesită.

7.2. Generarea de clase noi

Pentru generarea de clase noi, am creat un pachet numit Templates, iar pentru fiecare tip de clasă ce se necesită să fie generată, am creat o clasă template.

Prin metoda `fillTemplate`, clasa template generează textul aferent codului, pe baza parametrilor primiți la apelare.

Un exemplu de astfel de clasă template se regăsește în Anexe.

Apelarea acestei metode, pentru cele trei tipuri de clase, va fi realizată din `SystemConfigCreationForm`, care va colecta datele necesare de la utilizator și va oferi aceste date ca parametri în apelul metodelor de populare a claselor necesare.

```
PsiDirectory dir= PSIHelper.createDirectory(directory, textField1.getText().toLowerCase(Locale.ROOT));
String baseClass=textField1.getText();
String baseClassContent = BaseClassTemplate.fillTemplate(baseClass,options);
String daoClass = baseClass+"ConfigDAOImpl";
String daoClassContent = ConfigDAOClassTemplate.fillTemplate(baseClass, options);
String factoryClass=daoClass+"Factory";
String factoryClassContent = SystemConfigFactoryTemplate.fillTemplate(baseClass,factoryClass);
String configClass=baseClass+"Config";
```

```

        WriteCommandAction.runWriteCommandAction(p, new Runnable() {
    @Override
    public void run() {
        PSIHelper.createFileInDirectory(directory, configClass+".java", "test", "JAVA");
    }
});

PsiFile base= PSIHelper.createFileInDirectory(dir,baseClass+".java",baseClassContent, "JAVA");
PsiFile daoFile = PSIHelper.createFileInDirectory(dir,daoClass+".java",daoClassContent, "JAVA");
PsiFile factory=PSIHelper.createFileInDirectory(dir,factoryClass+".java",factoryClassContent, "JAVA");

```

Deoarece dorim să includem clasele nou generate într-un director nou, vom începe prin crearea directorului. Pentru a putea realiza acest aspect, în clasa PSIHelper am creat o metodă numită `createDirectory`.

```

public static PsiDirectory createDirectory(PsiDirectory parent, String name)
    throws IncorrectOperationException {
    Project p = ProjectManager.getInstance().getDefaultProject();
    final PsiDirectory[] result = {null};

    for (PsiDirectory dir : parent.getSubdirectories()) {
        if (dir.getName().equalsIgnoreCase(name)) {
            result[0] = dir;
            break;
        }
    }
    if (null == result[0]) {
        WriteCommandAction.runWriteCommandAction(p, new Runnable() {
            @Override
            public void run() {
                result[0] = parent.createSubdirectory(name);
            }
        });
    }
    return result[0];
}

```

Această metodă acceptă ca parametru un director părinte și un nume. Pentru a determina directorul părinte vom porni de la o clasă pe care o folosim ca și reper, vom obține directorul părinte al acestei clase, iar apoi vom folosi directorul obținut ca și parametru cu rol de director părinte pentru clasele nou generate.

O altă remarcă interesantă este că orice acțiune de creare de directoare sau fișiere trebuie realizată prin intermediul unui `WriteCommandAction`.

```

final PsiFile[] files = FilenameIndex.getFilesByName(p, "SystemConfiguration.java",
    GlobalSearchScope.allScope(p));
if (files != null && files.length > 0) {
    file = files[0];
}
final PsiDirectory directory = file.getContainingDirectory();

```

Am folosit această metodă deoarece nu am găsit o posibilitate de a obține obiectul director direct, după aceeași metodă, în baza numelui acestuia, după cum vedem că este posibil pentru fișiere.

Probleme rezolvate pe parcursul implementării

În procesul de editare a unui fișier existent, am întâmpinat probleme în momentul în care a fost nevoie de a apenda același document de mai multe ori în cadrul aceluiași proces. Problema se prezenta prin faptul că doar ultima modificare putea fi regăsită după terminarea procesului respectiv, iar modificările intermediare erau pierdute.

Am identificat ca și soluție acțiunea de “commitDocument”, care permite salvarea modificărilor făcute într-un VirtualFile pentru a fi folosite de către următorii pași ai procesului.

```
public static void appendFile(PsiFile file, String data, String insertBeforeLastOccurrence){
    @NotNull Project[] p = ProjectManager.getInstance().getOpenProjects();
    Project project = p[0];
    VirtualFile vFile = file.getVirtualFile();
    OpenFileDescriptor descriptor = new OpenFileDescriptor(project, vFile);
    descriptor.navigateInEditor(project, true);
    StringBuilder src = new StringBuilder(file.getText());
    int i = src.lastIndexOf(insertBeforeLastOccurrence);
    src.insert(i, data);
    WriteCommandAction.runWriteCommandAction(project, new Runnable() {
        @Override
        public void run() {
            try {
                vFile.setBinaryContent(src.toString().getBytes("utf-8"));
                System.out.println(file.getNode().getChildren(null));
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    });
    FileDocumentManager fileDocumentManager = FileDocumentManager.getInstance();
    Document document = fileDocumentManager.getDocument(vFile);
    PsiDocumentManager manager = PsiDocumentManager.getInstance(project);
    manager.commitDocument(document);
}
```

O altă nevoie întâmpinată pe parcursul implementării, în special în procesul de generare de clase noi pe baza unor template-uri prestabilite, a fost faptul că textul generat nu era și formatat.

Astfel că am identificat ca și soluție clasa CodeStyleManager oferită de IntelliJ API care permite reformatarea textului unui fișier.

```
public static PsiFile createFileInDirectory(final PsiDirectory directory, String name, String content, String language) throws IncorrectOperationException {
    @NotNull Project[] p = ProjectManager.getInstance().getOpenProjects();
    Project project = p[0];
    final PsiFile currentFile = directory.findFile(name);
    if (currentFile != null) {
```

```

        return currentFile;
    }
final PsiFileFactory factory = PsiFileFactory.getInstance(directory.getProject());
final PsiFile file = factory.createFileFromText(name, Language.findLanguageByID(language), content);

//format the code
CodeStyleManager styleManager = CodeStyleManager.getInstance(project);
WriteCommandAction.runWriteCommandAction(project, new Runnable() {
    @Override
    public void run() {
        styleManager.reformat(file);
    }
});
final PsiFile[] psifile = {null};
WriteCommandAction.runWriteCommandAction(project, new Runnable() {
    @Override
    public void run() {
        psifile[0] = (PsiFile) directory.add(file);
    }
});
return psifile[0];
}

```

Capitolul 8. Folosirea serviciilor web de REST API pentru integrarea cu diverse aplicații

Pentru a putea realiza integrarea cu mai multe servicii web de tip Rest API, am implementat o structură de clase pentru realizarea conexiunii HTTP cu posibilitatea extinderii cu mai multe tipuri de autorizări.

Jira și Propadmin, cele două aplicații cu care am realizat deja integrarea, necesită două tipuri de autorizări diferite (authorization token și user/pass), astfel că aplicația plugin suportă deja aceste tipuri de autorizări.

```
public Connection(String urlString, Method method, String payload, Authorization auth) throws IOException {
    URL url = new URL(urlString);
    HttpURLConnection con = (HttpURLConnection)url.openConnection();
    con.setRequestMethod(method.name());
    if (auth != null) {
        auth.addAuthorization(con);
    }
    con.setRequestProperty("Content-Type", "application/json");
    con.setRequestProperty("Accept", "application/json");
    con.setRequestProperty("Connection", "keep-alive");
    con.setDoOutput(true);
    try(OutputStream os = con.getOutputStream()) {
        byte[] input = payload.getBytes(StandardCharsets.UTF_8);
        os.write(input, 0, input.length);
    }
    try(BufferedReader br = new BufferedReader(
            new InputStreamReader(con.getInputStream(), StandardCharsets.UTF_8))) {
        StringBuilder response = new StringBuilder();
        String responseLine = null;
        while ((responseLine = br.readLine()) != null) {
            response.append(responseLine.trim());
        }
        String json = response.toString();
        JsonObject convertedObject = new Gson().fromJson(json, JsonObject.class);
        this.code = con.getResponseCode();
        this.responseObject = convertedObject;
    }
}
```

Constructorul acestei clase va primi ca și parametri :

- API endpoint sub formă de String
- Metoda (GET, PUT, POST) sub formă de valoare a unui enum predefinit pentru a asigura consistență și a reduce posibilitatea de a oferi valori invalide

```
public enum Method {
    GET, PUT, POST
}
```

- Payload-ul sub formă de String
- Informațiile necesare autorizării. Aici folosim o interfață pentru a ne asigura că primim ca și parametru un obiect de autorizare valid. Clasele care implementează această interfață sunt obligate să implementeze metoda addAuthorization. Am definit această interfață direct în cadrul clasei Connection, deoarece aceasta există și este utilizată doar în acest context.

```
interface Authorization {
    void addAuthorization(HttpURLConnection connection);
}
```

Metoda addAuthorization acceptă ca și parametru un HttpURLConnection și, în momentul apelării, va adăuga prin metoda setRequestProperty headerul Authorization conexiunii primite ca și parametru.

Avem două clase care implementează interfața Authorization: BasicAuthorization și TokenAuthorization. Implementând interfața, acestea sunt obligate să implementeze metoda addAuthorization, astfel că fiecare clasă va seta conexiunii primite ca parametru proprietatea Authorization cu valoarea generată în mod propriu.

```
public class TokenAuthorization implements Connection.Authorization {
    private final String header;
    public TokenAuthorization(String token) {
        header = "Bearer " + token;
    }
    @Override
    public void addAuthorization(HttpURLConnection con) {
        con.setRequestProperty("Authorization", header);
    }
}
```

TokenAuthorization va seta ca și Authorization Bearer token-ul.

```
public class BasicAuthorization implements Connection.Authorization {
    private final String header;

    public BasicAuthorization(String user, String pass) {
        header = "Basic " +
            Base64.getEncoder().encodeToString((user+":"+pass).getBytes(StandardCharsets.UTF_8));
    }
    @Override
    public void addAuthorization(HttpURLConnection con) {
        con.setRequestProperty("Authorization", header);
    }
}
```

BasicAuthorization va compune stringul de autorizare din username-ul și parola primite ca și parametri.

Se poate pune aici întrebarea dacă ar fi mai potrivită crearea unei clase abstracte, având în vedere că metoda addAuthorization este de fapt identică în cele două clase, iar diferențele în implementare se află în proprietățile claselor care implementează interfața. Totuși, ținând cont de faptul că în Java putem implementa oricără interfețe avem nevoie, dar putem extinde o singură clasă, am considerat mai potrivită varianta unei interfețe, pentru a nu introduce posibile limitări pentru nevoile de extindere ulterioară a claselor noastre. De asemenea, avem nevoie de acest „contract” pentru a ne asigura că toate clasele care implementează interfața au fost obligate să implementeze metoda pe care ne bazăm.

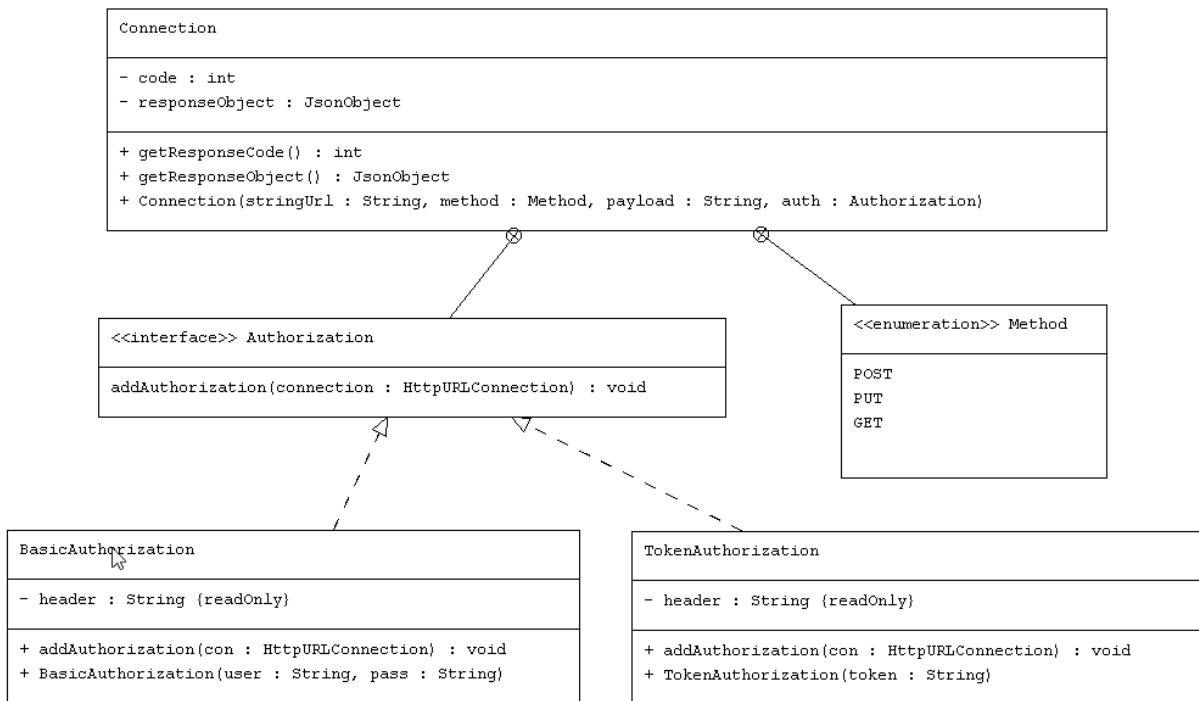


Fig. 8.1 Diagrama UML a claselor folosite pentru conexiunea HTTP pentru REST API

8.1. Integrarea cu Jira pentru crearea de sub-taskuri

Ideea de integrare cu Jira a reieșit și din faptul că, în prezent, prin intermediul altor procese implementate în cadrul companiei, există posibilitatea de a lăsa commentarii automate în baza commiturilor.

Conform documentației Jira, pentru a putea crea un sub-task prin intermediul REST API, trebuie să trimitem serverului ca și payload o structura de forma următoare:

```
{
  "fields": {
    "project": {
      "key": "TEST"
    },
    "parent": {
      "key": "TEST-101"
    },
    "summary": "Sub-task of TEST-101",
    "description": "Don't forget to do this too.",
    "issuetype": {
      "id": "5"
    }
  }
}
```

Am creat astfel o structura de clase JiraTask, care înglobează clasa JiraTaskFields, clasele Project, Parent si IssueType.

Structura de clase este urmatoarea:

```
public class JiraTask {
  JiraTaskFields fields;
  public JiraTask(String parentKey, String description){
    this.fields = new JiraTaskFields(parentKey,description);
  };
  public JiraTaskFields getTask(){
    return fields;
  }
  public static class JiraTaskFields {
    Project project = new Project();
    Parent parent;
    String summary = "Insert SQL for new meta tag";
    String description;
    IssueType issuetype = new IssueType();
    public JiraTaskFields (String parentKey, String description){
      this.parent=new Parent(parentKey);
      this.description = description;
    }
  }
}
```

```

public class Project{
    String key = PluginConfigurationStrings.jiraProject,
}

public class Parent {
    String key;
    public Parent(String key){
        this.key=key;
    }
}

public class IssueType{
    String id=PluginConfigurationStrings.jiraSubtaskId,
}
}
}

```

Pentru a putea testa într-un mediu sandbox și a nu impacta server-ul folosit de firmă în producție, am creat un server de test Jira, unde am creat un proiect nou cu identificatorul PLUGINDEV.

Conform documentației Atlassian, pentru a obține valoarea corectă pentru issuetype id, va trebui accesat via REST API endpoint-ul /issue/createmeta pentru a obține valoarea corectă corespunzătoare acestui tip de ticket pentru serverul de Jira la care legăm conexiunea.

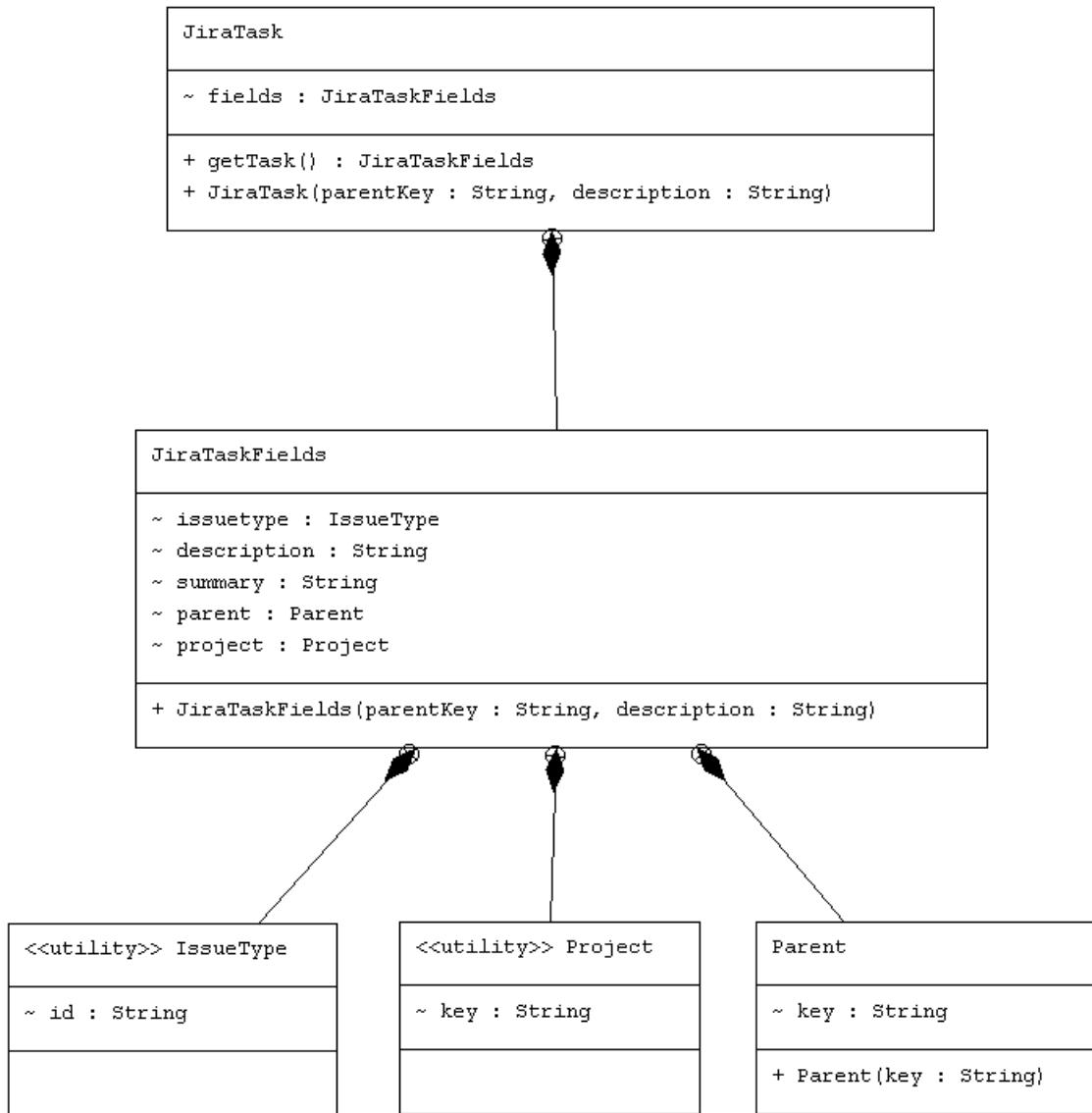


Fig 8.2 Diagrama UML a claselor folosite pentru modelarea unui task Jira

Dintr-o clasă helper numită **JiraRequestHelper**, unde am colectat mai multe metode utile în lucruri cu taskuri, am definit și metodă responsabilă pentru generarea payloadului pentru request-ul de creare a unui nou task/sub-task.

```

public static String getJiraSubTaskCreationRequestBody(String parentId, String description){
    JiraTask jt = new JiraTask(parentId, description);
    return new Gson().toJson(jt);
}

```

Am folosit aici librăria **Gson** pentru a putea genera răspunsul în format JSON.

La apelarea acestei metode, se va genera un obiect ce va fi trimis ca payload în momentul solicitării de creare a unui nou sub-task pentru taskul la care programatorul lucrează și care a fost definit prin cheia unică în momentul lansării aplicației.

Aveam mai jos exemplul de apel al conexiunii, specific pentru procesul de creare a unui ticket în sistemul Jira, unde autorizarea folosită este cea Basic, pe baza de username și parolă:

```

AddSQLToJira.addActionListener(e -> {
    if (CurrentUser.jiraPassword.equals("")) {
        JPasswordField pf = new JPasswordField();
        int clicked = JOptionPane.showConfirmDialog(null, pf, "Enter your Jira password",
            JOptionPane.OK_CANCEL_OPTION, JOptionPane.PLAIN_MESSAGE);
        if (clicked == JOptionPane.OK_OPTION) {
            CurrentUser.jiraPassword = new String(pf.getPassword());
        }
    } else {
        try {
            Connection createJiraSubTask = new Connection(PluginConfigurationStrings.jiraIssueRoot,
                Connection.Method.POST, JiraRequestHelper.getJiraSubTaskCreationRequestBody(parentId, sql), new
                BasicAuthorization(CurrentUser.email, CurrentUser.jiraPassword));
            String response = createJiraSubTask.getResponseObject().get("key").getAsString();
            Connection assignTask = new Connection(PluginConfigurationStrings.jiraIssueRoot + response,
                Connection.Method.PUT, JiraRequestHelper.getJiraTaskAssignmentToUserBody(assignee), new
                BasicAuthorization(CurrentUser.email, CurrentUser.jiraPassword));
            if (createJiraSubTask.getResponseCode() < 299 && assignTask.getResponseCode() < 299) {
                JOptionPane.showMessageDialog(null, "Sub-task created and assigned for commit on the init
                    script. The key for the created Jira ticket is " + response);
            } else if ((createJiraSubTask.getResponseCode() < 299) || (createJiraSubTask.getResponseCode()
                >= 400)) {
                JOptionPane.showMessageDialog(null, "The sub-task could not be created" +
                    createJiraSubTask.getResponseObject());
            } else {
                JOptionPane.showMessageDialog(null, "The sub-task was created but it could not be assigned.
                    The key for the created Jira ticket is " + response);
            }
        } catch (IOException ioException) {
            JOptionPane.showMessageDialog(null, "The sub-task could not be created. Please check your
                credentials and try again" );
            ioException.printStackTrace();
        }
    }});
}

```

Deoarece Jira necesită autentificare, vom solicita utilizatorului să introducă parola pentru contul său de Jira la prima încercare de folosire a acestei funcționalități. Pentru utilizări ulterioare, având în vedere că folosim o clasă cu proprietăți statice pentru a reține această informație, nu va mai fi nevoie de o nouă reintroducere a parolei, decât în cazul în care mediul de dezvoltare IntelliJ Idea va fi oprit.

În cazul de față, parola utilizatorului va fi un token ce trebuie emis prin intermediul aplicației Jira, conexiunea Basic cu parola fiind deprecată pentru acest tip de server. [7]

Prin metoda getJiraTaskAssignmentToUserBody, vom assigna sub-taskul.

```

<<utility>> JiraRequestHelper

+ getJiraTaskAssignmentToUserBody(assignee : String) : String
+ getJiraSubTaskCreationRequestBody(parentId : String, description : String) : String

```

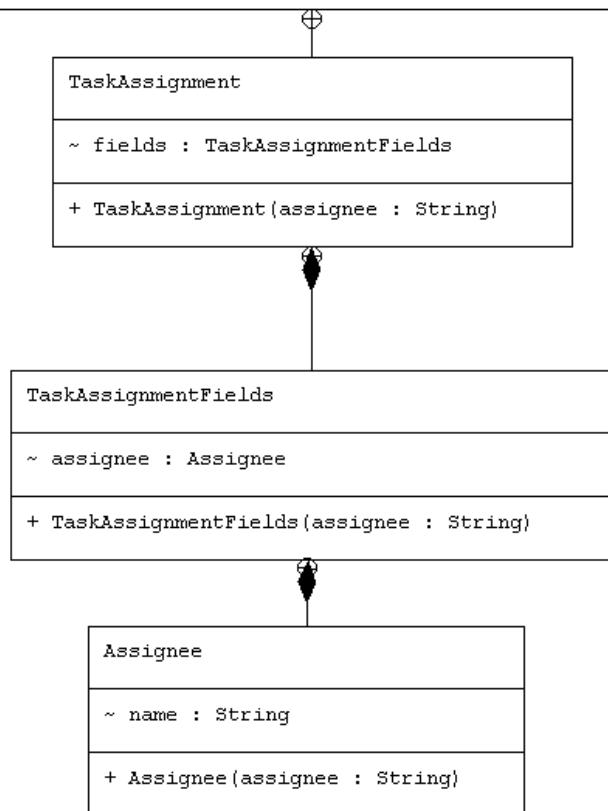


Fig 8.3 Diagrama UML pentru clasele folosite la compunerea requestului de assignare a unui task Jira

8.2. Integrarea cu aplicația de management al localizării

În mod similar integrării cu Jira, am realizat integrarea cu aplicația Propadmin, dezvoltată intern.

Prin această integrare, vom colecta numele și descrierea introduse de către programator prin intermediul formularului de creeare a unui metatag, vom prelua cheile generate pentru aceste texte și vom facilita crearea imediată a textelor în aplicația specializată pe managementul localizării, simplificand procesul care ar presupune navigarea către o altă aplicație și completarea tuturor informațiilor din nou.

Pentru această integrare am implementat clasa Bundle, care va îngloba toate atributurile necesare creării unui nou bundle prin intermediu REST API.

```
public class Bundle {
    public List<String> bundleLocations = new ArrayList<>();
    public String identifier;
    public String context;
    public List<Translations> translations;
    public boolean fullTranslated;
    public String createdBy;
    public boolean staysEmpty = false;
    public boolean isLocked = false;
    public final String component ="default";
    public final String part ="default";
    public final String services=":backend:";
    public void addTranslation(Translations t) {
        if (translations == null) {
            translations = new ArrayList<>();
        }
        translations.add(t);
    }
    public Bundle (String identifier, String context, String createdBy){
        this.identifier=identifier;
        this.context=context;
    }
    public static class Translations {
        public String language, code, value;

        public Translations(String language, String code, String value) {
            this.language = language;
            this.code = code;
            this.value = value;
        }
    }
}
```

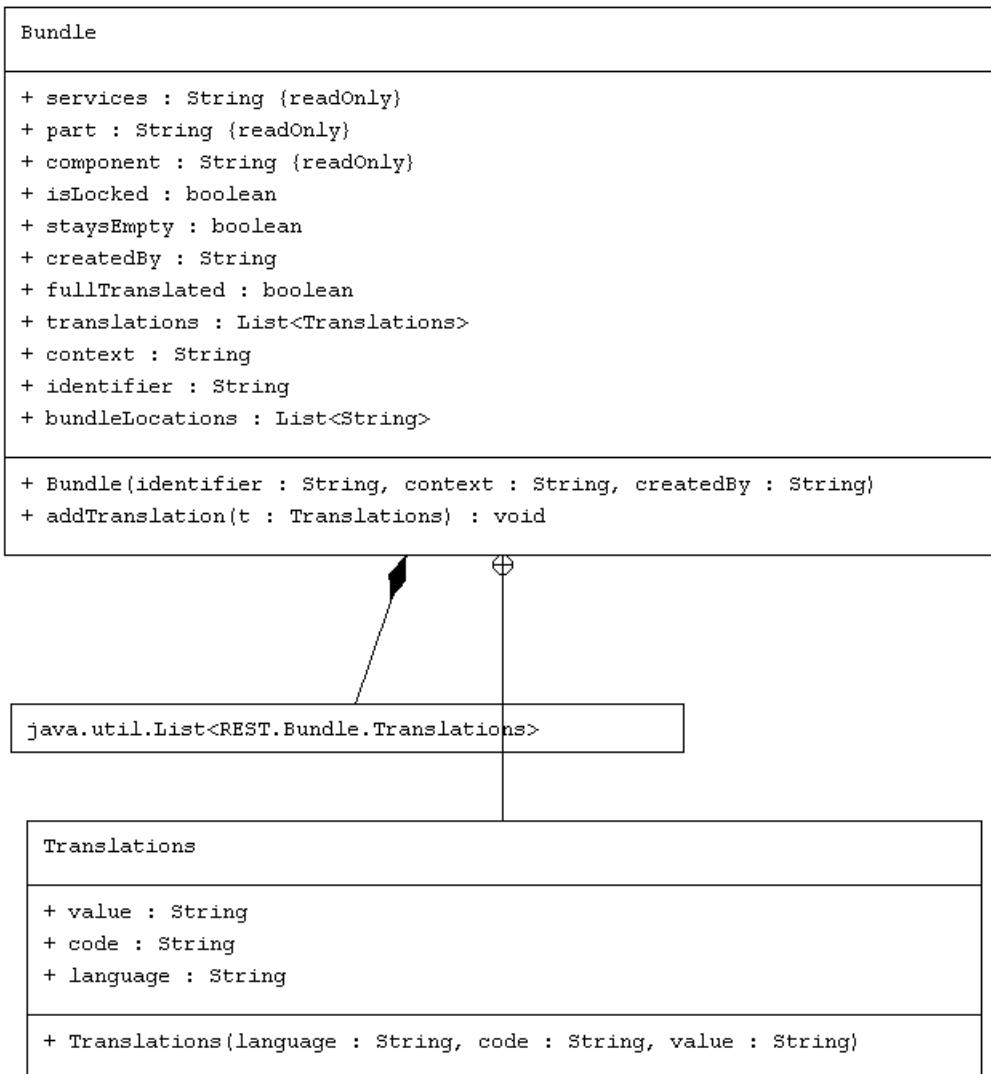


Fig. 8.4 Diagrama UML a claselor folosite la compunerea unui request de creare a unui bundle

Deoarece aplicația Propadmin oferă mai multe facilități decât cele necesare proiectului pe care îl dezvoltăm, fiind folosită că aplicație de management al textelor și pentru alte aplicații, vom folosi pentru anumite attribute niște valori default, care nu vor putea fi modificate în acest proces. Acest lucru ne asigură că vom crea doar bundle-uri compatibile cu aplicația pentru care folosim pluginul și în același timp scurtând procesul care altfel ar necesita niște clickuri în plus pentru a specifica aceste valori.

Un exemplu de astfel de atribut cu valoare default este cel de „services”, unde întotdeauna vom folosi valoarea `:backend:`, astfel că această valoare este hardcodată în clasa noastră.

Procesul de creare a unui bundle presupune conexiunea prin `username` și `password` pentru a obține tokenul de autorizare, urmând ca toate celelalte request-uri către aplicație să folosească o autorizare prin token.

```

CreateBundles.addActionListener(e -> {
    if (PluginConfigurationStrings.propadminAuth.equals("")) {
        JTextField path = new JTextField();
        int clicked = JOptionPane.showConfirmDialog(null, path, "Enter the localization server URL",
JOptionPane.OK_CANCEL_OPTION, JOptionPane.PLAIN_MESSAGE);
        if (clicked == JOptionPane.OK_OPTION) {
            PluginConfigurationStrings.propadminAuth = path.getText();
        }
    }
    else if (PluginConfigurationStrings.propadminBundle.equals("")) {
        JTextField path = new JTextField();
        int clicked = JOptionPane.showConfirmDialog(null, path, "Enter the bundle endpoint",
JOptionPane.OK_CANCEL_OPTION, JOptionPane.PLAIN_MESSAGE);
        if (clicked == JOptionPane.OK_OPTION) {
            PluginConfigurationStrings.propadminBundle = path.getText() + "?user=" + CurrentUser.email;
        }
    }
    else if (CurrentUser.localizationPassword.equals("")) {
        JPasswordField pf = new JPasswordField();
        int clicked = JOptionPane.showConfirmDialog(null, pf, "Enter your Propadmin password",
JOptionPane.OK_CANCEL_OPTION, JOptionPane.PLAIN_MESSAGE);
        if (clicked == JOptionPane.OK_OPTION) {
            CurrentUser.localizationPassword = new String(pf.getPassword());
        }
    } else {
        String jsonAuth = "{\"username\":\"" + CurrentUser.email + "\", \"password\":\"" +
CurrentUser.localizationPassword + "\"}";
        // authenticate to Propadmin
        try {
            Connection propadminAuth = new Connection(PluginConfigurationStrings.propadminAuth,
Connection.Method.POST, jsonAuth, null);
            // fetch authorization token
            String token = propadminAuth.getResponseBody().get("token").getAsString();
            // compose the body for the call to create the bundles
            String context = "Name of the meta tag";
            String metaTagName =
PropadminRequestHelper.getBundleCreationRequestBody(MetaTagCreationForm.metaTagNameKey,
MetaTagCreationForm.metaTagName, context);
            context = "Description of the meta tag";
            String metaTagDescription =
PropadminRequestHelper.getBundleCreationRequestBody(MetaTagCreationForm.metaTagDescriptionKey,
MetaTagCreationForm.metaTagDescription, context);
            Connection createNameBundle = new Connection(PluginConfigurationStrings.propadminBundle,
Connection.Method.POST, metaTagName, new TokenAuthorization(token));
            Connection createDescriptionBundle = new
Connection(PluginConfigurationStrings.propadminBundle, Connection.Method.POST, metaTagDescription,
new TokenAuthorization(token));
            if (createNameBundle.getResponseCode() < 299 && createDescriptionBundle.getResponseCode() <
299) {
                JOptionPane.showMessageDialog(null, "Bundles successfully created");
            } else if (createNameBundle.getResponseCode() < 299 || createDescriptionBundle.getResponseCode() <
299) {
                JOptionPane.showMessageDialog(null, "One or both of the bundles could not be created");
            }
        } catch (IOException ioException) {
            ioException.printStackTrace();
        }
    }
});

```

Și în acest caz, similar cu cel al conexiunii cu Jira, vom solicita utilizatorului să introducă parola corespunzătoare contului sau pentru aplicația de management al localizării.

De asemenea, având în vedere că avem nevoie de URL-ul corespondent serverului, dacă acesta nu este prezent în baza de date, îl vom solicita utilizatorului. Deși în mod normal acest lucru nu ar trebui să se întâpte, am adăugat această posibilitate pentru scopuri de testare, având în vedere că baza de date nu va pre-popula această valoare.

Capitolul 9. Administrare

Pentru a facilita modificarea unor parametri, am introdus posibilitatea ca un administrator să se poată loga prin intermediul pluginului și să configureze anumite aspecte ce țin de clasele de bază pe care proiectul părinte trebuie să le conțină pentru ca pluginul să poată funcționa corect.

Acești parametri sunt salvați într-o baza de date MySQL.

Pentru aplicații de tip IntelliJ Plugin, nu toate dependințele se pot declara ca și dependințe de proiect. Pentru a putea realiza conexiunea cu baza de date, a fost nevoie de a adăuga mysqlconnector ca și dependință în build.gradle.

Accesul la panoul de administrare se realizează printr-un sub-menuu al meniului IDT, pe care l-am adăugat în plugin.xml sub opțiunea de lansare a aplicației de către un non-administrator.

Structura bazei de date

Baza de date conține următoarele tabele:

users - tabela în care sunt stocate datele utilizatorilor

configurations – tabela în care sunt stocate configurările de care aplicația are nevoie pentru a procesa diverse acțiuni.

Deoarece configurările sunt configurații de sistem, unice astfel indiferent de utilizatorul aplicației, nu este nevoie de a defini nicio cheie externă în niciuna dintre tabele, considerând că nu este nevoie să monitorizăm cine a introdus sau modificat configurația, dat fiind faptul că deocamdată există un singur utilizator du drept de administrator, care poate accesa și modifica aceste valori.

Bineînțeles, o posibilitate de extindere există în această zonă, în contextul în care mai mulți utilizatori li se va acord acest drept, fiind util într-un astfel de caz să monitorizăm cine a adus modificări acestor valori.

The screenshot shows two tables from a MySQL database named '00490502_lgdb'.
The first table, 'users', has the following structure:

- id : int(11)
- username : varchar(50)
- password : mediumtext
- # isAdmin : tinyint(1)

The second table, 'configurations', has the following structure:

- id : int(11)
- name : text
- value : varchar(50)

Fig 9.1 Structura bazei de date

9.1. Încărcarea valorilor pentru configurări

Deoarece stocăm în baza de date valori precum endpoint-ul către serverul de Jira, codul proiectului, la fel și id-ul corespunzător unui sub-task pe serverul la care ne conectăm, pentru ca aplicația să poată funcționa corect, vom încărca aceste valori la lansarea aplicației, printr-o clasă similară cu cea folosită pentru stocarea datelor utilizatorului conectat.

Astfel, în clasa PluginConfigurationSettings vom declara atribute statice corespunzătoare valorilor pe care dorim să le încărcăm, iar la încărcarea clasei LaunchDevTools, prin intermediul lui ConfigStringsDAO, vom obține din baza de date toate datele necesare.

Obținerea informațiilor este procesată de un thread separat, care va apela metoda run a clasei PluginConfigurationSettings.

O altă variantă de implementare ar fi putut fi încărcarea acestor informații doar la momentul în care utilizatorul dorește să le folosească, însă datorită faptului că informația nu este amplă, iar o posibilă problemă de acces la baza de date apărută după ce utilizatorul a început activitatea în plugin poate conduce la frustrare pentru munca deja depusă în completarea datelor necesare, am ales varianta de a realiza încărcarea de la început. Astfel, în cazul unei probleme de acces la baza de date cu care aplicația este conectată, este util să fim avertizați cât mai curând.

În cazul în care anumite valori, precum URL-ul serverului aplicației de management al localizărilor nu sunt definite în baza de date, utilizatorului i se va solicita să introducă această informație de la tastatură pentru a putea continua procesul de conectare cu acel server. Acest lucru poate fi întâlnit în perioada de testare, când pluginul este conectat posibil la o bază de date locală unde aceste informații nu sunt disponibile.

9.2. Autentificare

Parola utilizatorului cu rol de administrator este stocată în baza de date sub formă de hash de tip SHA2. Astfel că pentru a realiza autentificarea administratorului prin intermediul aplicației, vom prelua valoarea introdusă de acesta în câmpul destinat parolei și vom transforma valoarea prin metoda sha, urmând apoi să comparăm valoarea obținută cu cea stocată în baza de date. Pentru aceasta am creat o funcție numită hashpass apartinând bazei de date, funcție pe care am apelat-o și la crearea utilizatorului cu rol de administrator.

Sintaxa de creare a funcției este următoarea:

```
DELIMITER $$  
CREATE FUNCTION `hashpass`(upass text) RETURNS text CHARSET utf8  
DETERMINISTIC  
begin  
SET @pass=concat (upass,"saltstr");  
SET @shahex = sha2(@pass, 512);  
return @shahex;
```

```
end$$  
DELIMITER ;
```

Dar fiind că metoda de hashare SHA este o funcție deterministă, pentru aceeași valoare oferită de utilizator vom obține mereu aceeași valoare hash.

Pentru un plus de siguranță am folosit și un string adițional, numit “salt”, pe care funcția noastră îl va concatena la valoarea oferită de utilizator.

Concluzii și direcții de dezvoltare

Concluzii

În urma implementării cu succes a funcționalităților de generare de cod, integrare cu aplicații externe și conectarea cu baza de date, putem concluziona că aplicații de tip IntelliJ plugin (și nu numai) pot fi folosite pentru o varietate de scopuri, deși în prezent majoritatea pluginurilor existente în Plugin store au ca și scop oferirea de teme personalizate sau suport pentru limbaje customizează.

Din teste efectuate până în prezent, folosirea aplicației contribuie într-o mare măsură la reducerea timpului necesar implementării unor funcționalități comune și taskuri repetitive, iar disponibilitatea aplicației ca și plugin pentru mediul de dezvoltare folosit de programator îl motivează pe acesta să apeleze la această funcționalitate mai mult decât dacă acesta aplicație ar fi una independentă.

De asemenea, SDK-ul oferit de JetBrains pentru implementarea de pluginuri pentru IDE-urile oferite de aceștia este suficient de cuprinzător cât să ofere toate funcționalitățile necesare pentru a putea adapta o astfel de soluție la orice proiect.

Direcții de dezvoltare

Aplicația de față este un început în direcția automatizării, dar necesită extinderi pentru a o transforma într-o soluție de durată.

Având în vedere că în prezent template-urile folosite pentru generarea de clase fac parte din codul aplicației plugin, se vede nevoia extinderii aplicației cu posibilitatea definirii acestor template-uri extern, astfel ca un administrator să poată fie modifica, fie adăuga noi template-uri necesare altor implementări.

De asemenea, dată fiind integrarea cu alte platforme, o altă oportunitate de extindere ar fi suport pentru SSO, astfel ca programatorul să nu mai fie nevoie să se autentifice pentru a putea executa operații în legătură cu aplicațiile conectate.

În ceea ce ține de configurații, dar și de preferințe de lucru, aplicația se poate extinde cu posibilitatea de a configura și salva anumite aspecte și de către programator, astfel încât aplicația să-i servească într-o manieră cât mai apropiată nevoilor acestuia.

O integrare mai profundă cu Jira ar fi de asemenea de utilitate.

Bibliografie

- [1] Diagrama arhitecturii monolit vs. Microservicii -
https://dev.to/alex_barashkov/microservices-vs-monolith-architecture-411m
[05.06.2021]
- [2] Definiția noțiunii de timp - <https://ro.wikipedia.org/wiki/Timp> [13.06.2021]
- [3] Revoluția industrială - https://en.wikipedia.org/wiki/Industrial_Revolution
[13.06.2021]
- [4] Scurtă descriere a metodologiei Agile oferită de Atlassian, dezvoltatorul Jira -
<https://www.atlassian.com/agile> [13.06.2021]
- [5] User Stories Applied for Agile Software Development – Mike Cohn, Pearson Education, 2004 – pag 4, 17
- [6] Agile Estimating and Planning – Mike Cohn, 2006, Pearson Education, pag.36-38
- [7] Agile Estimating and Planning – Mike Cohn, 2006, Pearson Education, pag.43-47
- [8] Diferențe între Java și Kotlin - <https://kotlinlang.org/docs/comparison-to-java.html>
[13.06.2021]
- [9] Null-safety in Kotlin - <https://kotlinlang.org/docs/null-safety.html> [13.06.2021]
- [10] Cele mai populare limbaje de programare -
<https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/> [13.06.2021]
- [11] Cele mai populare motoare de baze de date -
<https://www.statista.com/statistics/809750/worldwide-popularity-ranking-database-management-systems/> [13.06.2021]
- [12] Servicii web RESTful - https://en.wikipedia.org/wiki/Representational_state_transfer
[13.06.2021]
- [13] Gson - <https://en.wikipedia.org/wiki/Gson> [13.06.2021]
- [14] Busy plugin developers series. Episode 0 publicat de Jetbrains -
<https://www.youtube.com/watch?v=-6D5-xEaYig> [03.03.2021]
- [15] IntelliJ Platform SDK - <https://plugins.jetbrains.com/docs/intellij/welcome.html>
[06.06.2021]
- [16] Declararea de dependințe Gradle -
https://www.reddit.com/r/gradle/comments/apme5g/how_to_add_mysqlconnector_to_gradle_build/ [06.06.2021]
- [17] Documentație Jira - <https://developer.atlassian.com/server/jira/platform/jira-rest-api-example-create-issue-7897248/> [06.06.2021]
- [18] Documentație Jira - <https://developer.atlassian.com/cloud/jira/platform/basic-auth-for-rest-apis/> [06.06.2021]

Anexe

Anexa 1

Exemplu de template de generare a unei clase.

```
public class BaseClassTemplate {  
  
    public static String fillTemplate (String category, List<ConfigOption> options){  
        StringBuilder sb = new StringBuilder();  
        sb.append("package configuration.");  
        sb.append(category.toLowerCase(Locale.ROOT));  
        sb.append("\n");  
        sb.append("public class ");  
        sb.append(category);  
        sb.append("\n");  
        for(ConfigOption option:options){  
            sb.append("private ");  
            sb.append(option.getType());  
            sb.append(" ");  
            sb.append(option.getTitle());  
            sb.append("\n");  
        }  
        for (ConfigOption option:options){  
            sb.append("public ");  
            sb.append(option.getType());  
            sb.append(" ");  
            sb.append("get");  
            sb.append(option.getTitle());  
            sb.append(" () {\n");  
            sb.append("return ");  
            sb.append(option.getTitle());  
            sb.append(");  
            sb.append(");  
            sb.append("\n");  
            sb.append("}");  
            sb.append("public void ");  
            sb.append("set");  
            sb.append(option.getTitle());  
            sb.append(" () {\n\n");  
            sb.append("this. ");  
            sb.append(option.getTitle());  
            sb.append(" = ");  
            sb.append(option.getTitle());  
            sb.append("\n");  
        }  
        sb.append("}");  
        return sb.toString();  
    }  
}
```