UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAŞI

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Interfaţă web pentru RMT şi alte instrumente similare ca mod de funcţionare

propusă de

Oana-Iuliana Anghelache

Sesiunea: *iulie, 2017*

Coordonator ştiinţific

Titlu Ştefan Ciobâcă

UNIVERSITATEA "ALEXANDRU IOAN CUZA" DIN IAŞI
FACULTATEA DE INFORMATICĂ

Interfaţă web pentru RMT şi alte instrumente similare ca mod de funcţionare

Oana-Iuliana Anghelache

Sesiunea: *iulie, 2017*

Coordonator ştiinţific
Titlu Ştefan Ciobâcă

DECLARAȚIE PRIVIND ORIGINALITATE ŞI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licență cu titlul „*Interfață web pentru RMT și alte instrumente similare ca mod de funcționare*” este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituție de învățământ superior din țară sau din străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi dețin referința precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alți autori deține referința precisă;
- codul sursă, imaginile etc. preluate din proiecte *open-source* sau alte surse sunt utilizate cu respectarea drepturilor de autor şi dețin referințe precise;
- rezumarea ideilor altor autori precizează referința precisă la textul original.

Iaşi, *28.06.2017*

Absolvent *Oana-Iuliana Anghelache*

_____

(semnătura în original)

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul „*Interfaţă web pentru RMT şi alte instrumente similare ca mod de funcţionare*”, codul sursă al programelor şi celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” din Iaşi, să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iaşi, *28.06.2017*

Absolvent *Oana-Iuliana Anghelache*

_____

(semnătura în original)

Cuprins

Introducere

O aplicaţie web aduce multe beneficii atunci când vrem să oferim anumite funcţionalităţi într-un mod cât mai accesibil, şi unui public cât mai mare. Interacţiunea cu utilizatorul este realizată pe un navigator web, iar aplicaţia se poate folosi de către oricine, indiferent de tipul de dispozitiv: desktop, tabletă, telefon, IPAD. În plus, se poate accesa de pe orice sistem de operare şi nu are nevoie de instalare, aşa cum se întâmplă în cazul unei aplicaţii de tip desktop. Pentru întreţinere, în urma unei actualizări, utilizatorii vor avea acces imediat la noua versiune.

Un alt mare beneficiu, pentru dezvoltatori, este comunitatea imensă ce contribuie la oferirea unor numeroase pachete, framework-uri, librării ce pot fi folosite gratuit şi ajută în dezvoltarea propriei aplicaţii. În plus, există comunităţi ce oferă metode de utilizare, tutoriale, sau soluţii de rezolvare a unor probleme ce pot fi întâmpinate în diferite tehnologii.

Am ales ca temă de licenţă crearea unei aplicaţii web folosind AngularJS, împreună cu PHP, ce permite execuţia mai multor instrumente similare ca mod de funcţionare. Acestea sunt instrumentele software ce rulează de la linia de comandă, de obicei prototipuri realizate de echipe de cercetători care nu au timp să dezvolte o interfaţă web, însă doresc să poată fi folosite de cât mai mulţi utilizatori. Proiectul realizat oferă posibilitatea de a testa sau de a învăţa aceste instrumente folosind o interfaţă web uşor de utilizat.

Aplicaţia este accesibilă oricărui tip de utilizator, în special persoanele care nu sunt învăţate să folosească comenzile de Linux sau Windows pentru a executa instrumentele sau pentru a scrie propriile programe pentru acestea, şi preferă o interfaţă web.

În plus, pentru a avea acces la funcţionalitatea oferită, nu sunt necesare toate instalările şi configurările, împreună cu toate dependenţele pe care le are un instrument pentru a fi executat. Toate acestea pot fi greu de realizat şi sunt şanse mari ca utilizatorul să întâmpine tot felul de dificultăţi. Astfel, instalările au fost făcute în etapa de dezvoltare, la nivel de server. Detaliile configurărilor pentru instrumentele folosite, pot fi citite la capitolul Tipuri de instrumente.

Pe lângă avantajul simplităţii în a folosi aplicaţia implementată, aceasta a fost adaptată astfel încât datele să fie prezentate printr-un design plăcut, alegând câteva culori ce se potrivesc

şi un mod de aranjare a secţiunilor în pagină, ce se pot integra în funcţie de dimensiunile ecranului sau rezoluţia dispozitivului, de la mobil, la desktop.

Lucrarea de faţă este structurată în patru capitole. Capitolul întâi prezintă funcţionalitatea proiectului, împreună cu o descriere succintă a tipului de instrumente pe care le-am inclus pentru utilizatori. În capitolul al doilea am descris cum am instalat proiectul local, înainte de a începe implementarea, şi ce tehnologii am folosit. Capitolul al treilea cuprinde mai multe detalii de implementare şi de structurare a proiectului.

În final, capitolul al patrulea descrie câteva măsuri de securitate pe care le-am luat, în urma publicării proiectului pe un server ce foloseşte sistemul de operare Linux.

Contribuţii

Aplicaţia foloseşte, la nivel de client, AngularJS ca framework de Javascript, iar pentru funcţionalitatea şi interacţiunea cu utilizatorul, prezentate în Capitolul I, am realizat o implementare modulară, în care am definit propriile componente.

Pentru editorul de text, am folosit o librărie gratuită, cu ajutorul căreia am creat câte un mod de definire a sintaxei, pentru fiecare instrument din cele existente în proiect. Pe partea de backend, am realizat în principiu execuția sau întreruperea rulării programelor, precum şi încărcarea exemplelor care au fost definite pe paginile de documentaţie ale fiecărui instrument, şi salvate local, la nivelul proiectului.

Capitolul I

Funcţionalitate

1. Introducere

Trecerea de la interfețele bazate pe linia de comandă la o interfaţă web ajută utilizatorul să înţeleagă şi să folosească mult mai uşor unele programe, câştigând astfel în utilizabilitate. În plus, acţiunile dorite sunt realizate mult mai uşor în cazul unei interfeţe simple, chiar dacă acestea sunt limitate.

Scopul proiectului realizat este de a permite utilizatorului să execute orice programe ce rulează de la linia de comandă, având ca date de intrare comenzile scrise într-un fişier de text, şi vizualizarea rezultatelor obţinute. Aplicaţia web realizată oferă suport de execuţie pe două sisteme de operare, deoarece implementarea locală a fost făcută folosind un server Apache ce a rulat pe Windows, iar publicarea proiectului a fost făcută mai târziu, pe un server de Linux.

2. Prezentare

ToolRunner oferă utilizatorului o interfaţă cu ajutorul căreia să poată executa RMT, Z3, sau orice alte programe similare, care primesc la intrarea standard date într-un format corespunzător şi produc la ieşirea standard un text care reprezintă rezultatul în urma execuţiei de la linia de comandă.

Proiectul permite scrierea datelor de intrare pentru fiecare instrument în parte, într-un editor de text ce oferă o sintaxă ce evidenţiază cuvintele cheie sau comentariile pentru fiecare instrument, perechi care se formează automat la scrierea acoladelor sau ghilimelelor, indentări şi o numerotare a rândurilor scrise. Pentru afişarea rezultatului există o secţiune separată ce poate fi ascunsă sau afişată în pagină.

Aplicaţia conţine câteva exemple predefinite care pot fi încărcate uşor în editorul de text spre a fi executate sau modificate. Exemplele sunt stocate în fişiere locale, având conţinutul citit

la nivel de backend şi afişat la nivel de client, pe partea de frontend. Asemănător cu secţiunea de afişare a rezultatului, exemplele pot fi ascunse pentru o vizibilitate cât mai bună asupra editorului, chiar şi pe dispozitive cu ecran mic.

În momentul execuţiei, datele de ieşire vor fi afişate pe măsură ce sunt produse de către program. Aşteptarea poate fi de lungă durată, iar în acest caz utilizatorul poate urmări progresul sau ar putea să întrerupă execuţia comenzii pentru a-şi continua lucrul, fără să rămână blocat până la finalizarea rezultatului.

Sunt permise folosirea mai multor instrumente ce se pot executa de la linia de comandă (în acest caz RMT, Z3), într-un mod asemănător. Schimbarea unui instrument se poate face fie din meniul de tip `dropdown` vizibil în pagină, fie prin schimbarea URL-ului, cu numele instrumentului corespunzător, ca în exemplul de mai jos:

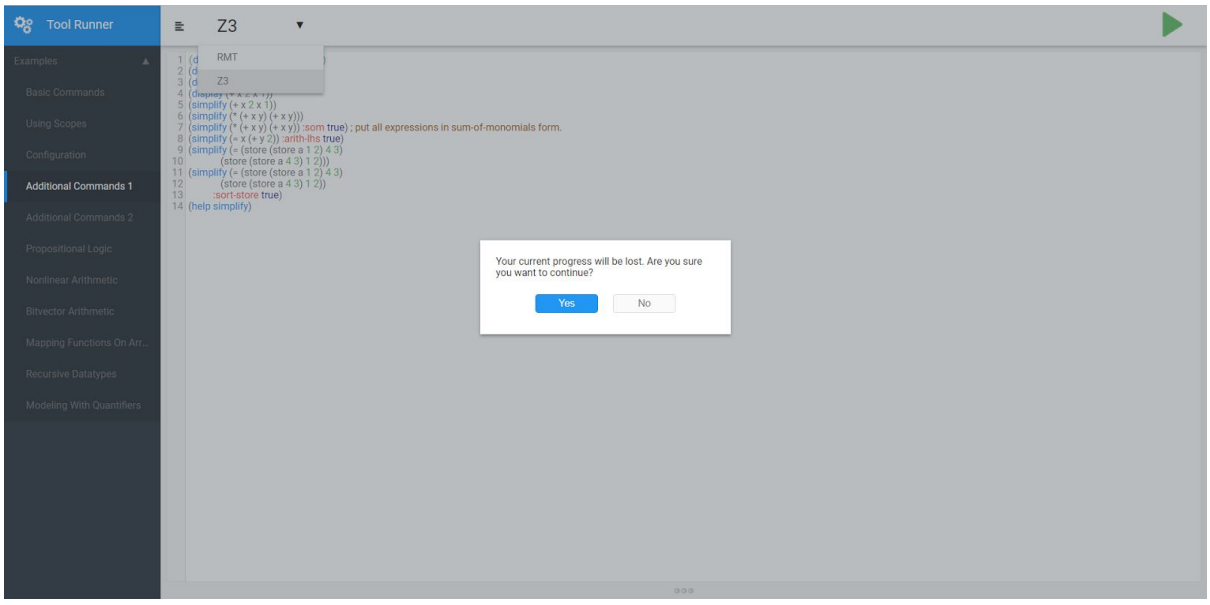
De la: `http://fmse.info.uaic.ro:9909/licenta/frontend/app/#/rmt`,
la `http://fmse.info.uaic.ro:9909/licenta/frontend/app/#/z3`.

În momentul unei astfel de schimbări, se reîncarcă view-ul cu noile date pentru instrumentul ales, însă fără să se facă refresh. Întreaga aplicaţie funcţionează pe o singură pagină, iar comportamentul rămâne acelaşi pentru fiecare instrument în parte.

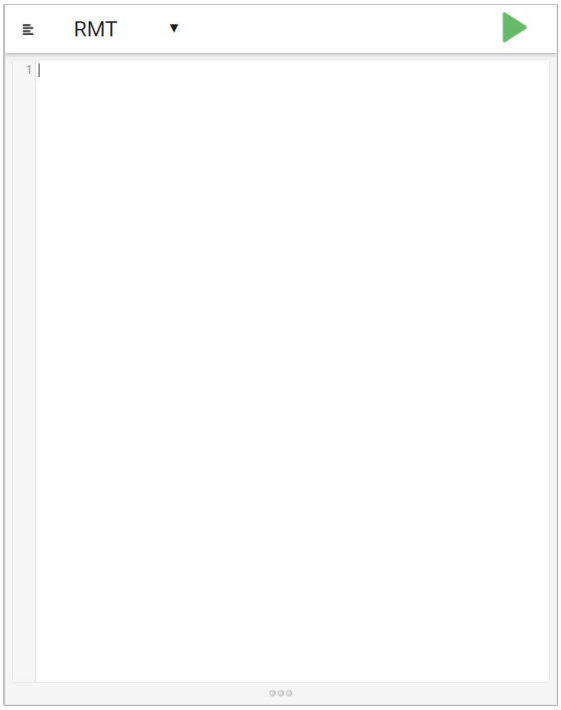
Astfel, sistemul devine flexibil şi permite folosirea altor instrumente care funcţionează după acelaşi principiu.

Designul ales permite o înţelegere uşoară a aplicaţiei, precum şi o utilizare a acesteia axată pe partea de a scrie propriul program, spre a fi executat. Următoarele imagini prezintă interfaţa folosită pentru câteva stări ale aplicaţiei.

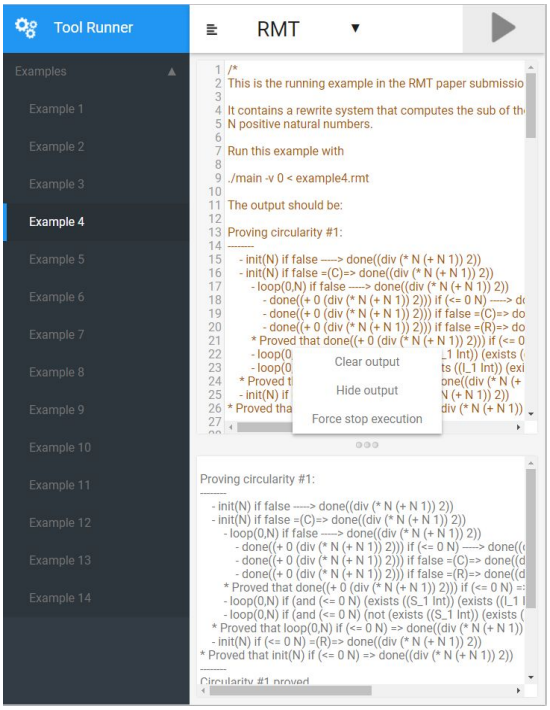
Figură: Starea inițială a aplicației



Figură: Mesajul de confirmare la schimbarea unui tool din meniul de tip `dropdown`



Figură: Focalizare asupra editorului de text



Figură: Momentul execuţiei unui exemplu RMT

3. Tipuri de instrumente

Iniţial, sunt câteva instrumente predefinite (şi anume RMT şi Z3), dar pot fi adăugate relativ uşor şi altele. În continuare, sunt prezentate instrumentele predefinite. RMT este un program scris în C++ care poate demonstra proprietăţi de accesibilitate în termeni cu constrângeri, iar Z3 poate fi folosit pentru a verifica satisfiabilitatea unor formule logice peste una sau mai multe teorii.

Pentru a folosi RMT, este nevoie de descărcarea programului, împreună cu Z3, iar în final compilarea proiectului într-o versiune recentă de Visual Studio pentru obţinerea executabilului. Pentru execuţie se folosesc comenzile:

- `rmt.exe < example.rmt`, în cazul sistemului de operare Windows
- `./rmt < example.rmt`, pentru sistemul de operare Linux

Z3 poate fi rulat apoi prin comanda ”`z3 < example.z3`”, unde ”example.z3” este fişierul ce conţine programul scris în z3, primit la intrarea standard.

Aplicaţia execută aceste comenzi în funcţie de sistemul de operare de pe care rulează serverul folosit, şi în plus, redirecţionează rezultatul la ieşirea standard spre a fi citit şi afişat utilizatorului.

Capitolul II

Configurarea proiectului

1. Tehnologii folosite

Implementarea aplicației ToolRunner pune accentul pe interfața grafică și pe interacțiunea cu utilizatorul. În proiect am folosit HTML (Hypertext Markup Language) împreună cu SASS (Syntactically Awesome StyleSheets) pentru crearea unor template-uri folosite în directivele de AngularJS. Astfel, pagina este redată dinamic în funcție de acțiunile utilizatorului.

HTML oferă structura semantică a elementelor, iar cu SASS am definit variabile, mixin-uri, am utilizat operatori și, cel mai important, încapsularea selectorilor. SASS este un preprocesor de CSS și oferă o opțiune de importare a altor fișiere SCSS ce sunt combinate într-un singur fișier CSS inclus în aplicație, cu scopul de a reduce numărul cererilor HTTP pentru fiecare dintre acestea, la unul singur.

Mixin-urile ajută la reutilizarea unui grup de stiluri, oferind și flexibilitate datorită parametrilor. Am descris prin mixin-uri proprietățile care necesită prefixe specifice mai multor tipuri de navigatoare web pentru suport, cum ar fi: calc, border-radius, box-shadow, transition, user-select, background-size, transform, animation.

Astfel, dimensiunile fișierelor SCSS au fost reduse și am evitat repetițiile. Un exemplu de mixin pe care l-am folosit este următorul:

```
@mixin transition($transition...) {
     -moz-transition: $transition;
     -o-transition: $transition;
     -webkit-transition: $transition;
     transition: $transition;
}
```

”-moz-” este specific motorului de randare pentru navigatorul Firefox, ”-webkit-” pentru Chrome şi Safari, iar ”-o-” pentru Opera. Acestea ajută la evitarea inconsistenţelor dintre navigatoare, în cazul anumitor proprietăţi mai noi de CSS. Pentru ca această declaraţie să fie folosită, se include următoarea linie SCSS ca proprietate unui selector, împreună cu parametrii necesari:

```
@include transition(height .2s ease);
```

Javascript este un limbaj de programare interpretat, ce rulează la nivel de client într-o aplicaţie web. Am ales să folosesc ca framework AngularJS deoarece am vrut în primul rând să implementez o aplicaţie pe o singură pagină (de tip `Single Page Application`). În al doilea rând, m-a ajutat să îmi structurez proiectul pe componente cu ajutorul modulelor, ce pot fi importate ca dependenţe şi refolosite. Tot cu ajutorului framework-ului am realizat comunicarea dintre client şi server prin folosirea serviciului `$http`, şi navigarea între instrumente cu ajutorul serviciului `$routeProvider`.

La nivel de server, am folosit ca limbaj de programare PHP (Hypertext PreProcessor) pentru execuţia sau întreruperea rulării instrumentului ales, atât pentru sistemul de operare Windows, cât şi pentru Linux. De asemenea, la nivel de server se citeşte şi conţinutul exemplelor existente în fişierele locale, spre a fi afişate la utilizator, în editorul de cod.

2. Instalare

Pentru a putea utiliza tehnologiile folosite în proiect, am instalat câteva dependenţe. Ruby este necesar pentru a instala SASS, iar Node.js este folosit pentru a crea un pachet de gestionare a modulelor.

Acest pachet este un JSON (Javascript Object Notation) şi cuprinde pe lângă altele, şi proprietatea `dependencies`, unde se găsesc toate dependenţele necesare proiectului şi care pot fi instalate cu o singură comandă: `npm install`. Mai jos am redat conţinutul acestui fişier.

```
{
    ”name”: ”licenta”,
```

```
    "version": "1.0.0",
    "author": "Anghelache Oana",
    "description": "",
    "main": "frontend/bundle.js",
    "scripts": {
        "build-js": "browserify frontend/app/app.js >
    frontend/bundle.js",
        "watch-js": "watchify frontend/app/app.js -o
    frontend/bundle.js",
        "build-css": "sass frontend/app/app.scss
    frontend/app/app.css",
        "watch-css": "sass—watch
    frontend/app/app.scss:frontend/app/app.css",
        "watch-all": "npm-run-all -p watch-js watch-css"
    },
    "dependencies": {
        "normalize.css": "5.0.0",
        "angular": "1.5.8",
        "angular-route": "latest",
        "codemirror": "latest",
        "angular-ui-codemirror": "latest",
        "browserify": "latest",
        "watchify": "latest",
        "npm-run-all": "latest"
    }
}
```

Acest fişier l-am creat prin comanda `npm init`, ce iniţializează pachetul cu un nume, o versiune, autorul, descrierea şi o proprietate numită `main`, ce conține calea fişierului principal de Javascript.

În momentul în care am folosit o nouă librărie, am instalat-o folosind comanda `npm install <numele librăriei> --save`. Flag-ul `--save` o adaugă automat în acest `package.json`, ca valoare a proprietăţii `dependencies`.

Voi descrie în continuare lista dependenţelor folosite în proiectul ToolRunner:

- `normalize.css`, utilizat pentru a reseta stilurile CSS ce diferă de la un browser la altul;
- `angular`, ca framework de Javascript;
- `angular-route`, pentru navigarea între diferite view-uri;
- `CodeMirror` pentru editorul de text;
- `angular-UI-CodeMirror` pentru a putea utiliza librăria `CodeMirror` în Angular, sub forma unei directive;
- `browserify`, pentru importarea în mod recursiv, a tuturor fişierelor de Javascript, create în mod dependent, într-un singur fişier;
- `watchify`, pentru actualizarea automată a fişierului Javascript ce conţine toate dependenţele;
- `npm-run-all`, ce permite rularea mai multor scripturi npm în paralel sau secvenţial;

Cu `browserify`, am putut folosi metoda `require()`, definită de Node.js, ce importă toate fişierele de Javascript, în mod recursiv, în unul singur, inclus apoi în aplicaţie sub denumirea de `bundle.js`. `Browserify`, împreună cu `watchify`, ajută la actualizarea acestui fişier principal, de fiecare dată când apar modificări în Javascript, în timpul implementării. Mai jos am explicat rolul pe care îl au scripturile definite, după care legătura acestora cu dependenţa numită `npm-run-all`.

În acest JSON am creat câteva script-uri care ajută la rularea unor comenzi mai complexe, printr-o altă comandă, ce se execută cu `npm run <numele scriptului creat>`. Comenzile următoare trebuie rulate din directorul în care se află package.json.

- `npm run build-js` execută comanda `"browserify frontend/app/app.js > frontend/bundle.js"`; Aceasta generează fişierul `bundle.js` ce cuprinde toate fişierele Javascript, implementate ca dependenţe în `app.js`.

- `npm run watch-js` execută comanda `"watchify frontend/app/app.js -o frontend/bundle.js"`, asemănătoare cu cea de mai sus, exceptând faptul că se urmăresc modificările aduse în orice fişier Javascript, pentru a putea fi actualizate automat în `bundle.js`. Această metodă se recomandă la implementare.

- `npm run build-css` execută comanda `"sass frontend/app/app.scss frontend/app/app.css"` ce va crea fişierul `app.css` din `app.scss`. Acesta va cuprinde toate stilurile folosite.

- `npm run watch-css` execută comanda `"sass—watch frontend/app/app.scss:frontend/app/app.css"`, de creare automată a stilurilor CSS, având acelaşi rol pe care îl are `Watchify` pentru Javascript. Această comandă este recomandată la implementarea stilurilor.

- `npm run watch-all` execută comanda `"npm-run-all -p watch-js watch-css"`, ce rulează în paralel cele două comenzi de `watch-js` şi `watch-css` definite mai sus. Astfel, orice modificări aduse fie într-un fişier Javascript, fie într-un fişier SCSS, vor fi identificate pentru o nouă creare a bundle-ului, respectiv CSS-ului, în mod automat.

În ultimul caz, a fost nevoie de instalarea dependenţei `npm-run-all` pe care am amintit-o mai sus. Toate aceste comenzi care nu folosesc `npm` necesită locaţia unui fişier la intrarea standard şi locaţia pentru un fişier la ieşirea standard ce trebuie să existe într-un anumit director. Pentru implementare, este mai uşoară şi mai sigură folosirea comenzilor ce rulează cu npm.

Pentru a aduce proiectul local, pe o nouă maşină, indiferent de sistemul de operare, acesta trebuie configurat astfel:

1. Se instalează: Ruby (versiunea > 2.2.5), NodeJs (versiunea > 6.9.4), PHP, Git;
2. Este necesară clonarea proiectului aflat la adresa https://github.com/oana-a/licenta;
3. Rularea următoarelor comenzi din directorul ”licenta/frontend/app”:
    a. `npm install`
    b. `npm run build-js`
    c. `npm run build-css`
4. Configurarea unui server Apache pentru PHP;
5. Instalarea şi adăugarea executabilelor pentru instrumentele folosite de aplicaţie (RMT, Z3), în directorul ”licenta/tools/RMTsrc”, respectiv ”licenta/tools/z3”. Acestea trebuie să poată fi executate în funcţie de sistemul de operare pe care rulează serverul.

Pentru a actualiza orice modificări aduse asupra componentelor scrise la nivel de frontend, se execută mai întâi, următoarea comandă:

- `npm run watch-all`, din directorul ”licenta/frontend/app”.

Pentru publicarea proiectului în urma finalizării implementării, este necesară realizarea următorilor paşi:

1. Configurarea serverului Apache, şi eventual instalarea PHP-ului;
2. Copierea proiectului ce trebuie să conţină directoarele ”backend”, ”frontend” şi ”tools”;
3. Adăugarea în directorul ”licenta/node_modules” dependenţele necesare, fie manual, fie prin rularea comenzii `npm install` (în acest caz, trebuie să existe fişierul ”licenta/package.json” şi să fie instalat NodeJS).
4. Instalarea fiecărui instrument şi înlocuirea executabilelor din directorul ”tools”, cu cele corespunzătoare şi funcţionale;
5. În cazul sistemului de operare Linux, este necesară instalarea platformei Docker din motive de securitate, detaliate la capitolul IV.

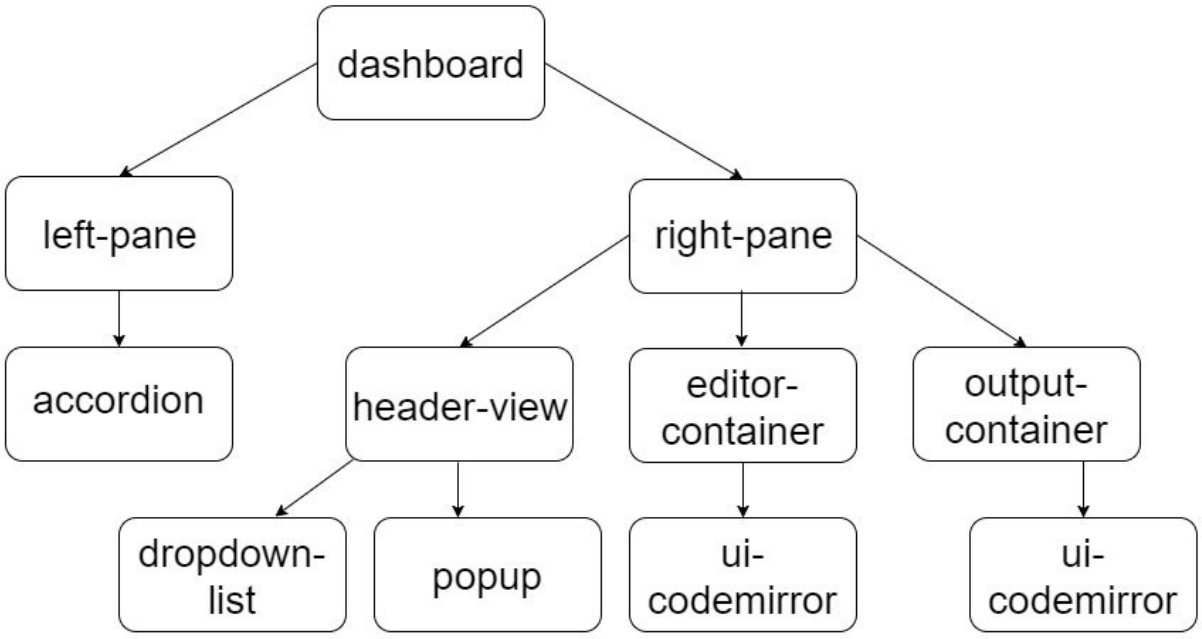
În concluzie, toate configurările făcute cu ajutorul unui `package.json`, uşurează întregul proces de implementare, instalare, întreţinere sau publicare a acestui proiect, cu dependenţele corecte.

Capitolul III

Detalii de implementare

1. Arhitectura aplicaţiei

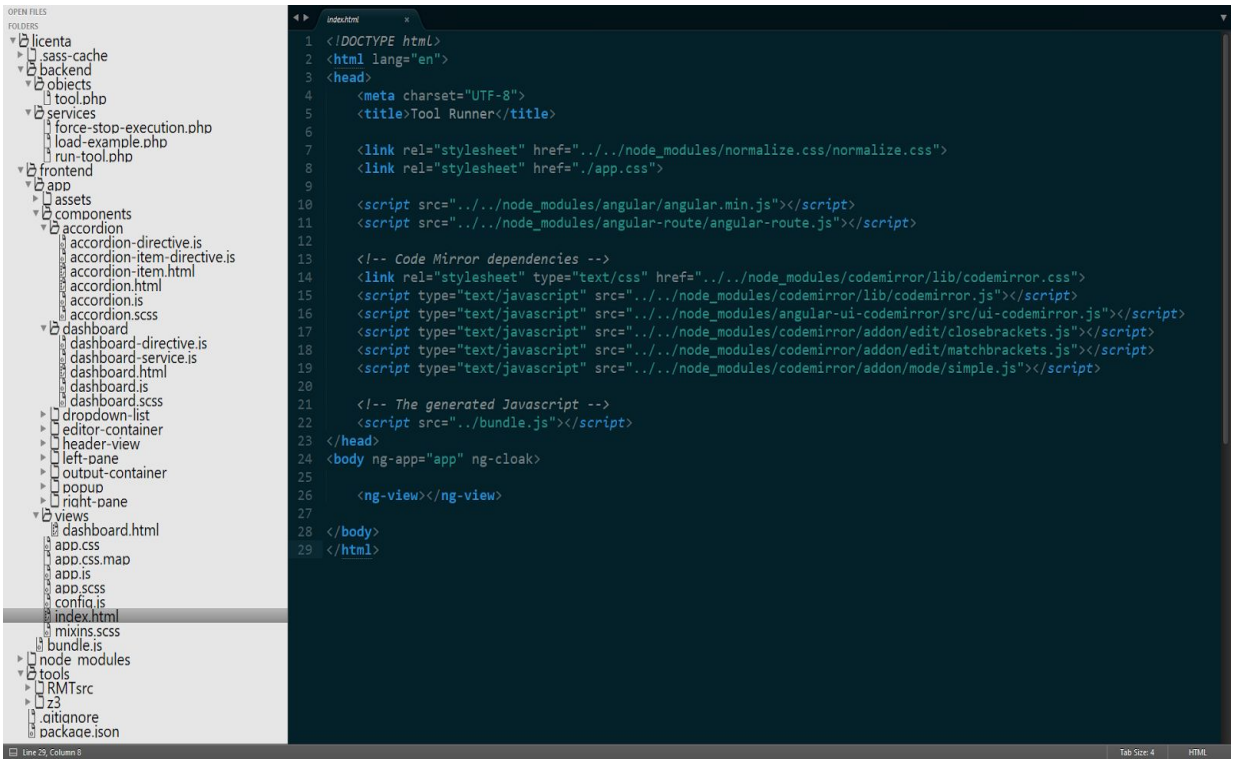
Proiectul ToolRunner este format din mai multe tipuri de componente ce comunică între ele şi formează o structură ierarhică care alcătuieşte întreaga aplicaţie, de la componente de prezentare şi independente, la componenta principală `dashboard`. Am prezentat mai jos această structură ierarhică.



Diagramă: Ierarhia aplicaţiei

În AngularJS, am creat câte un modul pentru fiecare componentă pentru a putea încapsula directivele din care a fost alcătuită, serviciile existente, şi pentru a include dependenţele cu alte module.

Structura fişierelor este cea prezentată mai jos, cu toate componentele create. Partea de frontend este separată de cea de backend, comunicarea făcându-se prin cereri de tip HTTP.



Figură: Structura fişierelor

Fiecare componentă poate conţine:

- Un fişier Javascript cu numele corespunzător, în care trebuie creat modulul, împreună cu importarea fişierelor dependente.
- Una sau mai multe fişiere Javascript pentru fiecare directivă definită;
- Opţional, un serviciu;
- Câte un fişier HTML definit ca template pentru fiecare directivă;
- Un fişier SCSS de stilizare a componentei;

În modulul principal, element de bază ce constituie aplicaţia, am adăugat componenta centrală şi configurarea navigării. View-ul componentei principale, numită `"dashboard"`, l-am separat într-un alt director pentru a fi reutilizat la navigarea în pagină.

Am implementat câteva componente mici şi reutilizabile, care nu au alte dependenţe, cum ar fi acordeonul pentru afişarea exemplelor, popup-ul de confirmare ce apare când se schimbă instrumentul, pentru atenţionarea asupra faptului că utilizatorul va pierde progresul sau rezultatele curente, şi componenta `dropdown-list`. Acestea sunt de prezentare deoarece nu fac altceva decât să genereze un view pentru anumite date de intrare, comportamentul fiind mereu acelaşi.

Alte componente, cum ar fi containerul editorului de text, sunt mai complexe deoarece trebuie să primească informaţii cu exemplul ce trebuie încărcat, sau dacă programul urmează să fie executat. În acest caz, este necesar numele instrumentului selectat. Toate aceste informaţii le primeşte prin comunicarea cu celelalte componente. Comportamentul diferă în funcţie de informaţiile primite, iar ca rezultat poate afecta şi alte părţi ale aplicaţiei.

Directivele implementate conţin un controller de manipulare a datelor de pe `scope`, un `templateUrl` al HTML-ului ce generează view-ul dinamic şi tipul de afişare a directivei, şi anume ca element HTML. În plus, mai poate conţine un scope în cazul în care componenta trebuie să primească anumite date de la componenta părinte, sau o funcţie de link pentru a selecta sau manipula elemente din DOM, după ce view-ul a fost încărcat.

Implementarea unui serviciu a fost făcută la nivelul componentei de dashboard, iar ca rol principal, ea se ocupă de expunerea metodelor ce creează cereri HTTP. Rezultatul este o promisiune ce trebuie rezolvată din controller, de către funcţia care foloseşte o metodă de acest tip. În plus, am mai folosit serviciul pentru a face posibilă comunicarea între mai multe componente ce împart aceleaşi informaţii, dat fiind faptul că acesta poate fi accesat oriunde în aplicaţie.

Tot pe partea de frontend, am inclus un director cu numele `"assets"` pentru imagini şi am separat într-un alt fişier toate mixin-urile, adăugate apoi la stilurile principale de SCSS, pentru a putea fi folosite oricând.

Toate instrumentele ce sunt executate de aplicaţie, au fost instalate în directorul numit `"tools"`. Tot aici am adăugat şi exemplele care au fost incluse în proiect, precum şi definirea unui `Dockerfile` pentru a crea imaginea din care se obţine un container în cazul folosirii

proiectului publicat pe serverul de Linux. Mai multe informaţii pot fi citite la capitolul ”Securitate”.

La nivel de backend, am creat câte un fişier pentru fiecare cerere de tip HTTP ce va extrage datele de intrare primite ca parametri de tip ”query string”, dintr-un format JSON. În continuare, va apela o metodă dintr-o clasă PHP care rezolvă acţiunea dorită şi trimite înapoi răspunsul transformat în alt JSON.

Prin structura modulară a proiectului, dezvoltarea aplicaţiei devine uşor de înţeles şi de menţinut în cazul în care ar trebui introduse noi funcţionalităţi.

2. Navigarea

Pentru a folosi modulul ngRoute din AngularJS ce se ocupă de navigare, am instalat librăria ”angular-route” cu npm. Am inclus ca dependenţă ngRoute în modulul principal al aplicaţiei. Directiva ng-view am introdus-o ca element în body-ul din fişierul ”index.html”. Aici se va încărca HTML-ul ce cuprinde întreaga aplicaţie, iar în funcţie de URL, se va face o altă iniţializare a datelor pentru unele componente.

În fişierul ”config.js” am definit configurările pentru $routeProvider, şi anume ce template trebuie încărcat pentru navigarea la o anumită cale din URL, aşa cum apare în structura de mai jos:

```
angular.module("app")
.config(function($routeProvider) {
     $routeProvider
          .when('/rmt', {
               templateUrl: "./views/dashboard.html"
          })
          .when('/z3', {
               templateUrl: "./views/dashboard.html",
          })
          .otherwise({
```

```
                redirectTo: '/rmt'
            });
});
```

Am folosit acelaşi template pentru fiecare URL, şi anume componenta "dashboard", ce cuprinde întreaga aplicaţie. Calea, în schimb, diferă de la un instrument la altul şi se compară cu calea din URL urmată de caracterul "#". În momentul în care nu se găseşte nicio potrivire, se face o redirecţionare către calea pentru RMT, datorită opţiunii `otherwise()`, definită.

De exemplu, URL-ul:

`http://fmse.info.uaic.ro:9909/licenta/frontend/app/#/z3`, va încărca componenta cu datele pentru instrumentul "z3", cale găsită în fişierul de configurare.

Când utilizatorul schimbă instrumentul pe care vrea sa îl execute, se iniţializează directiva "`app.dashboard`" care extrage din URL şi salvează într-un serviciu numele instrumentului ales. Implicit, se realizează iniţializarea tuturor celorlalte directive, fiind dependente unele de altele. Astfel, voi şti ce program trebuie executat, sau ce exemple trebuie încărcate în funcţie de alegerile utilizatorului şi calea URL-ului.

Pentru a extinde funcţionalitatea la alte instrumente, se urmează acelaşi şablon, şi anume, se foloseşte aceeaşi componenta principală ca view, însă se alege o altă cale în fişierul de configurare şi se fac modificările necesare pe partea de frontend, respectiv backend, pentru noua funcţionalitate.

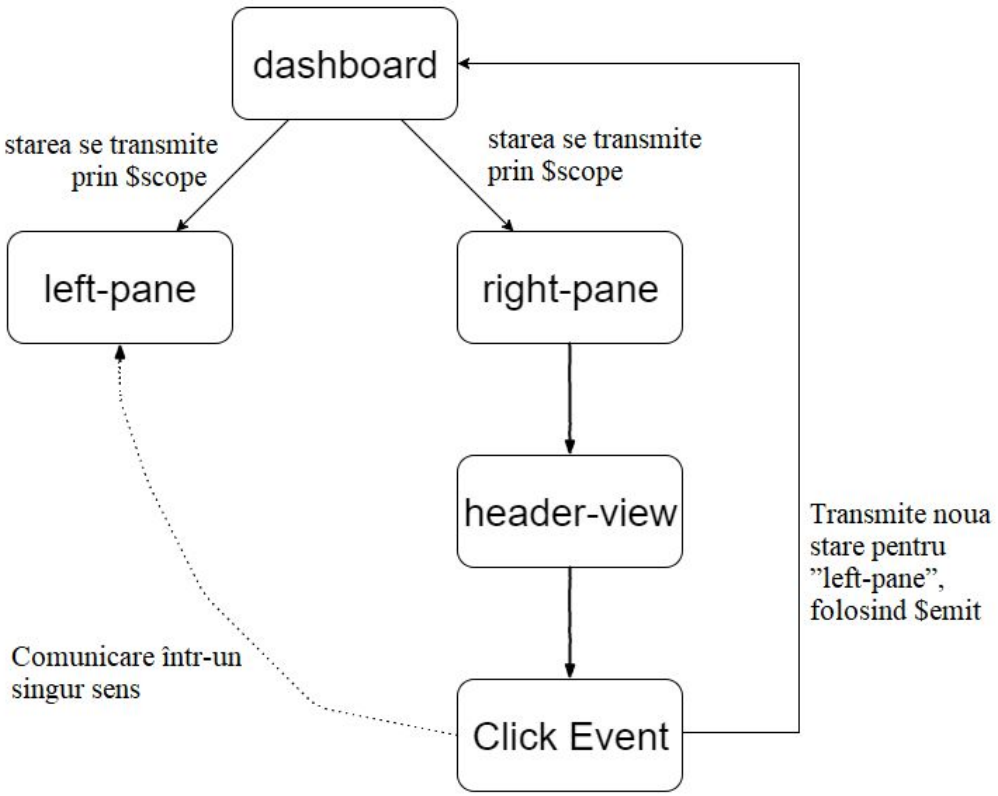
3. Comunicarea dintre componente

Comunicarea între componentele din Angular am realizat-o în funcţie de locul pe care îl ocupă în ierarhie. Aceasta poate fi de la copil la părinte, sau invers, unidirecţională sau bidirecţională, dar şi între componente ce se află pe acelaşi nivel în ierarhie, adică cele care au acelaşi părinte.

În cazul în care o componentă vrea să transmită părintelui informaţii, am folosit "callback-urile", metode expuse din cadrul componentei curente prin "scope-ul" directivei, şi definite la părinte. Această abordare are avantajul de a încapsula funcţionalitatea componentei şi

de a expune doar acţiuni ce pot fi definite din afară. Aşa am implementat şi componentele de prezentare a unui view, adică cele independente. Pentru a transmite informaţii la un copil, m-am folosit tot de ”scope-ul” directivei, datele fiind transmise ca atribute. Astfel, am realizat şi tipul de comunicare bidirecţională, ce foloseşte aceeaşi referinţă la date, spre a fi mereu sincronizate.

O altă abordare de comunicare este cea cu ajutorul serviciilor `$emit` şi `$broadcast` din Angular. Serviciul $emit transmite datele prin evenimente la toate componentele aflate mai sus din ierarhie, iar $broadcast la toţi copiii. M-am folosit de serviciul $emit pentru a face posibilă comunicarea dintre două componente de pe acelaşi nivel. Pentru a închide/deschide panoul din stânga, a trebuit să extrag informaţia de la un click din header-ul panoului din dreapta, la ambele componente: ”left-pane” şi ”right-pane”. În imagine am detaliat printr-o structură vizuală componentele implicate.



Diagramă: Dashboard-ul şi componentele implicate în comunicare

La evenimentul de click, am transmis prin $emit un parametru cu starea inițială a panoului din stânga al aplicației. În controller-ul directivei ”dashboard”, care are cel mai înalt nivel în ierarhie, am salvat această informație și am transmis-o mai departe la cele doua componente descendente, ”left-pane” și ”right-pane”, pentru ca panelul din stânga să se închidă sau deschidă, iar panelul din dreapta să se mărească respectiv, micșoreze printr-o tranziție. Folosind serviciul $emit, am transmis datele direct la componenta centrală chiar dacă aceasta se află cu două nivele mai sus în ierarhie.

Deși este o soluție bună pentru acest caz, am evitat folosirea excesivă a acestor servicii deoarece evenimentele trec prin mai multe componente, și pot fi gestionate în mai multe locuri, fără ca dezvoltatorul să își dea seama foarte ușor de acest lucru, în special în cazul unei aplicații de dimensiuni foarte mari. Atunci când sunt folosite mult serviciile $emit și $broadcast pentru comunicare, aplicația devine foarte greu de întreținut din cauza acestor metode greu de depistat prin care trec evenimentele. Astfel, se poate ajunge la un comportament neașteptat și greu de rezolvat.

O altă metodă prin care am comunicat între componente, este cea prin serviciul implementat de mine, în `dashboard`, la nivelul aplicației. Toate componentele pot avea acces la acest serviciu și la informațiile pe care le expune, printr-o simplă importare în directivă. Comunicarea se realizează foarte ușor, și de aceea am salvat aici numele instrumentului curent, informație folosită în mai multe componente, structura informațiilor din acordeonul aplicației, și alți câțiva parametri necesari la încărcarea sau execuția unui exemplu. Scopul serviciului este de a realiza cereri de comunicare cu backendul, însă conține și informații ce trebuie partajate între mai multe componente dependente. Comunicarea se face în acest caz, indiferent de nivelul acestora în ierarhie.

În concluzie, am încercat să realizez comunicarea, de cele mai multe ori prin callback-uri sau date transmise ca atribute direct la scope-ul directivei, pentru a putea urmări și înțelege mai ușor cursul informațiilor transmise.

4. Comunicarea dintre Frontend şi Backend

Comunicarea frontend - backend se realizează prin cereri HTTP, cu ajutorul unor date de tip JSON. Cererile HTTP făcute sunt de tip POST în cazul execuţiei unui instrument, respectiv întreruperea acestei rulări, şi de tip GET, realizată în cazul încărcării unui exemplu în editorul de text sau pentru a citi rezultatele din timpul execuţiei.

Am folosit serviciul `$http` din Angular pentru facilitarea comunicării. Informaţiile le-am trimis într-un obiect, ca parametri de tip ”query string” la cerere, pentru ca serviciul să serializeze automat datele ce ajung la backend într-un JSON, respectiv primirea deserializată a răspunsului.

Cererea HTTP se face la un fişier de PHP în cazul în care vreau să rezolv o anumită acţiune pe partea de backend. Aici, setez în primul rând header-ul răspunsului, să aibă conţinutul de tip application/json, după care decodific răspunsul primit, într-o variabilă de PHP. Cu aceste date, apelez o metodă ce rezolvă acţiunea, iar răspunsul îl codific înapoi în JSON pentru a fi trimis la nivel de frontend ca răspuns. Serviciul folosit în Angular returnează o promisiune spre a fi rezolvată după ce am primit răspunsul. Cererea se realizează într-un mod asincron, deoarece rezultatul de la server nu vine imediat, iar în plus, ar putea să întâmpine o eroare. Astfel, promisiunea trebuie rezolvată folosind funcţia `then()`, ce primeşte ca parametri metodele ce se apelează în caz de succes, sau în caz de eroare.

Pentru a rezolva problema afişării în timp real a răspunsurilor parţiale în timp ce programul rulează, m-am folosit de un fişier pentru a scrie şi citi aceste date. Din directivă, am apelat o metodă ce face cererea POST către backend, împreună cu doi parametri: conţinutul din editorul de text, de tip string, şi numele instrumentului ce trebuie executat. În PHP, apelez metoda ce execută un program extern de la linia de comandă în funcţie de sistemul de operare şi numele instrumentului. Rezultatul este redirecţionat către un nou fişier de ieşire, pentru ca datele să poată fi accesate din afară. În acelaşi timp, din momentul în care se face cererea, din Angular, se execută o altă metodă din 500 în 500 de milisecunde, care citeşte acest fişier scris din PHP, folosind o altă cerere de tip GET. Răspunsul găsit în fişier este afişat utilizatorului. Când programul îşi termină execuţia, se mai citeşte o dată fişierul pentru a obţine un răspuns integral,

iar în final, revoc execuţia metodei repetitive. Astfel, utilizatorul primeşte câte un răspuns parţial pentru a şti ce se întâmplă în timpul execuţiei ce poate fi de durată.

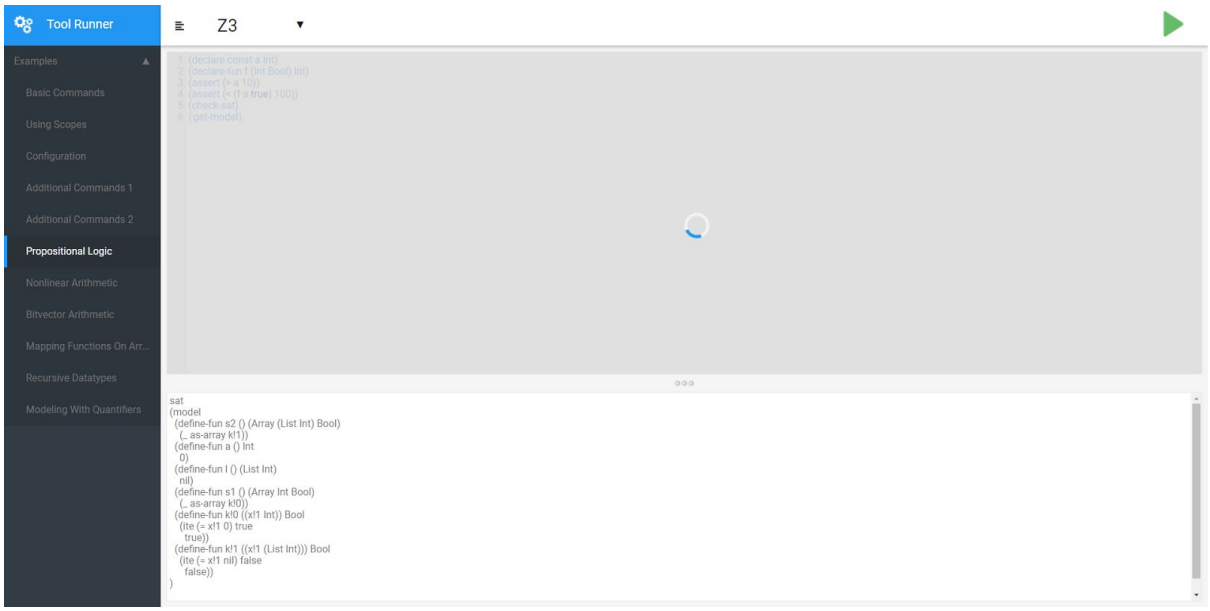
Rularea tuturor instrumentelor, respectiv oprirea, se realizează cu ajutorul metodei `"shell_exec()"` din PHP, ce execută o comandă primită ca parametru de tip string. Ca exemplu, am descris mai jos linia de cod folosită la pornirea unei execuţii a unui instrument şi redirectarea rezultatului la un fişier extern:

```
shell_exec('rmt.exe < examples/new-rmt-example.rmt > pending-rmt-results.txt');
```

În momentul rulării programelor, am dezactivat butonul ce porneşte execuţia pentru a evita cererile multiple de acelaşi tip ce ar fi putut bloca întreaga aplicaţie. Am adăugat în schimb o opţiune în meniul secţiunii cu rezultatele obţinute, ce opreşte rularea instrumentului. Această funcţionalitate este utilă în cazul în care execuţia devine de lungă durată, iar utilizatorul se gândeşte să modifice codul pentru a reporni rularea aplicaţiei. Oprirea este o nouă comandă ce se execută în funcţie de sistemul de operare de pe care rulează serverul. Pentru RMT, am folosit următoarele comenzi:

- `shell_exec('taskkill /F /IM "rmt.exe"')`, la nivel de Windows,
- `shell_exec('killall -KILL rmt')`, în cazul sistemului de operare Linux.

Pentru a încărca un exemplu, am făcut o cerere GET la un fişier PHP care citeşte conţinutul exemplului salvat local, în funcţie de numele acestuia şi numele instrumentului. Până când se execută această cerere, iar răspunsul este primit, am afişat o secţiune de loading peste editorul de text, pentru a nu întâmpina unele funcţionalităţi neaşteptate. De exemplu, utilizatorul ar putea să înceapă să scrie în editor sau să încarce repede mai mai multe exemple, iar la execuţie să obţină rezultate neaşteptate, sau să se schimbe brusc conţinutul editorului de text în timpul folosirii aplicaţiei. Încărcarea de pe pagină blochează orice acţiune în momentul şi zona respectivă, iar comportamentul devine cel aşteptat, evitându-se cazurile neprevăzute, din exemplu.



Figură: Blocarea editorului în momentul încărcării unui exemplu

Un alt caz special ce ar putea bloca cursul evenimentelor, este ca utilizatorul să execute un program după care să schimbe instrumentul în timp ce acesta încă rulează. Aici, am forţat oprirea execuţiei pentru ca aplicaţia să funcţioneze ca la început, având toate datele resetate.

Prin această structură şi mod de comunicare, există o separare bine definită între backend şi frontend, deoarece orientarea proiectului este îndreptată mai mult pe partea dezvoltării aplicaţiei la nivel de client.

5. Editorul de text

Pentru a crea editorul de text, am folosit librăria CodeMirror, specializată pe a edita cod scris în navigatorul Web. Această librărie oferă şi alte extensii pentru anumite limbaje de programare dar şi mai multe teme de stiluri CSS, însă eu am definit propriile moduri pentru programele incluse, şi propriile culori în aplicaţie.

Editorul de text este implementat pentru ca utilizatorul să poată scrie codul într-un mediu de lucru cât mai confortabil. Am oferit ca funcţionalitate diferenţierea prin culori a cuvintelor

cheie sau a unor tipuri de variabile, indentarea cu ajutorul taburilor, numerotarea rândurilor şi închiderea automată a parantezelor, respectiv identificarea perechilor de acest tip.

Pentru a include CodeMirror în Angular, am folosit directiva `"ui-codemirror"` ce foloseşte această librărie. Pentru configurare, am adăugat ca parametri câteva opţiuni pentru editor. În primul rând am folosit modul creat pentru fiecare instrument în parte. Un mod este un fişier Javascript care defineşte un vocabular cu ajutorul expresiilor regulate ce sunt identificate printr-un cuvânt de tip string, numit ”token” şi care prezintă un anumit stil pe care l-am suprascris în SCSS pentru respectarea unor standarde în culorile aplicaţiei.

Am ales o interfaţă cât mai simplă a modului, fapt pentru care am folosit o extensie a librăriei CodeMirror, numită ”mode/simple”. Aceasta m-a ajutat să definesc modul printr-o structură declarativă uşoară, potrivită pentru început dar în acelaşi timp, suficient de expresivă în a defini ceea ce aveam nevoie.

Prin metoda `"defineSimpleMode"` am creat noi moduri, ce includ un nume, împreună cu stările ce conţin mai multe reguli. Pentru început am definit la instrumentul RMT tipul string ce se află cuprins între ghilimele sau apostrofuri. Expresia regulată pentru primul tip este următoarea: `/"([\W\w]*)"/`

În Javascript, expresiile regulate sunt definite între cele două simboluri `"/"` de început şi sfârşit. Am folosit şi câteva caractere speciale pentru a defini acest şablon, cum ar fi `"\w"` ce include orice caracter alfa-numeric, respectiv `"\W"` reprezentând orice caractere, mai puţin cele alfa-numerice. Astfel între caracterele de ghilimele, putem avea orice tipuri de date, mai puţin linia nouă, pentru a putea fi identificat cu un string în editor.

Pentru cuvintele cheie, operatori şi constante, am descris un şablon simplu cu respectivele cuvinte. Numerele le-am definit ca fiind de tip hexazecimal, pentru şablonul `"/0x[a-f\d]+/"`, unde `\d` reprezintă orice combinaţie de numere de la 0 la 9. Am mai definit cazul numerelor întregi, pozitive sau negative, unde ar putea să apară caracterul ”.” şi cazul numerelor exponenţiale, la întâlnirea caracterului ”e” împreună cu alte cifre.

Pentru a defini stări mai complexe, se poate folosi metoda `"next"` ce realizează trecerea de la o stare la alta prin numele proprietăţii. De exemplu, la definirea unui comentariu, am început în starea `"start"`, unde dacă se identifică caracterele `"/*"`, se schimbă starea la

cea de comentariu, în care se pot identifica orice tip de date. Aici, la întâlnirea caracterelor ″*/″ se face din nou o trecere la starea de la care am pornit, şi anume cea de ″start″, pentru a reveni la regulile definite anterior.

Pe lângă aceste stări, am mai descris încă o proprietate ce se numeşte ″meta″, ce cuprinde informaţiile globale despre mod. Aici am adăugat opţiunea de a nu se indenta stările de tip comentariu.

Diferența principală dintre modurile definite pentru toate celelalte instrumente, au fost cuvintele cheie, unele declarări de variabile şi tipul de a scrie un comentariu. De exemplu, pentru Z3, comentariul începe cu caracterul ”;”, aflat pe o singură linie, însă se poate scrie şi pe mai multe linii, ca în RMT.

Pe lângă definirea modurilor, am inclus în directivă următoarele configurări:

```
$scope.editorOptions = {
     mode: dashboardService.CurrentTool.title,
     lineWrapping: false,
     lineNumbers: true,
     indentWithTabs: true,
     indentUnit: 4,
     matchBrackets: true,
     autoCloseBrackets: true
};
```

Numele modului l-am definit ca fiind acelaşi cu numele instrumentului pe care utilizatorul l-a ales. Proprietatea ″lineWrapping″ poate aduce textul pe rândul următor dacă este prea lung sau, în cazul utilizat, se foloseşte scroll-ul. Vizualizarea numărului de linii este o altă funcţionalitate oferită de librărie şi aplicată ca în exemplul de cod de mai sus. Unitatea de indentare este de patru spaţii, iar restul parametrilor sunt alte funcţionalităţi pentru orice tip de acolade, ce ajută la indentare.

În final, am definit stiluri SCSS de schimbare a culorii fontului pentru fiecare dintre aceste reguli, însă pentru elementele comune, am păstrat stilurile la fel, chiar dacă instrumentele sunt diferite.

Pentru secţiunea de afişare a rezultatului, am folosit aceeaşi librărie, însă cu următoarele opţiuni de configurare:

```
$scope.outputOptions = {
     lineWrapping: false,
     readOnly: 'nocursor'
};
```

CodeMirror oferă posibilitatea de a interzice utilizatorului să aducă modificări de scriere în editor. În această secţiune el poate doar să vizualizeze rezultatele oferite. Am eliminat şi orice tip de evidenţiere a sintaxei, mod sau numerotare a rândurilor. Deşi această secţiune nu avea nevoie de o librărie specială, am decis să o refolosesc pentru a păstra o consistenţă.

6. Adăugarea unui nou instrument

Pentru a implementa funcţionalitatea unui nou instrument, se pot urmări paşii prezentaţi mai jos. Modificările ce trebuie aduse la nivel de frontend sunt următoarele:

- În fişierul ”dashboard-service.js” trebuie actualizată variabila `this.tools` astfel încât să conţină numele şi exemplele pentru noul instrument. Tot aici, trebuie actualizată şi metoda de citire a fişierului utilizat la nivel de backend pentru stocarea datelor de ieşire;
- În fişierul ”config.js” trebuie adăugată noua cale cu numele URL-ului folosit la pasul precedent;
- Opţional, se poate defini un mod pentru sintaxă, creat în directorul ”editor-container”, cu numele definit mai sus. Acest fişier trebuie importat în ”editor-container.js”;

La nivel de backend, trebuie implementate pornirea, respectiv întreruperea execuţiilor pentru noul instrument, în funcţie de sistemul de operare. Modificările trebuie aduse doar în fişierul ”tool.php”. La nivel de server, se poate defini un container pentru acest instrument şi se poate folosi noua comandă ce rulează cu `docker`.

Capitolul IV

Securitate

Iniţial, aplicaţia a fost implementată pe un server local, Apache, ce rulează pe sistemul de operare Windows. Pe urmă, aceasta a fost publicată şi pe un server de Linux, pentru a putea fi accesată de pe orice dispozitiv, indiferent de sistemul de operare pe care îl foloseşte utilizatorul.

Dacă instrumentele conţin vulnerabilităţi, utilizatorul le poate exploata deoarece are control asupra datelor de intrare. În acest caz, pentru ca cele două executabile, RMT şi Z3 să ruleze într-un mediu sigur, am folosit containerele Docker. Acestea asigură ca eventualele exploatări să nu blocheze sau să afecteze întregul sistem.

Docker este o platformă folosită pentru a încapsula, distribui şi gestiona aplicaţii Linux în containere izolate. O imagine este un executabil care include tot ce are nevoie aplicaţia pentru a rula. Aceasta se creează printr-un fişier Dockerfile, ce conţine o serie de comenzi, executate atunci când se construieşte imaginea. Un container este o instanţă a unei imagini, şi se obţine de fiecare dată când imaginea este rulată.

În aplicaţie, am folosit câte o imagine pentru fiecare executabil RMT, respectiv Z3. Ca exemplu, fişierul Dockerfile pentru RMT a fost definit astfel:

```
FROM ubuntu
COPY rmt /
RUN useradd -ms /bin/bash rmt
USER rmt
ENTRYPOINT ["/rmt"]
```

`Ubuntu` este imaginea de bază pentru a se construi noua imagine, cu numele `rmt`. Se va copia executabilul rmt în directorul curent, după care se creează un nou utilizator cu acelaşi nume. Utilizatorul non-root este folosit pentru execuţia imaginii şi pentru opţiunea `"entrypoint"` care permite configurarea containerului sa ruleze ca un executabil. Prin

comanda ”docker build -t rmt:latest .” se construieşte imaginea cu numele rmt, în directorul curent. În PHP, am folosit următoarea comandă docker:

```
”docker run -i --rm rmt < examples/new-rmt-example.rmt > pending-rmt-results.txt 2>&1”
```

Aceasta creează un nou container ce va rula RMT, având la intrare fişierul cu programul ce se doreşte a fi executat. Rezultatul standard întreg, de ieşire sau eroare, va fi redirecţionat în alt fişier, pentru a putea fi citit pe partea de frontend şi afişat utilizatorului chiar şi în timpul execuţiei. La final, containerul va fi şters automat datorită flag-ului ”--rm”.

Pentru rularea imaginii ce conţine executabilul Z3, am folosit următoarea comandă:

```
”docker run -i --rm z3 --in < examples/new-z3-example.z3 > pending-z3-results.txt”
```

Asemănător cu funcţionalitatea pentru RMT, şi aici am folosit un nou fişier de intrare cu numele ”new-z3-example.z3”, fiind populat în momentul execuţiei cu datele scrise de utilizator în editorul de text.

Containerele Docker folosite oferă astfel securitate prin faptul că procesele sunt rulate într-un container izolat, de către un utilizator care nu are privilegiile pe care le are un utilizator root (administrativ).

Prin publicarea proiectului, acesta poate fi accesat de oricine, indiferent de sistemul de operare folosit, iar execuţia programelor cu ajutorul unor comenzi, la nivel de server, devine mai sigură datorită măsurilor de securitate ce implică folosirea containerelor.

Concluzii

Prin această aplicaţie am reuşit să ofer utilizatorilor un mediu de lucru ce poate fi accesat oriunde, fără a fi nevoie de configurările şi instalările instrumentelor existente. Am oferit posibilitatea de a-şi implementa propriile comenzi, ce pot fi executate sau testate rapid, într-un editor de cod.

Încă de la început, am configurat proiectul astfel încât să poată fi instalat local cât mai simplu, pentru a putea fi dezvoltat în continuare, iar codul să fie reutilizat şi uşor de menţinut datorită structurii sale.

Astfel, funcţionalitatea poate fi extinsă la mai multe instrumente ce se execută în linia de comandă, sau se pot implementa noi idei. Se pot aduce îmbunătăţiri asupra încărcării exemplelor astfel încât acestea să poată fi citite automat în funcţie de structura ierarhică a fişierelor aflate pe server şi nu introduse prin actualizarea unui fişier PHP, ca în cazul de faţă. Atunci când utilizatorul îşi scrie comenzile pentru un instrument, acestea ar putea fi stocate într-o bază de date printr-o nouă opţiune de salvare sau modificare într-un alt exemplu. În plus, se poate crea şi o modalitate de exportare şi partajare a codului scris, între alţi utilizatori.

Toate aceste îmbunătăţiri se pot aduce pe viitor, împreună cu multe altele, pentru a încuraja cât mai mulţi utilizatori să folosească aplicaţia.

Bibliografie

1. Pro AngularJS, Adam Freeman, May 2014, paginile 476-480 , 561- 565;
2. AngularJS, Brad Green and Shyam Seshadri, April 2013, paginile 126-128, 132-133;
3. https://docs.angularjs.org/guide
4. http://codemirror.net/doc/manual.html
5. https://codemirror.net/demo/simplemode.html
6. http://php.net/manual/ro/function.shell-exec.php
7. https://developer.mozilla.org/en/docs/Web/JavaScript/Guide/Regular_Expressions