Facultatea Calculatoare, Informatica si Microelectronica

Universitatea Tehnica a Moldovei

Medii Interactive de Dezvoltare a Produselor Soft

Lucrarea de laborator#3

Web Development

Autor:
Eugen MIROVSCHI

lector asistent:
Irina Cojanu
lector superior:
Svetlana Cojocaru

Laboratory work #2

1 Scopul lucrarii de laborator

Studierea tehnologiilor de creare a paginilor web ca HTML, JS și CSS. Implementarea unui backend sub modelul unui REST API care va accesa o bază de date. Folosirea request-urilor de tipa AJAX pentru a crea o aplicație de tip single page.

2 Obiective

a) Realizarea unui simplu Web Site personal

b) Familiarizarea cu HTML și CSS

c) Interacțiuni Javascript

3 Laboratory work implementation

3.1 Tasks and Points

a) Realizarea unui site

b) Site-ul trebuie să păstreze toată informația într-o baza de date

c) Site-ul trebuie să conțină AJAX Requests.

d) Implementarea XHR sau JSON responses. Careva din informație trebuie să fie dinamic încărcată pe pagină.

3.2 Analiza lucrarii de laborator

a) Primul pas a fost inițializarea unui nou repozitoriu pe GitHub și clonarea acestuia pe calculatorul personal: `https://github.com/emirovschi/MIDPS\-3`.

b) Proiectul a fost creat utilizând IntelliJ[1] și folosind Maven[2] ca build manager. Acesta oferă posibilitatea de a adăuga automat toate dependențele inclusiv cele tranzitive.

c) Backend-ul este bazat pe Spring framework. Acesta oferă un sistem de dependency injection[3] care oferă posibilitatea de a crea ușor componente și de a suplini dependențele acestora. O altă parte necesară din framework este integrarea pattern-ului MVC[4] care permite crearea unui REST API.

```
@Component("postConverter")
public class PostConverter implements Converter<PostModel, PostDTO>
{
    @Resource
    private Populator<PostModel, PostDTO> postMinimalPopulator;

    @Resource
    private Populator<PostModel, PostDTO> postVotesPopulator;

    @Override
    public PostDTO convert(final PostModel post)
    {
        final PostDTO postDTO = new PostDTO();
        postMinimalPopulator.populate(post, postDTO);
        postVotesPopulator.populate(post, postDTO);
        return postDTO;
    }
}
```

Figure 3.1 – Crearea componentelor și injectarea dependențelor folosind Spring framework

```java
@RestController
@RequestMapping("/tags")
public class TagController
{
    @Autowired
    private TagFacade tagFacade;

    @RequestMapping(value = "/search", method = RequestMethod.POST)
    public ListDTO<TagDTO> searchTags(@RequestBody final SearchDTO search) { return tagFacade.sea
}
```

Figure 3.2– Definirea unui REST endpoint

d) Spring oferă posibilitatea de a defini entități folosind JPA framework[5] și repozitorii care vor gestiona aceste entități. Astfel acest framework permite implementarea proiectului fără a depinde de o anumită bază de date. În acest caz am utilizat local pentru dezvoltare și testare o baza de date de tip H2 însă pe server se folosește PostgresSQL fără a efectua careva schimbări majore cu excepția configurărilor.

```java
@Entity
@Table(name = "tags")
public class TagModel
{
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "tags_id_seq")
    @GenericGenerator(name = "tags_id_seq", strategy = "org.hibernate.id.enhanced.SequenceStyleGe
    private long id;

    @Column(unique = true)
    private String name;

    public String getName() { return name; }

    public void setName(final String name) { this.name = name; }
}
```

Figure 3.3– Definirea unei entități

```java
@Repository
public interface TagRepository extends JpaRepository<TagModel, Long>
{
    TagModel findByName(String name);

    @Query(SEARCH_TAGS)
    List<TagModel> findTags(@Param("query") String query, @Param("addsCount") long addsCount,
                            @Param("adds") Set<TagModel> adds, @Param("excludes") Set<TagModel> e
}
```

Figure 3.4– Definirea unui repozitoriu

e) O altă funcționalitate importantă oferită de acest framework este Spring Boot[6] care include în sine HTTP server și elimină dependeța de careva aplicație gen Tomcat.

f) Partea front-end a fost implementată utilizând framework-ul AngularJS[7]. Aceta oferă posibilitatea de a crea un single page application utilizând MVW pattern.

```
app.controller("login", function($scope, $mdDialog, auth)
{
    $scope.request = auth.request;

    $scope.isLoading = function()
    {
        return auth.isLoading();
    }

    $scope.close = function()
    {
        $mdDialog.hide();
    }

    $scope.login = function()
    {
        auth.login(function()
        {
            $scope.close();
        },
        function(data)
        {
            $scope.loginForm["username"].$setValidity("bad", false);
            $scope.loginForm["password"].$setValidity("bad", false);
        });
    }
});
```

Figure 3.5 – Exemplu de controller

```
<md-dialog flex="90" flex-gt-xs="75" flex-gt-sm="50" flex-gt-md="30">
    <form name="loginForm" role="form" ng-submit="login()" ng-cloak>
        <md-toolbar class="md-primary md-hue-2" md-theme="login">
            <div class="md-toolbar-tools">
                <h2>Log In</h2>
                <span flex></span>
                <md-button class="md-icon-button" ng-click="close()">
                    <md-icon><i class="material-icons">close</i></md-icon>
                </md-button>
            </div>
        </md-toolbar>

        <md-dialog-content>
            <div class="md-dialog-content" layout="column" layout-align="start stretch">
                <md-input-container class="md-block">
                    <label>Email</label>
                    <input ng-model="request.username" type="text" name="username" ng-disabled="isLoading()" />
                    <div ng-messages="loginForm.password.$error">
                        <div ng-message="bad"></div>
                    </div>
                </md-input-container>
                <md-input-container class="md-block">
                    <label>Password</label>
                    <input ng-model="request.password" type="password" name="password" autocomplete="new-password" ng-disabled="isLoading()" />
                    <div ng-messages="loginForm.password.$error">
                        <div ng-message="bad">Incorrect username or password</div>
                    </div>
                </md-input-container>
            </div>
        </md-dialog-content>

        <md-dialog-actions>
            <md-button flex class="md-flex md-primary md-raised md-hue-2" ng-disabled="isLoading()" type="submit">Log In</md-button>
        </md-dialog-actions>
    </form>
    <md-progress-linear md-mode="intermediate" ng-show="isLoading()"></md-progress-linear>
</md-dialog>
```

Figure 3.6 – Exemplu de template

g) Pentru ușura definirea elementelor a fost inclusă extensia Angular Material[8] care oferă un set de componente web, instrucțiuni de definire a structurii pagini și a stilurilor acesteia inclusiv pentru diferite dispozitive.

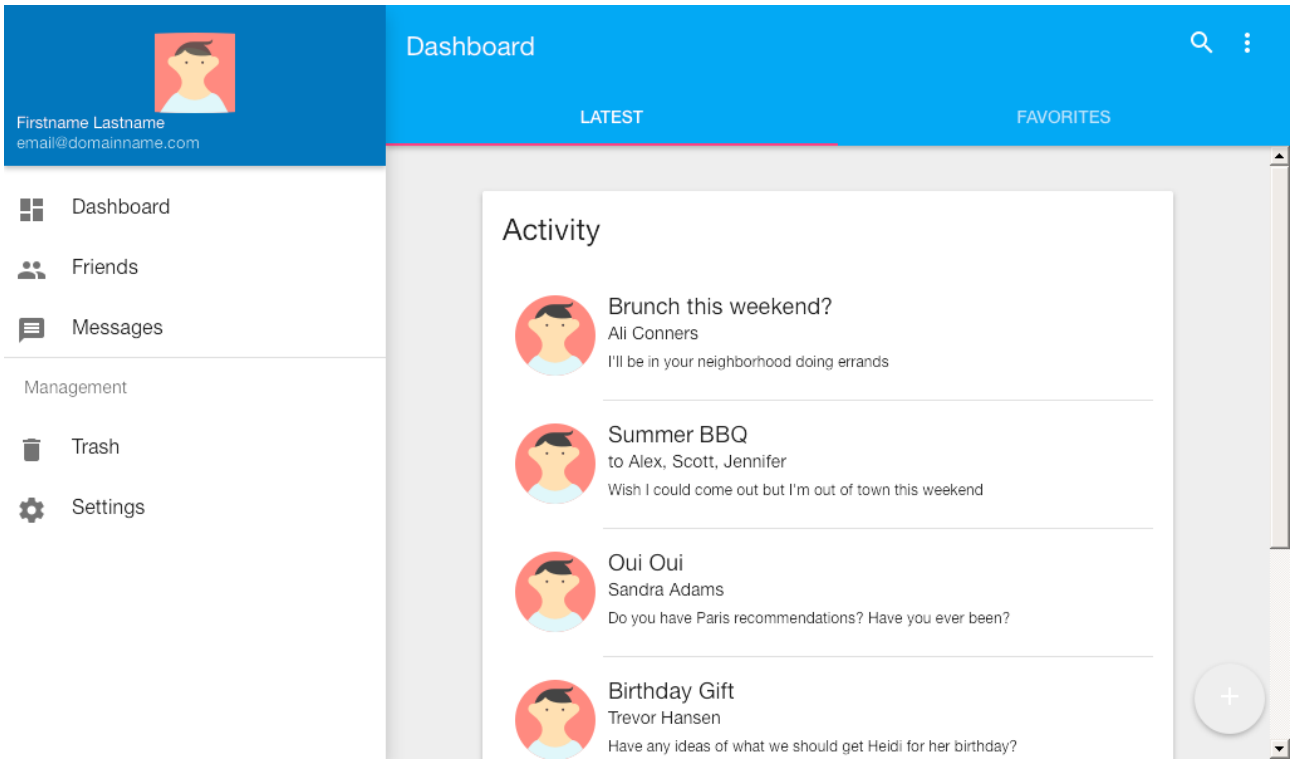

Figure 3.7 – Exemplu de pagină creată folosind Angular Material

3.3 Imagini

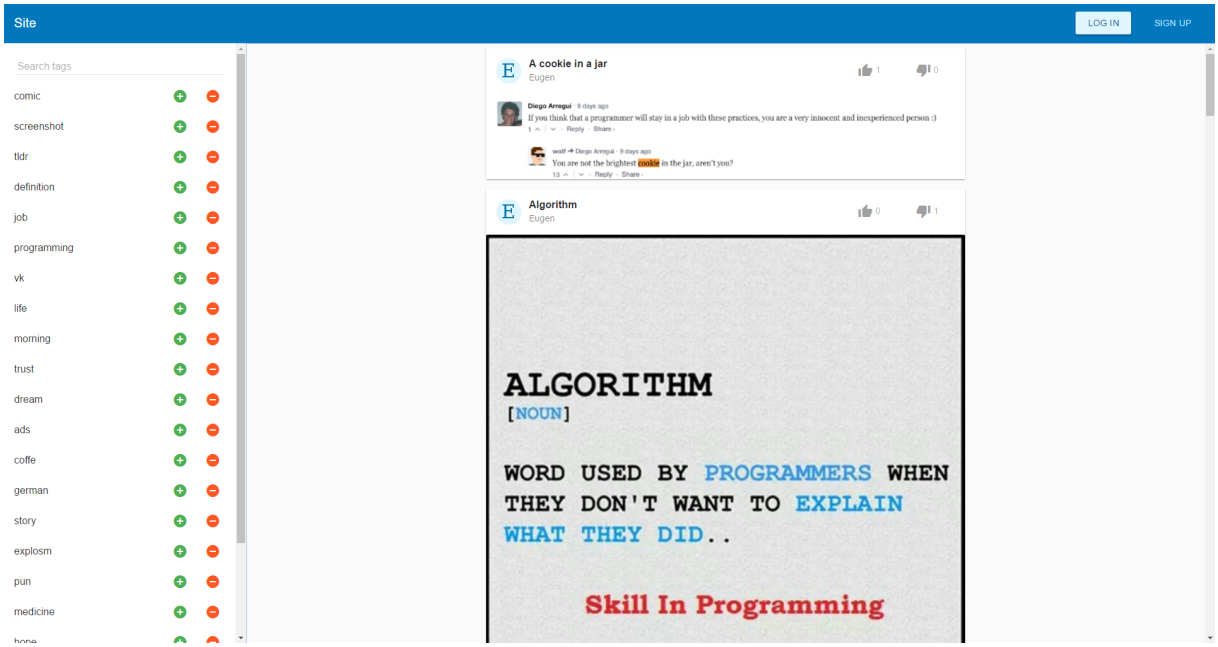

Figure 3.8 – Pagina principală pe Desktop

Figure 3.9 – Pagina principală pe Mobile

Figure 3.10 – Sidebar pe pagina principală de pe dispozitiv Mobile

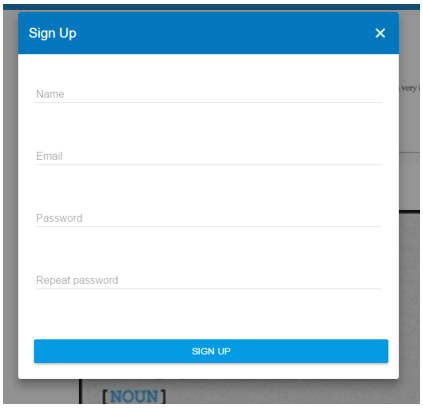

Figure 3.11 – Pagina de înregistrare

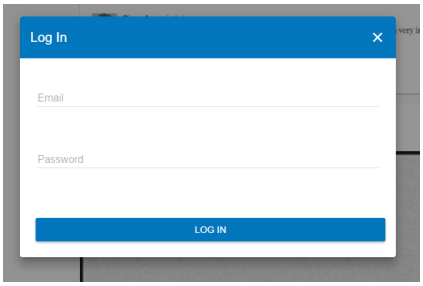

Figure 3.12 – Pagina de autentificare

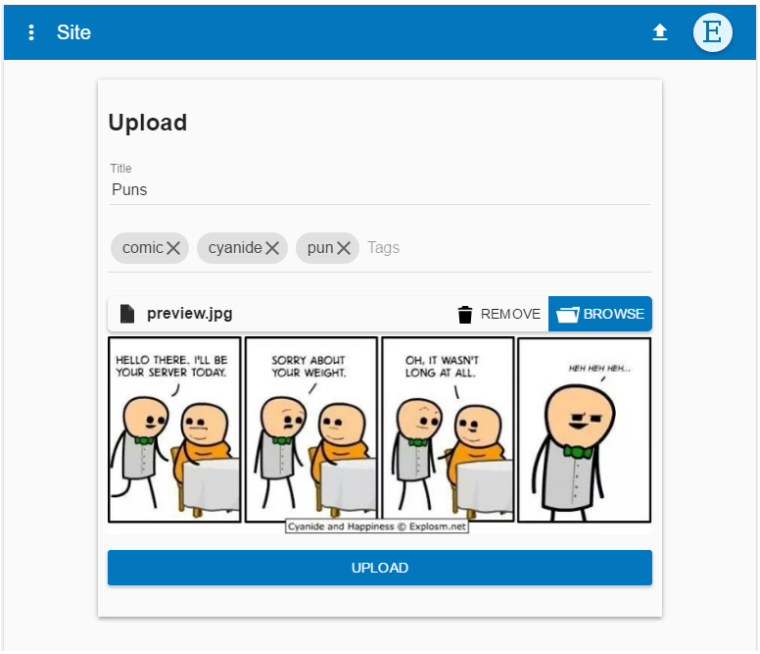

Figure 3.13– Încărcarea unei imagini

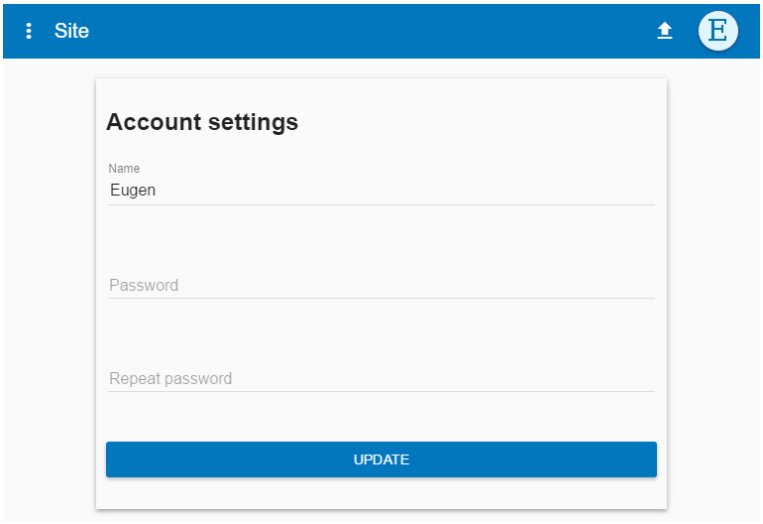

Figure 3.14– Modifcarea setărilor de cont

Concluzie

Pentru efectuarea acestei lucrări am utilizat un set de instrumente care m-au ajutat la implementarea unui site. Am folosit git ca sistem de control al versiunilor. Acesta oferă posibilitatea de a înregistra fiecare modificare ca o versiune separată precum și crearea mai multor branch-uri. Ca mediu de dezvoltare am folosit IntelliJ care permite redactarea optimă a codului Java și are integrarea cu git și Maven. Pentru a controla structura și modul de construire a proiectului, precum și includerea eficientă și rapidă a dependențelor externe am folosit managerul Maven. Acest site este compus din 2 părți principale: Backend și Frontend. Prima este scrisă în Java și conține toată logica care se executa pe partea de server side. Una din dependențele principale utilizate aici este Spring framework care ușureaza crearea componentelor injectând automat toate dependențele, permite crearea unui REST API folosind controllere si metode anotate, oferă posibilitatea de a integra o bază de date într-un mod generic care ulterior permite să aplici orice tip de bază de date relațională. Pentru partea de frontend am folosit 2 framework-uri: AngularJS si Angular Material care conlucrează între ele. Primul oferă posibilitatea de a crea un single page application folosind MVW pattern. Angular Material oferă un set de utilități care ușurează procesul de a crea toate elementele din pagină și permite reutilizarea acestora cu scopul de a defini strucutra paginii pentru mai multe dispozitive. Toate aceste instrumente mi-au permis sa dezvolt eficient un produs software calitativ.

References

1 JetBrains IntelliJ, *offical page*, https://www.jetbrains.com/idea/

2 Apache Maven, *offical page*, https://maven.apache.org/

3 Spring IoC, *documentation*, https://docs.spring.io/spring/docs/current/spring-framework-reference/html/beans.html

4 Spring MVC, *documentation*, https://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html

5 Spring Data JPA , *documentation*, https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

6 Spring Boot, *documentation*, https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/

7 AngularJS, *official guide*, https://docs.angularjs.org/guide

8 Angular Material, *official page*, https://material.angularjs.org/latest/