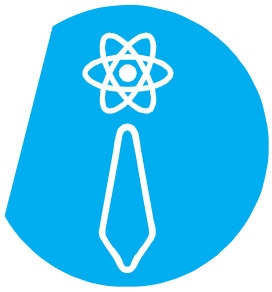

Staffr

Final Report Document

for Project Staffr

Prepared by
Kryštof Sýkora, Marek Szeles
ČVUT FEL SIT,
Enterprise architectures

Version 1.0
7. 1. 2018

1. Summary

1.1 Introduction

With Staffr, we aimed to produce a Java EE based Maven compilable program that allows for staff administration to a company representative with appropriate rights.

In effect, most of the program is heavily inspired by the provided “reporting tool” and “ear-setup” repositories, with other functions added to that base according to our assignment.

1.2 Development report

The main takeaway for us from developing an Enterprise application is that it takes a LONG time.

During development, the greatest hurdle was the nature of EAR application deployment – even though we started several months ahead of schedule and clocked in more than 300 hours, we still had difficulties to fulfil the basic expectations of a semestral project output. The main cause was that the greatest problem was not to create and implement the project, but instead most of the time we struggled with Java EE or React syntax and we spent more than half the time on the project debugging the individual technologies, rather than building and implementing our own solution.

1.3 Failed technologies

1.3.1 Javascript emulation

To save time, we have tried to use the DotVVM (https://www.dotvvm.com/) project as a framework to generate javascript, however implementations of this have failed to compile and be compatible with the main project, and so we abandoned the idea.

2. Used Technologies

For most of the program, we were inspired by the “reporting tool” concept, so we built our program on a similar base wireframe.

2.1 Backend

2.1.1 Functionality

The implemented functionality is based on the Staffr Software Requirements Specification version 1.2 from November 26th, 2017.

2.1.2 Persistence layer

The persistence layer is based on the functionality described in the Staffr Software Requirements Specification.

2.1.2.1 Cascade persist

The main cascade persist used is implemented when persisting a user, and it allows to persist all the skills associated with it. This is done using a prePersist implementation.

In AbstractRepositoryService.java:

```
@Transactional
@Override
public void persist(T instance) {
    Objects.requireNonNull(instance);
    prePersist(instance);
    getPrimaryDao().persist(instance);
}
```

And in the UserService.java:

```
@Override
void prePersist(User instance) {
    persistSkills(instance);
}

private void persistSkills(User instance) {
    Optional.ofNullable(instance.getSkills())
            .ifPresent(
                    skills -> skills.stream().filter(t -> t.getUser() == null)
                            .forEach( t -> {
                                    t.setUser(instance);
                                }
                            )
            );
}
```

It is also done by JPA annotations, for example in User.java:

```
@OneToMany(mappedBy = "user", cascade = CascadeType.PERSIST)
@OrderBy("name")
private Set<Skill> skills;
```

This cascade is also tested in UserServiceTest.java:

```
@Test
public void persistCascadesForSkills() {
    final UserDao ud = new UserDao();
    final UserService us=new UserService(ud);

    final User user = new User("Marek","Szeles",1996,"a@b.com","",3,
Status.ACTIVE);
    Set<Skill> skills=new HashSet<>();
    Skill skl_excel_3=new SoftSkill();
    skl_excel_3.setName("Excel");
    skl_excel_3.setProfficiency(SkillProfficiency.ADVANCED);
    skills.add(skl_excel_3);
    Skill skl_word_3=new SoftSkill();
    skl_word_3.setName("Word");
    skl_word_3.setProfficiency(SkillProfficiency.ADVANCED);
    skills.add(skl_word_3);

    user.setSkills(skills);
    us.persist(user);

    final User result = us.find(user.getId());
    Assert.assertEquals(skills.size(), result.getSkills().size());
}
```

2.1.2.2 Ordering

Ordering is used mainly when retrieving collections of users, in User.java Business Object:

```
@NamedQuery(name = "User.findAll", query = "SELECT l FROM User l ORDER BY
l.lastName DESC")
```

It is also paralelly done by JPA in

```
@OneToMany(mappedBy = "user", cascade = CascadeType.PERSIST)
@OrderBy("name")
private Set<Skill> skills;
```

This is also tested by UserDaoTest.java:

```
@Test
public void findAllReturnsUsersOrderedByNameDescending() {

    final User Peter_Smith = new User();
    Peter_Smith.setFirstName("Peter");
    Peter_Smith.setLastName("Smith");
    Peter_Smith.setEmail("P.SmithYo@Yahoo.com");

    final User Charlotte_Guido = new User();
    Charlotte_Guido.setFirstName("Charlotte");
    Charlotte_Guido.setLastName("Guido");
    Charlotte_Guido.setEmail("CG-see-gee@seznam.cz");

    final User Ivan_Terrible = new User();
    Ivan_Terrible.setFirstName("Ivan");
    Ivan_Terrible.setLastName("Terrible");
    Ivan_Terrible.setEmail("Impala@google.com");

    List<User> users = new LinkedList<User>();
    users.add(Ivan_Terrible);
    users.add(Charlotte_Guido);
    users.add(Peter_Smith);

    Collections.shuffle(users);
    dao.persist(users);

    final List<User> result = dao.findAll();
    Assert.assertEquals(users.size(), result.size());
    assertNameDescendingOrder(result);
}

private void assertNameDescendingOrder(List<User> users) {
    if (users.size() == 0) {
        return;
    }

    User previous = users.get(0);
    for (int i = 1; i < users.size(); i++) {
        final User current = users.get(i);
        assertTrue(current.getLastName().compareTo(previous.getLastName())<=0);
        previous = current;
    }
}
```

2.1.2.3 Named Querries

There are many named querries used in the DAO layer, for example, in the UserDao:

```
public User findByUsername(String username) {
    try {
        return em.createNamedQuery("User.findByUsername",
User.class).setParameter("username", username)
                .getSingleResult();
    } catch (NoResultException e) {
        return null;
    }
}
```

and in User.java Business Object

```
@NamedQueries({
        @NamedQuery(name = "User.findByName", query = "SELECT l FROM User l WHERE
LOWER(l.firstName) = :firstName AND LOWER(l.lastName) = :lastName"),
        @NamedQuery(name = "User.findByUsername", query = "SELECT p FROM User p WHERE
p.userName=:username"),
        @NamedQuery(name = "User.deleteById", query = "DELETE FROM User p WHERE
p.id=:id"),
        @NamedQuery(name = "User.findAll", query = "SELECT l FROM User l ORDER BY
l.lastName DESC")
})
```

2.1.3 CRUD layer

All parts of the CRUD layer are implemented, spread across the different roles:

- All users
  - Show user overview
    - C**R**UD read operation.
  - Edit own personal information
    - CR**U**D update operation.
- Project Leader only
  - Search for user according to criteria
    - C**R**UD read operation.
  - Create project pages
    - **C**RUD create operation.
  - Delete project pages
    - CRU**D** delete operation.
- Admin only
  - Register new user
    - **C**RUD create operation.
  - Remove user
    - CRU**D** delete operation.

2.1.4 Transactionality

The transactionality is used in the service layer, most notably in the AbstractRepositoryService.java masterclass, for example:

```
@Transactional(readOnly = true)
@Override
public List<T> findAll() {
    final List<T> result = getPrimaryDao().findAll();
    result.forEach(this::postLoad);
    return result;
}
```

2.1.5 Security

2.1.5.1 Authentication

Drawing inspiration from the Reporting Tool, authentication is handled through the authentication token architecture:

```
public AuthenticationToken(Collection<? extends GrantedAuthority> authorities, UserDetails userDetails) {
    super(authorities);
    this.userDetails = userDetails;
    super.setAuthenticated(true);
    super.setDetails(userDetails);
}
```

Naturally, all passwords are encrypted:

```
public void encodePassword(PasswordEncoder encoder) {
    if (password == null || password.isEmpty()) {
        throw new IllegalStateException("Cannot encode an empty password.");
    }
    this.password = encoder.encode(password);
}
```

2.1.5.2 Authorization

Authorization is mostly done on front-end and depends on user role, as defined by enum:

```
public enum Role {
    ADMIN_ROLE, USER_ROLE, ;
}
```

2.1.5.3 Bean access restriction

Using @PreAuthorize annotation, we restrict user profile edits only to self or admins:

```
@Override
@PreAuthorize("(#instance.userName==principal.username)or(principal.authorities.contains('ADMIN_ROLE'))")
public void update(User instance) {
    try {
        if (!exists(instance.getUserName())) {
            System.out.println("User doesn't exists");
        } else {
            if (instance.getPassword() != null) {
                instance.encodePassword(passwordEncoder);
            }
            super.update(instance);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

2.1.5.4 Functionality based on role

Different functionalities are available based on user role in database, or the relation the user has to a project, see chapters 2.1.3 and 2.1.5.2 for reference.

2.2 Frontend

2.2.1 ReactJS

The whole frontend is done in the ReactJS library, communicating with the backend using a REST interface.

As an example, we can look at a function in ProjectStore.js:

```
onGetAllProjects(){
    console.log("onGetAllProjects");
    axios.get("/rest/project").then((response) => {
        this.setState({
            projects: response.data
        });
    }).catch((error) => {
        console.log(error);
    })
}
```

3. Project outputs

The outputs are a Java EE Maven compilable program with pre-defined basic functionality, user manuals for it and multiple documents reporting on the development and purpose of the program.

3.1 Further documentation, source code

Further documentation can be
https://gitlab.fel.cvut.cz/B171_B6B33EAR/sykorkry

In case access is needed, please contact sykorkry@fel.cvut.cz, or szelemar@fel.cvut.cz.

4. Installation and deployment

4.1 Development Environment Setup

The following software needs to be installed on the system for development:

- JDK 8
- NodeJS v6 or later
- ReactJS
- Maven
- Apache Tomcat (or any other application server)

To start developing, first go to src\main\webapp and run npm install. This will download the necessary Node dependencies (they are used by the UI written in ReactJS). You can check that everything is working by running npm test.

4.2 Storage Setup

The application uses a standard relation database. It is preconfigured to a PostgreSQL server named “Staffr_db”, running at “localhost: 5432”, and credentials “ear” / “ear”.

4.3 Running the Application

To run the application locally, start JS compile watcher by running npm start from app/root/src/main/webapp. The watcher will recompile JS whenever a change is made to the UI code.

Running the application is simple, just build it with maven and deploy the artifact into you application server.

5. Prepared Sample User Walkthrough

5.1 Boot the application

5.2 Login

Credentials: “admin” / “heslo”

5.3 Show own user page

5.4 Edit personal information

5.5 Search for user according to criteria

5.6 Create project page

5.7 Delete project page