

23 stycznia 2017

Mateusz Burniak 218321
Artur Malarz 218355
Szymon Janora 218361

grupa: środa, 7³⁰-9⁰⁰

DOKUMENTACJA PROJEKTU

BOJAMO

Project Management Software

Prowadzący:
Mgr inż. Agnieszka Kaczmarek

Rok akademicki:
2016/2017

Spis treści

1	Założenia projektowe	3
1.1	Dane techniczne	3
1.2	Funkcjonalności	3
1.3	Schemat bazy danych	3
2	Opis implementacji	5
2.1	Struktura projektu	5
2.2	Przykłady implementacji	6
2.2.1	Implementacja routingu	6
2.2.2	Implementacja modelu	7
2.2.3	Implementacja formularza	8
2.2.4	Implementacja widoku administratora	8
2.2.5	Implementacja widoków aplikacji	9
2.2.6	Implementacja bazy danych	10
3	Funkcjonalność projektu	11
3.1	Screeny działającej aplikacji	11
4	Podsumowanie	19

1 Założenia projektowe

Celem projektu jest zaimplementowanie zgodnego z metodyką SCRUM oprogramowania do zarządzania projektami, w postaci aplikacji internetowej oraz projekt i implementacja bazy danych, na której omawiana aplikacja będzie operować.

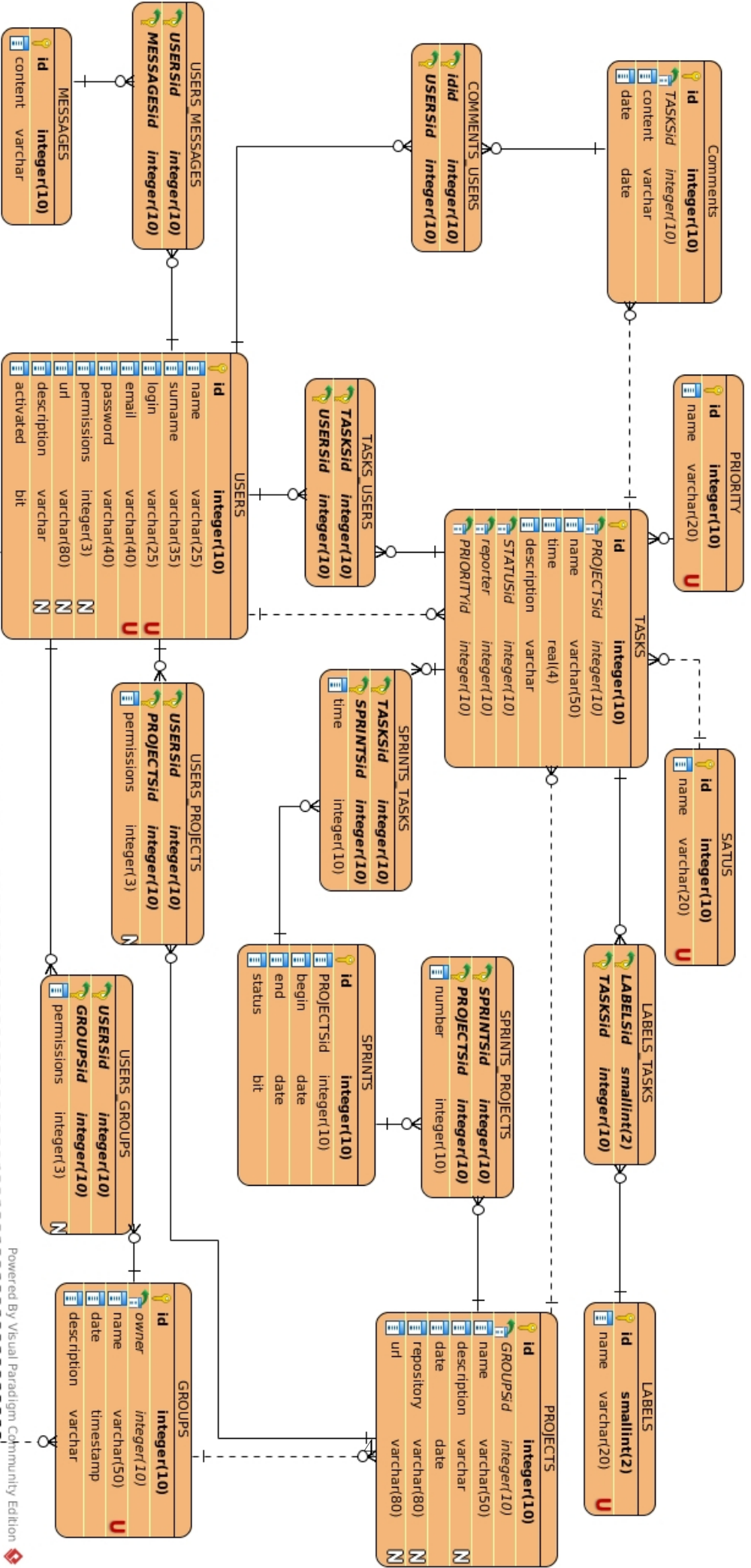
1.1 Dane techniczne

- Do implementacji BackEndu zostanie wykorzystany język programowania Python 3.51 oraz dostępny dla tego języka framework Django 1.10.
- Do zarządzania implementowaną bazą danych zostanie wykorzystany system PostgreSQL i SQLite (początkowa faza projektu).
- Do implementacji FrontEndu zostanie wykorzystany framework Twitter Bootstrap 3.
- W celu przeprowadzenia wirtualizacji środowiska, w którym pracować będzie system bazodanowy zostanie wykorzystane oprogramowanie Docker.
- W celu komunikacji i zarządzania danymi zawartymi w bazie danych zostanie wykorzystane mapowanie obiektowo-relacyjne, dzięki czemu aplikacja będzie kompatybilna z wieloma systemami bazodanowymi.

1.2 Funkcjonalności

- Aplikacja musi spełniać rolę menadżera projektów zgodnego z metodyką SCRUM, co za tym idzie musi dostarczać odpowiednie funkcjonalności i mechanizmy pozwalające na pracę zgodną z założeniami wybranej metodyki.
- Zgodnie z metodyką użytkownik będzie mógł wybierać odpowiadające mu zadania lub tworzyć nowe w obrębie aktywnego sprintu będącego elementem prac wykonywanych nad projektem w okresie czasu. Powinno być możliwe rozróżnienie zadań pod względem kategorii oraz wagi.
- Oprogramowanie musi pozwalać na śledzenie postępów prac wykonanych w projekcie, w związku z tym użytkownicy będą mieć możliwość logowania czasu spędzonego nad wykonywanym zadaniem.
- Oprogramowanie musi zawierać podstawowe zabezpieczenia w postaci walidacji użytkownika oraz jego dostępu do danych zasobów, aplikacja ma dostarczać podstawowe funkcje takie jak logowanie/rejestracja. Hasła muszą być odpowiednio zabezpieczone przed wyciekiem, w związku z tym zostanie wykorzystane szyfrowanie SHA2 z solą.

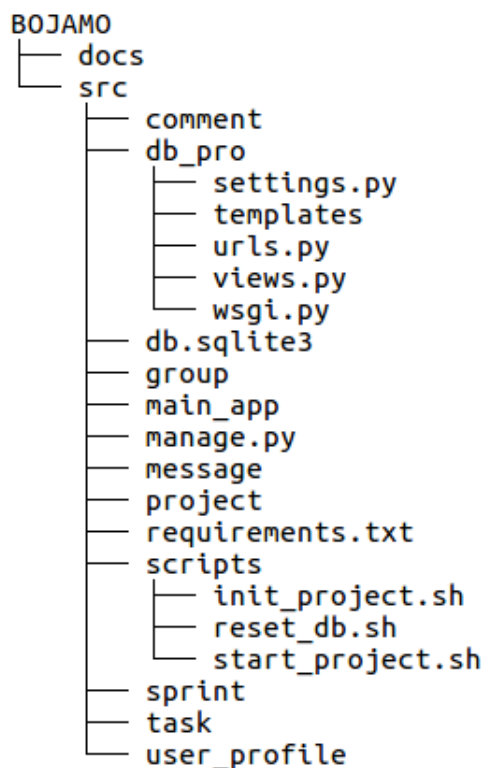
1.3 Schemat bazy danych



2 Opis implementacji

Realizowany przez nas projekt został zaimplementowany zgodnie ze wspieranym przez framework Django wzorcem MVC (Model View Controller), a ponadto aplikacje podzielono na pomniejsze serwisy według modeli bazy danych na których one operowały.

2.1 Struktura projektu



Rysunek 2: Schemat projektu

Aplikacja db-pro jest głównym serwisem projektu, odpowiada ona za podstawowy routing i działa jak główny kontroler wywołujący pozostałe serwisy.

Struktura folderu zawierającego serwis odpowiadający za operowanie na danych modelach, została przedstawiona na podstawie aplikacji group. Jak widać aplikacja group jak również pozostałe składa się z kilku plików oraz folderów migrations i templates. Folder migrations zawiera pliki wspierające ORM, wygenerowane przez Django i odpowiadające za migrację danych. Folder templates zawiera szablony stron wykorzystujące technologię HTML oraz Jinja, szablony odpowiadają za prezentację danych użytkownikowi. Jak można zauważyć w folderze aplikacji brakuje pliku controler, którym cechuje się wzorec MVC, brak kontrolera jest cechą frameworka Django, operacje obsługiwane przez kontroler zostały połączone z widokiem, dlatego dostępne mamy jedynie pliku models.py oraz views.py.

- models.py - odpowiada za utworzenie encji w bazie danych, odpowiadającej zapisanemu w języku Python obiektowi.

- `urls.py` - zawiera zapis routingu dla danego serwisu
- `views.py` - łączy w sobie funkcję widoku oraz kontrolera, służy do wyświetlania odpowiednich szablonów, operowaniu na danych oraz przekierowywaniu operacji do innych aplikacji.
- `admin.py` - pozwala zmodyfikować stronę administratora udostępnianą przez framework Django, wprowadzając widoki danej aplikacji i pozwalając na operowanie bazą danych w obrębie modeli dostępnych dla aplikacji.
- `forms.py` - przechowuje kod odpowiedzialny za utworzenie formularzy dla danego modelu oraz reguły walidacji jego poprawności.

W strukturze projektu można zauważyć plik bazy danych SQLite 3, jest to podstawowa baza danych, domyślnie tworzona przez framework. Omawiany system zarządzania bazą danych był wykorzystywany przez nas w początkowej fazie projektu, podczas dalszych prac i utworzeniu wszystkich potrzebnych nam modeli, zamieniliśmy omawiany system na PostgreSQL uruchamiany w wirtualnym kontenerze aplikacji Docker. Ze względu na zmianę systemu bazy danych został wprowadzony zestaw skryptów powłoki Bash, które dostępne są w folderze `scripts` i umożliwiają one inicjację danymi, uruchomienie, czy restart kontenera Docker z PostgreSQL. Obie wykorzystywane bazy danych zostały wypełnione wygenerowanymi przez nas danymi testowymi.

2.2 Przykłady implementacji

2.2.1 Implementacja routingu

Listing 1: `groups/urls.py`

```
from django.conf.urls import url

from project.views import ProjectCreateView
from .views import GroupCreateView, group_detail

urlpatterns = [
    url(r'^create/$', GroupCreateView.as_view(),
        name='group_create_view'),
    url(r'^(?P<name>\w+)/$', group_detail,
        name='group_detail'),
    url(r'^(?P<group>\w+)/add/$', ProjectCreateView.as_view(),
        name='project_create_view'),
]
```

Aplikacja pobiera każdy z adresów należących do jej domeny, w przypadku fazy rozwojowej jest to adres domeny `127.0.0.1:8000`. Odpytanie aplikacji adresem z parametrami powoduje próbę dopasowania adresu do zadanych wzorców w formie wyrażeń regularnych,

po dopasowaniu adresu zostaje wywołany przypisany do niego widok lub w przypadku głównego routingu, parametry z adresu zostają przekazane do przypisanej aplikacji.

Przykładowo, dopasowanie adresu 127.0.0.1:8000/group_name/add spowoduje wywołanie widoku ProjectCreateView, który umożliwi stworzenie nowego projektu przypisanego do grupy o nazwie group_name.

2.2.2 Implementacja modelu

Listing 2: groups/models.py

```
from django.contrib.auth.models import User
from django.db import models

class Group(models.Model):
    owner = models.ForeignKey(
        User
    )
    name = models.CharField(
        max_length=50
    )
    date = models.DateField(
        auto_now=True
    )
    description = models.TextField(
        blank=True,
        null=True
    )

    def __str__(self):
        return self.name

class UserGroup(models.Model):
    user = models.ForeignKey(User)
    group = models.ForeignKey(Group)
    permissions = models.PositiveSmallIntegerField(default=1)

    def __str__(self):
        return '({{ user.username }}, {{ group }})'.format(self.user.username, self.group)

class Meta:
    unique_together = ['user', 'group']
```

Powyższy widok przedstawia implementację modeli obiektów Group oraz UserGroup.

System ORM wspierany przez Django, pozwala na proste tworzenie obiektów bazy danych, których implementacja przechowywana jest w kodzie języka Python. Ponadto system ten pozwala na przeładowanie metod odziedziczonych po klasie bazowej Model, co można założyć przykładowo po przeładowanej metodzie `__str__`, zwracającej tekstową reprezentację obiektu. Poza tym framework Django umożliwia proste tworzenie relacji pomiędzy modelami/encjami, co jest widoczne w implementacji modelu UserGroup, stanowiącego relację pomiędzy modelem User i modelem Group.

2.2.3 Implementacja formularza

Listing 3: groups/forms.py

```
from django import forms
from .models import Group

class GroupForm(forms.ModelForm):
    class Meta:
        model = Group
        fields = ('name', 'description')
        widgets = {
            'name': forms.TextInput(
                attrs={
                    'class': 'form-control',
                }
            ),
            'description': forms.Textarea(
                attrs={
                    'class': 'form-control'
                }
            )
        }
```

Framework Django pozwala na prostą obiektową implementację formularzy, co przedstawia powyższy widok, na którym widoczna jest implementacja formularza służącego do utworzenia obiektu modelu Group. Django pozwala ponadto na przeładowanie metod odziedziczonych po klasie bazowej ModelForm, co pozwoli chociażby na utworzenie metody służącej do walidacji poprawności wypełnionego formularza.

2.2.4 Implementacja widoku administratora

Listing 4: groups/admin.py

```
from django.contrib import admin

from .models import Group, UserGroup
```



```
admin.site.register(Group)
admin.site.register(UserGroup)
```

Poprzez wywołanie metody `register(model_type)` dostępnej dla podstrony administratora, z podanym w parametrze modelem pozwala na dodanie funkcjonalności w widoku administratora, pozwalających na operowanie bazą danych w obrębie zadanego modelu.

2.2.5 Implementacja widoków aplikacji

Listing 5: groups/views.py

```
from django.contrib.auth.decorators import login_required
from django.contrib.auth.models import User
from django.http import HttpResponseRedirect
from django.shortcuts import get_object_or_404, render
from django.urls import reverse_lazy
from django.utils.decorators import method_decorator
from django.views.generic import CreateView, ListView

from project.models import Project
from .forms import GroupForm
from .models import Group, UserGroup

class GroupCreateView(CreateView):
    model = Group
    form_class = GroupForm
    template_name = 'group/create-view.html'
    success_url = 'user-profile/loggedin.html'

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        return context

    def form_valid(self, form):
        group = Group.objects.create(owner=self.request.user,
                                     name=form['name'].value(),
                                     description=form['description'].value())
        UserGroup.objects.create(user=self.request.user, group=group)
        return HttpResponseRedirect(reverse_lazy('dashboard'))

    @method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        return super(GroupCreateView, self).dispatch(*args, **kwargs)
```

```

class GroupListView(ListView):
    model = Group

@login_required
def group_detail(request, name):
    group = get_object_or_404(Group, name=name)
    owner = User.objects.get(id=group.owner_id)
    members = [i.user for i in UserGroup.objects.filter(group=group)]
    projects = Project.objects.filter(group=group)
    context = {'group': group, 'owner': owner, 'members': members,
               'projects': projects}
    return render(request, 'group/group_detail.html', context)

```

Widoki zostały zaimplementowane na dwa sposoby, podstawowej metody pobierającej obiekt request oraz w sposób obiektowy przy wykorzystaniu widoków generycznych. Wszystkie widoki zostały zabezpieczone fixturą login_required, która uniemożliwia wywołanie danego widoku użytkownikowi, który nie przeszedł przez proces uwierzytelniania podczas logowania.

2.2.6 Implementacja bazy danych

Listing 6: scripts/reset_db.sh

```

#!/usr/bin/env bash

docker rm -f bojamo-postgres
docker run \
    --name bojamo-postgres \
    -e POSTGRES_PASSWORD=mysecretpassword \
    -e POSTGRES_USER=user \
    -e POSTGRES_DB=bojamo_db \
    -p 54321:5432 \
    -d postgres:9.5

sleep 10
python3 manage.py makemigrations
python3 manage.py migrate
./apply_mock.py

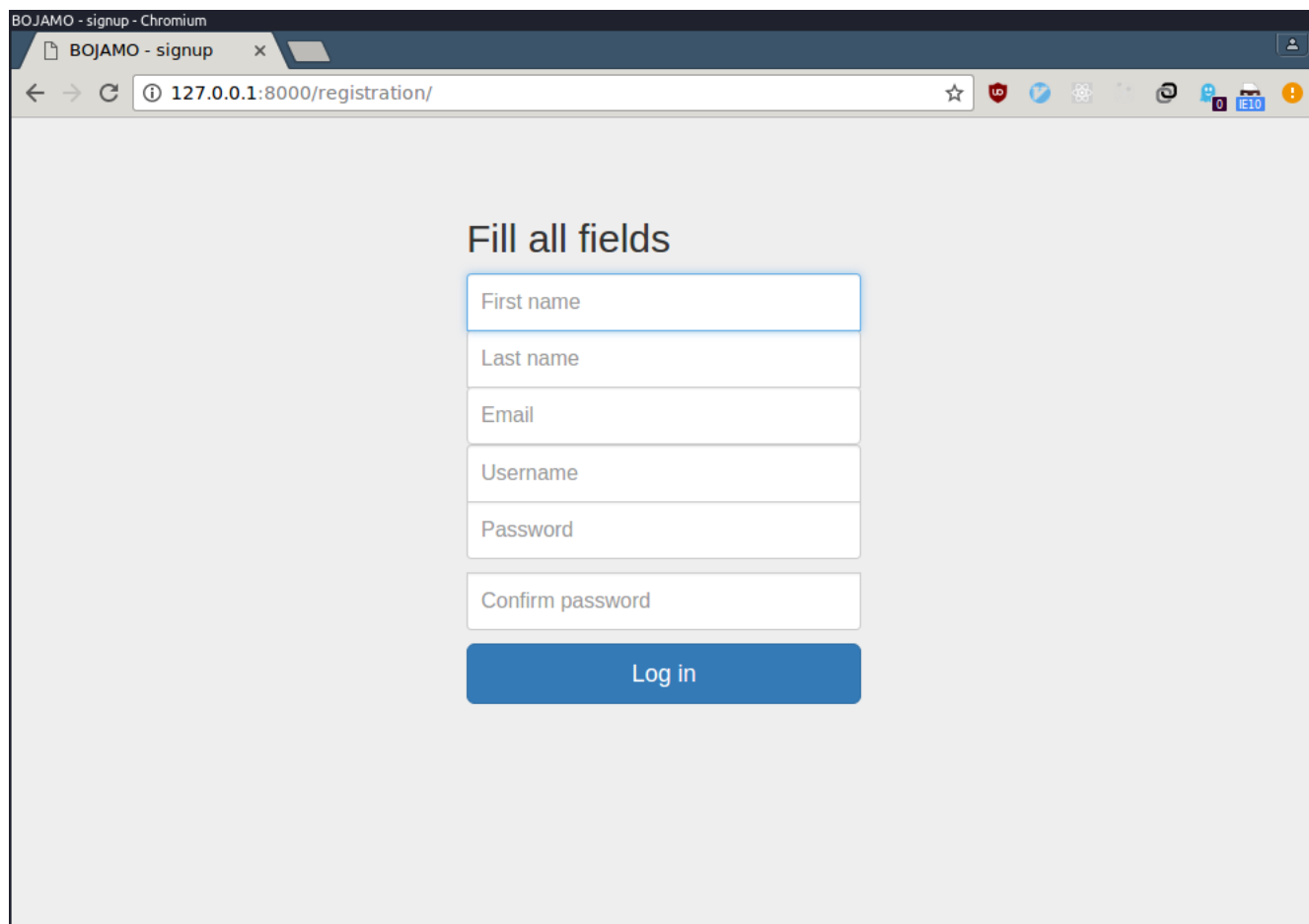
```

Jak widać na powyższym listingu najpierw usuwany jest poprzedni obraz dockerowy, następnie tworzony jest nowy i aplikowane są fixtury.

3 Funkcjonalność projektu

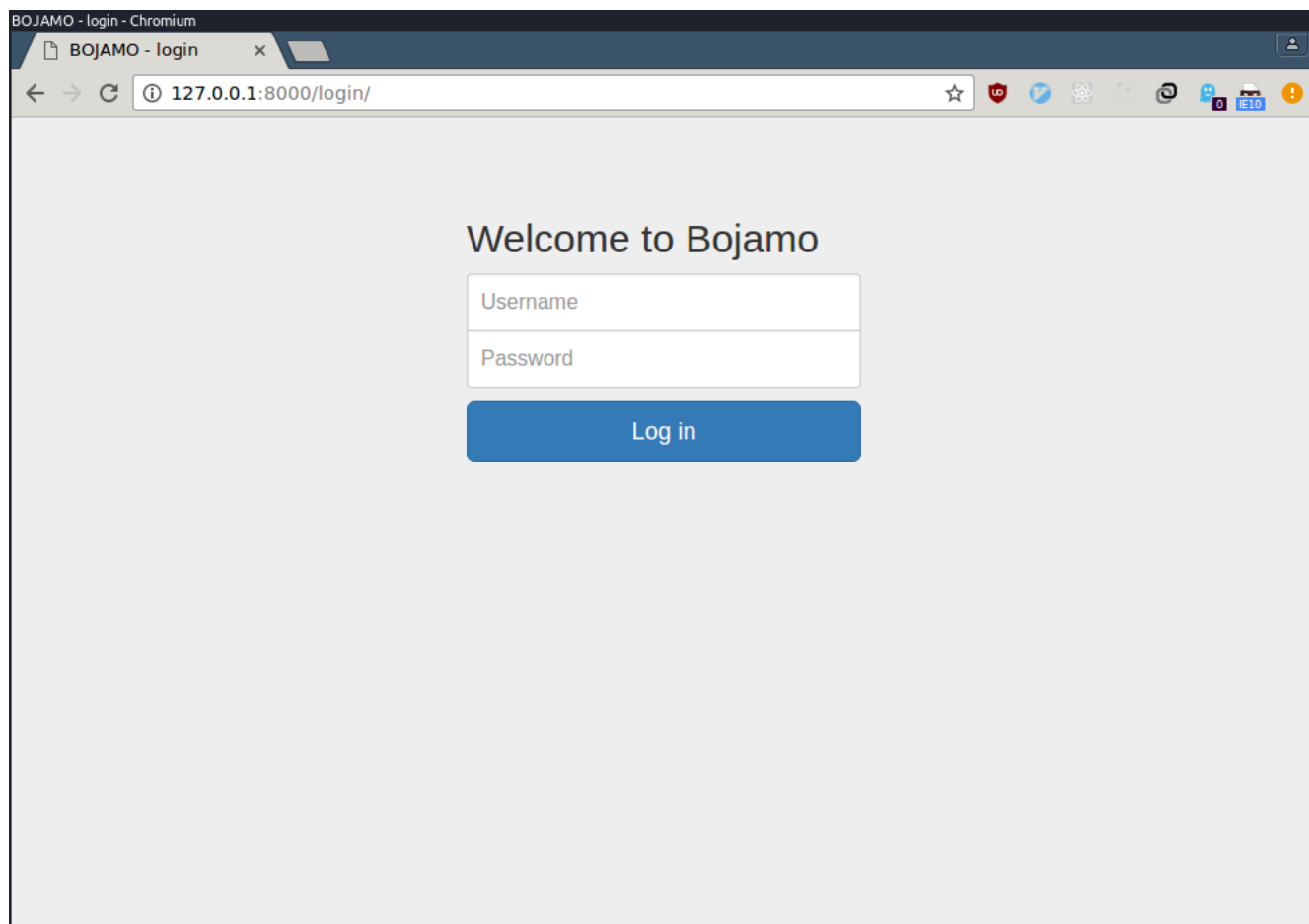
Niezałogowany użytkownik może się jedynie zalogować lub zarejestrować. Kiedy użytkownik jest już zalogowany, może dodawać nowe projekty, tworzyć grupy projektowe. Do grupy można dodawać nowe sprinty. Do sprintów można przypisywać taski.

3.1 Screeny działającej aplikacji

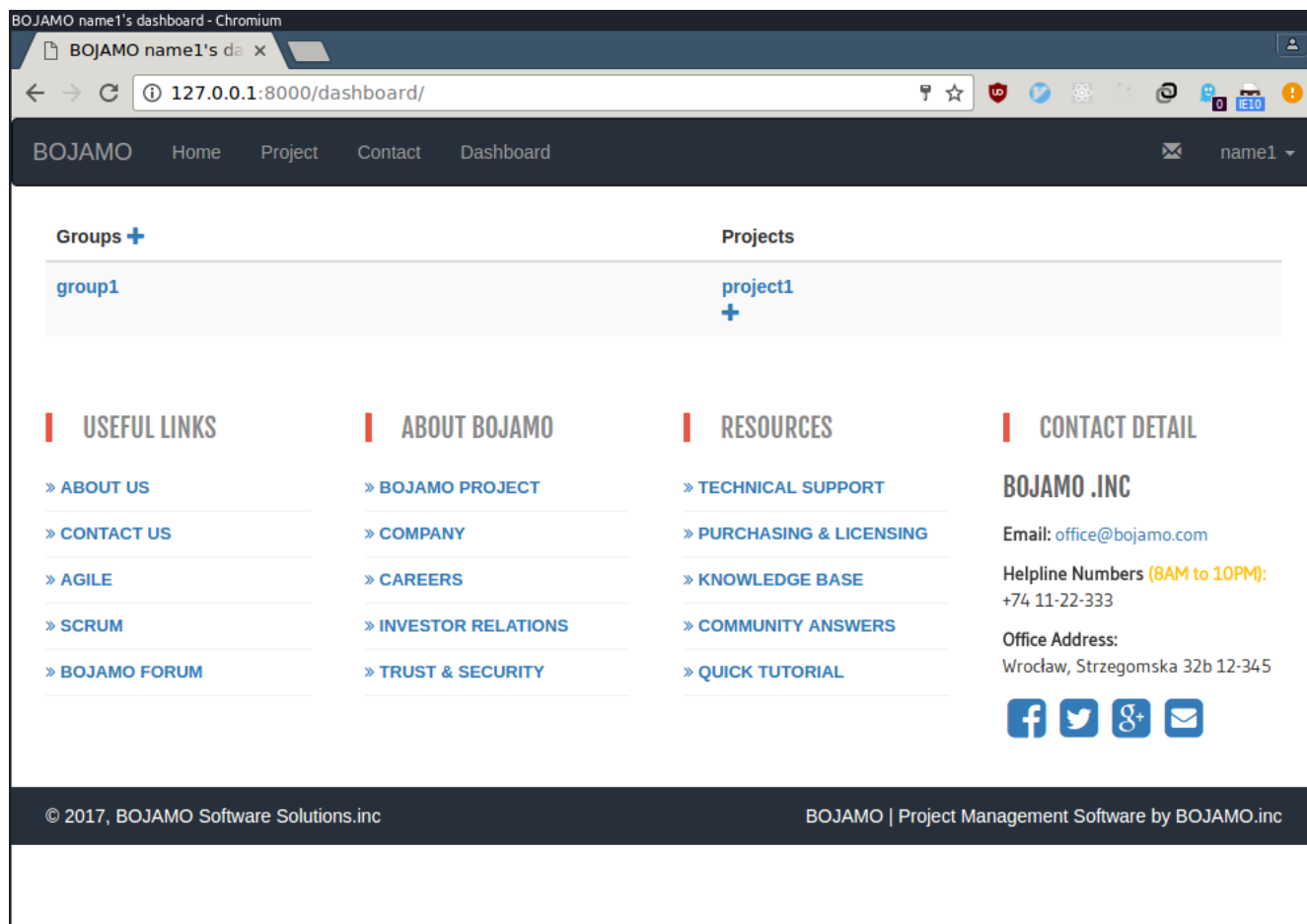


The screenshot shows a web browser window with the title 'BOJAMO - signup - Chromium'. The address bar displays '127.0.0.1:8000/registration/'. The main content area features a registration form with the heading 'Fill all fields'. The form consists of six input fields stacked vertically: 'First name', 'Last name', 'Email', 'Username', 'Password', and 'Confirm password'. Below these fields is a blue button labeled 'Log in'.

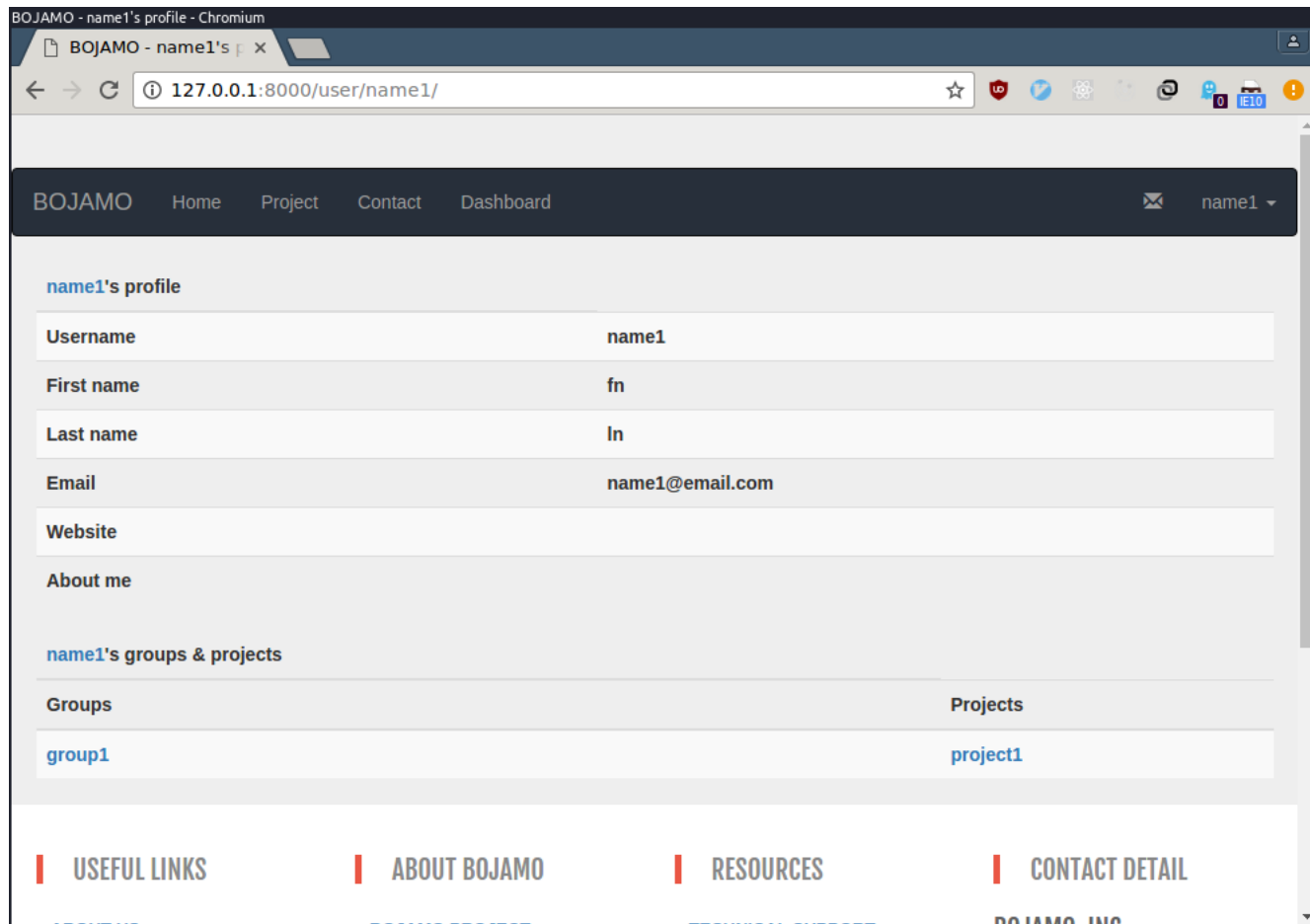
Rysunek 3: Rejestracja użytkownika



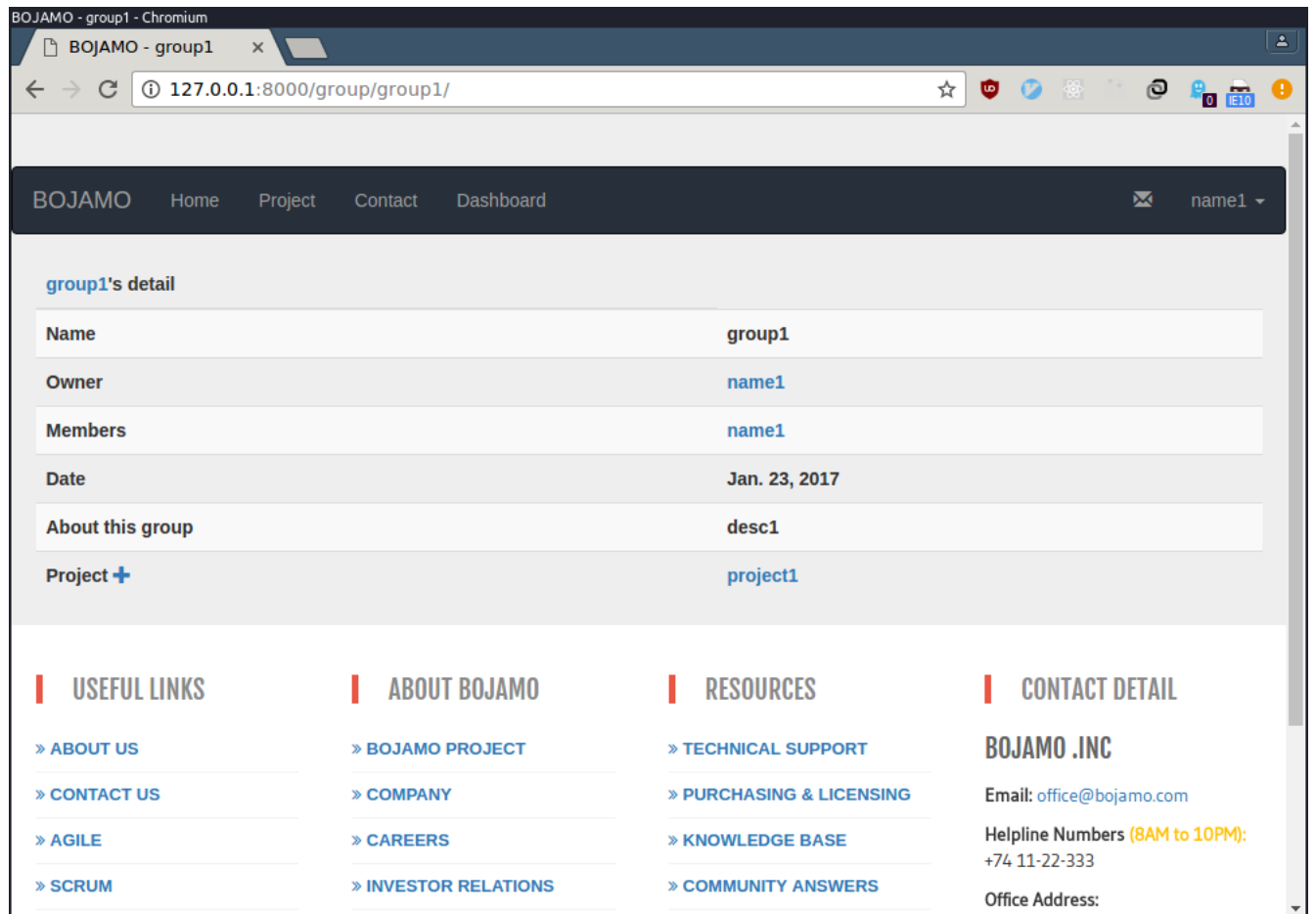
Rysunek 4: Logowanie użytkownika



Rysunek 5: Widok po zalogowaniu



Rysunek 6: Widok użytkownika



Rysunek 7: Widok grupy

BOJAMO - project1 - Chromium

BOJAMO - project1 x

127.0.0.1:8000/project/project1/

BOJAMO Home Project Contact Dashboard

name1

project1's detail

Name	project1
Owner	name1
Group	group1
Members	name1
Date	Jan. 23, 2017
Website	url1.com
Repository	repo1.com
About this project	desc pro 1

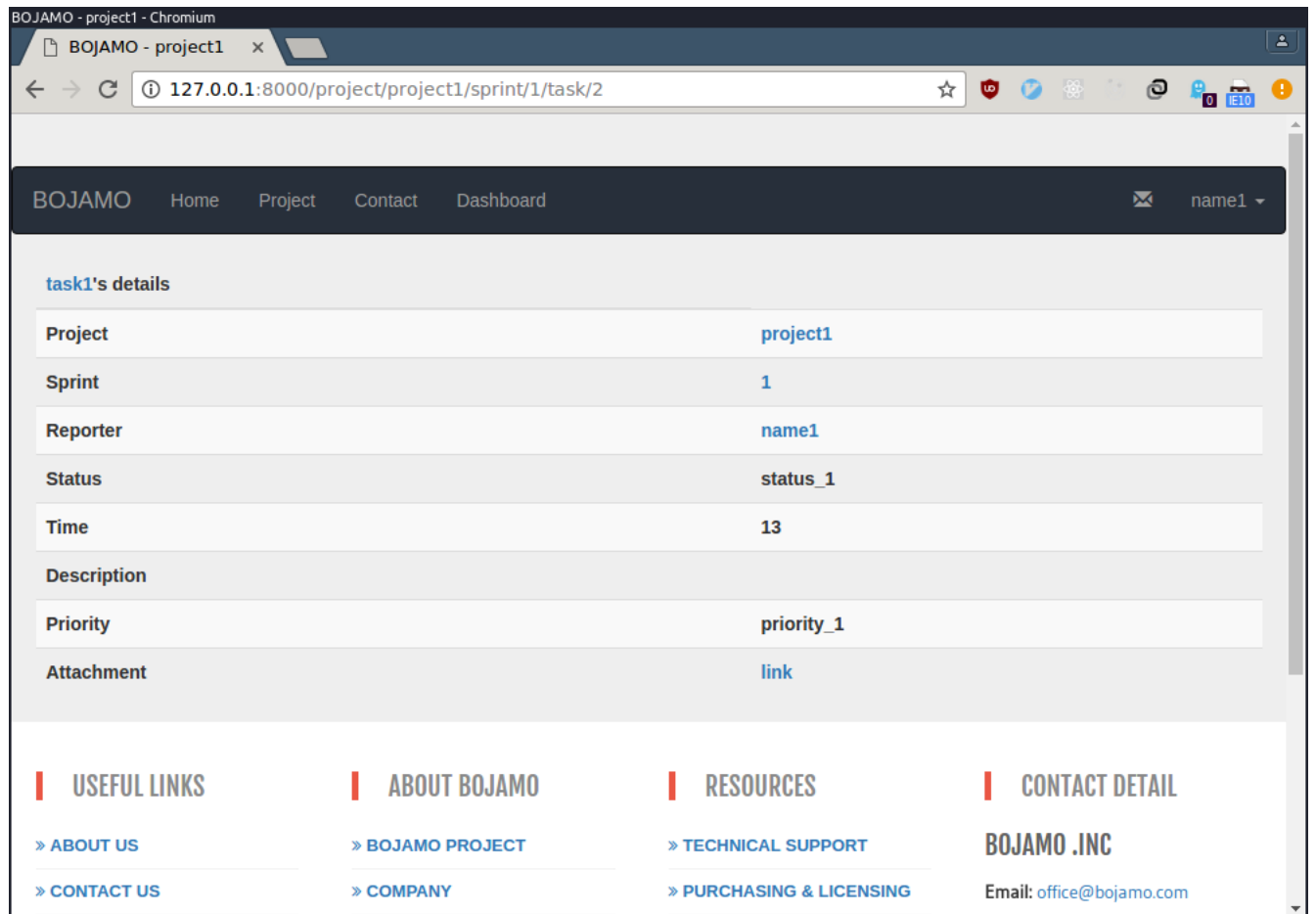
project1's sprints

Sprint +	Begin	End	Active
1	Jan. 23, 2017	Feb. 6, 2017	True

project1's active tasks

Task +	Time	Description
--------	------	-------------

Rysunek 8: Widok projektu



Rysunek 9: Widok zadania

4 Podsumowanie

Mając gotowe założenia stworzyliśmy kompletny projekt bazy danych oraz aplikacji. W fazie projektowania bazy danych poznaliśmy od podstaw etapy powstawania projektu bazy danych poczynając od diagramu związków encji, aż do powstania modelu fizycznego bazy danych. Podczas projektowania aplikacji poznaliśmy dokładniej Pythona i zasady działania frameworku Django, co pozwoliło nam szybciej stworzyć aplikację.

Jakość kodu jest na wysokim poziomie, zatem możliwy jest dalszy rozwój aplikacji. Korzystamy z najnowszych, obecnie dostępnych, technologii, które zyskują coraz większą popularność.

Mapowanie obiektowo-relacyjne jest to mechanizm znacznie ułatwiającym pracę z bazą danych. Dzięki niemu możemy komunikować się z bazą danych używając języka obiektowego zamiast wysyłania zapytań SQL. Ponadto nasza baza danych jest zwirtualizowana przy pomocy Dockera, co zapewnia większą przenośność i skalowalność.