

Wrocław, 31.05.2015r.

## **Aplikacje Internetowe i Rozproszone - Projekt**

### **Dokumentacja**

#### **Zespół:**

Tomasz Bagiński

Aleksandra Berjak

Piotr Chmiel

Jakub Stasiak

Maciej Stelmaszuk

Jędrzej Urbański

Sprawdzenie czy zadany zbiór zaszyfrowanych haseł może (i w jakim czasie) zostać złamany za pomocą komputera równoległego, jakim jest klaster stacji roboczych.

Rok akad. 2014/2015, kierunek: INF

#### **Prowadzący:**

Dr hab inż. Henryk

Maciejewski

## **Spis treści**

<b>1. Wstęp.....</b>	<b>3</b>
<b>2. Opis wykorzystywanych rozwiązań .....</b>	<b>3</b>
<b>2.1. Aplikacja kliencka - Python + Framework Django .....</b>	<b>3</b>
<b>2.1.1.Hierarchia szablonów .....</b>	<b>3</b>
<b>2.1.2.Opis widoków .....</b>	<b>4</b>
<b>2.1.3.Modele.....</b>	<b>13</b>
<b>2.2. Algorytmy łamania haseł .....</b>	<b>14</b>
<b>2.2.1.Metoda słownikowa (atak słownikowy) .....</b>	<b>14</b>
<b>2.2.2.Brute Force.....</b>	<b>18</b>
<b>2.2.3.Tablice tęczowe .....</b>	<b>19</b>
<b>2.3. Komunikacja między modułami .....</b>	<b>21</b>
<b>3. Konfiguracja i uruchomienie systemu .....</b>	<b>22</b>
<b>3.1. Środowisko pracy .....</b>	<b>22</b>
<b>3.2. Aplikacja wykonująca obliczenia (C++) .....</b>	<b>22</b>
<b>3.3. Aplikacja kliencka (Django).....</b>	<b>24</b>
<b>4. Testy wydajnościowe .....</b>	<b>25</b>

1.

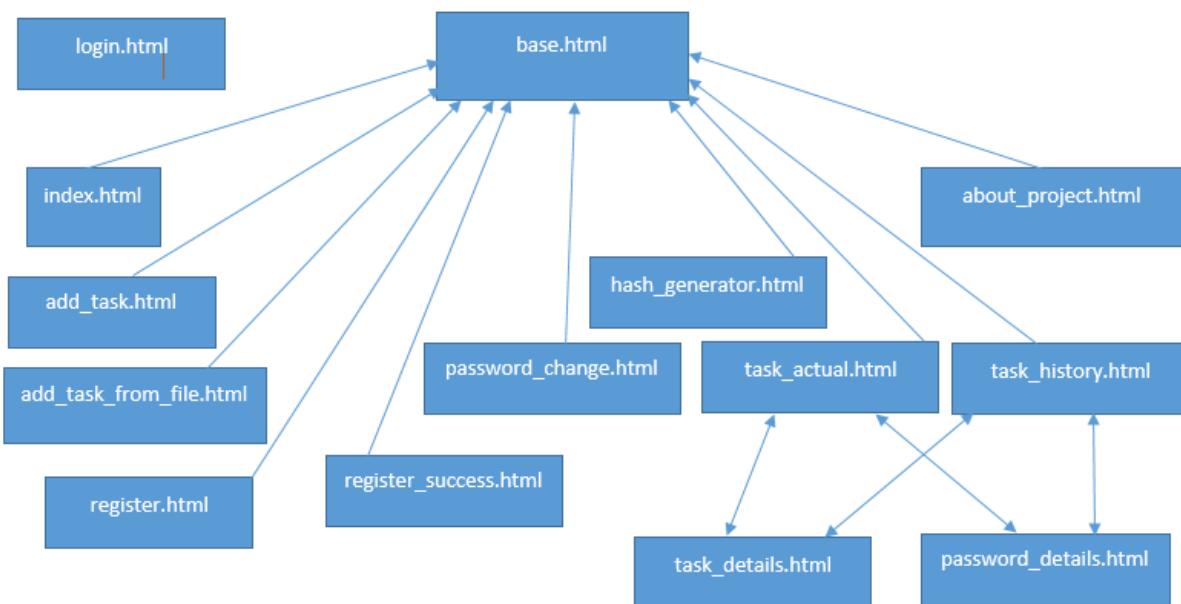
## 1. Wstęp

System stworzony do łamania haseł za pomocą komputera równoległego, jakim jest kластер stacji roboczych jest podzielony na dwa główne moduły - aplikację kliencką napisaną przy pomocy framework Django i moduł MPI wykonujący obliczenia przy pomocy klastra stacji roboczych. Aplikacja kliencka dostarcza użytkownikowi graficzny interfejs pozwalający na zlecanie zadań i monitorowanie ich wykonywania. Moduł MPI odpowiedzialny jest za rozdział zadań pomiędzy stacjami roboczymi i wykonywanie obliczeń.

## 2. Opis wykorzystywanych rozwiązań

### 2.1. Aplikacja kliencka - Python + Framework Django

#### 2.1.1. Hierarchia Szablonów



Rysunek 1 Schemat szablonów wykorzystywanych w aplikacji klienckiej

Głównym szablonem jest `base.html`, w nim znajduje się menu główne. Zawartość pod głównym paskiem nawigacyjnym wypełniają szablony dziedziczące po `base.html` widoczne na rysunku 1. Zagnieżdżenie szablonów utrzymuje się na jednym poziomie. Do każdej opcji możemy wejść z poziomu strony głównej. `Login.html` nie dziedziczy po `base.html`, zostaje on generowany w momencie pierwszego błędного logowania przez użytkownika. Szablony `task_details.html` oraz `password_details.html` generowane są, jako wypełnienia okna typu

pop-up ( klasa Dialog z jQuery UI) po kliknięciu przycisku szczegóły w widoku aktualne zadania lub historia zadań.

### **2.1.2. Opis widoków**

#### **2.1.2.1. Strona główna**

URL	/
Kontroler	views.py/home
Szablon	index.html
<b>Opis</b>	
Kontroler po otrzymaniu żądania od przeglądarki, metodą render generuje szablon zawarty w pliku index.html. Szablon wykorzystuje klasę bootstrap slider do stworzenia pokazu zdjęć. Niezarejestrowany użytkownik może się zalogować lub rozpocząć proces rejestracji klikając na przycisk rejestruj. Po zalogowaniu użytkownik na pasku głównym zobaczy menu mój profil oraz moje zadania. Użytkownik o uprawnieniach admina dodatkowo będzie miał dostęp do menu admin. Warunki sprawdzające status zalogowania użytkownika oraz jego uprawnienia znajdują się w szablonie base.html	



#### **2.1.2.2. Strona logowania**

URL	/login
-----	--------

<b>Kontroler</b>	views.py/login
<b>Szablon</b>	login.html
<b>Opis</b>	
<p>Kontroler po otrzymaniu żądania typu GET od przeglądarki generuje stronę logowania zawartą w szablonie login.html. Logowanie można wykonać na stronie głównej lub pod adresem /login. Jeżeli użytkownik błędnie wpisze dane do logowania na stronie głównej zostanie przekierowany do strony logowania. W przypadku, kiedy użytkownik wpisze dane do logowania i wciśnie przycisk zaloguj, kontroler obsługuje zgłoszenie POST. Wówczas, jeżeli dane są poprawne, zakładana jest sesja użytkownika i następuje przekierowanie do strony głównej. W przypadku wystąpienia błędu generowany jest szablon login.html z odpowiednim komunikatem.</p>	



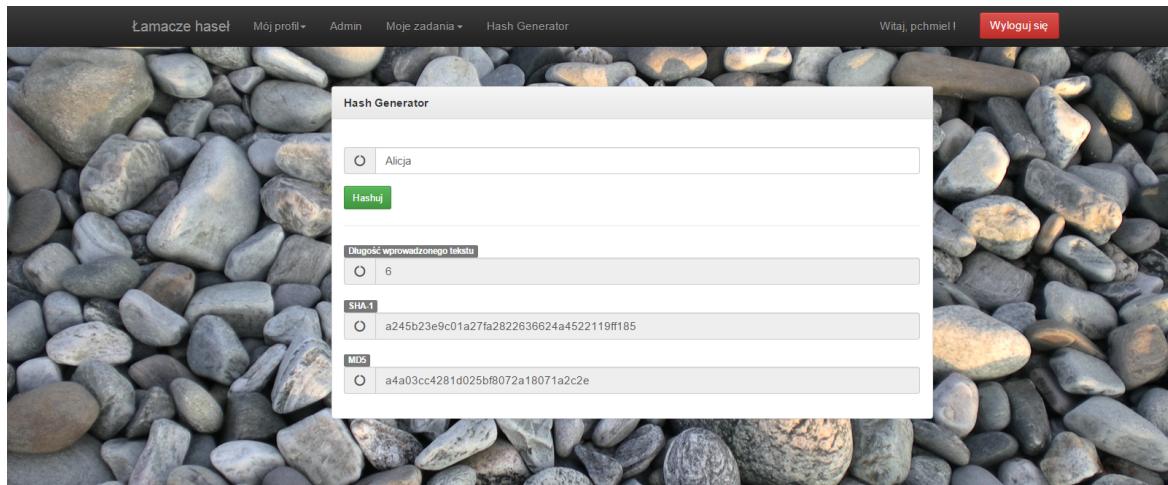
#### 2.1.2.3. Wylogowanie

<b>URL</b>	/logout
<b>Kontroler</b>	views.py/logout
<b>Szablon</b>	index.html
<b>Opis</b>	

Kontroler po otrzymaniu żądania od przeglądarki ( użytkownik wciśnie przycisk wyloguj na stronie głównej) zamyka sesję użytkownika i przekierowuje go na stronę główną

#### 2.1.2.4. Hash Generator

<b>URL</b>	/hash_generator /generate_hash
<b>Kontrolery</b>	views.py/hash_generator; views.py/generate_hash
<b>Szablon</b>	hash_generator.html
<b>Opis</b>	
Kontroler po otrzymaniu żądania od przeglądarki ( użytkownik wciśnie przycisk Hash Generator na stronie głównej) metodą render generuje szablon hash_generator.html. Użytkownik wpisuje text do hashowania i wciska przycisk hashuj. Wysyłane jest ajaxowe żądanie pod adres generate_hash. Obsługujący je kontroler wykonuje hashowanie za pomocą modułu hashlib. Do widoku zwracany jest json z hashami. Za pomocą funkcji jQuery hashe wpisywane są w odpowiednie pola.	

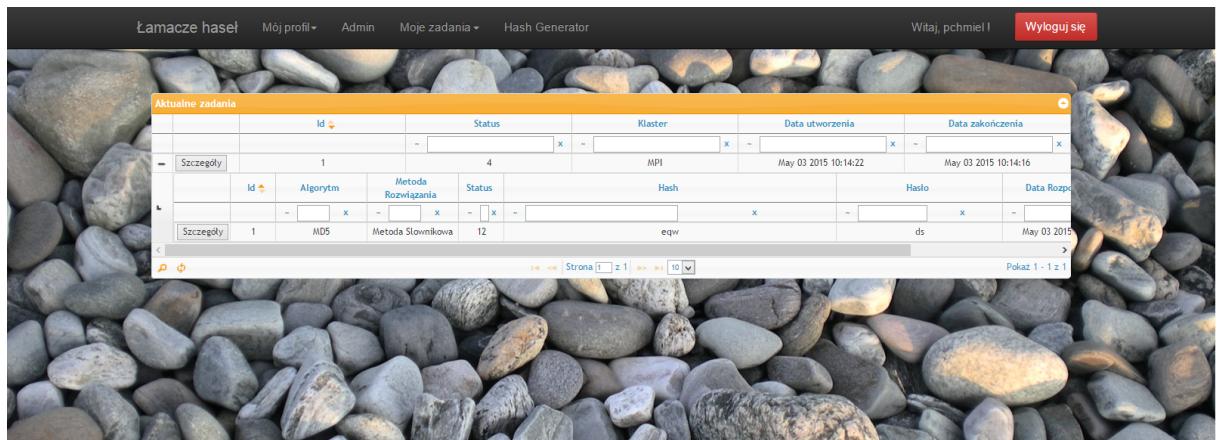


Rysunek 4 Widok hash generator

#### 2.1.2.5. Aktualne zadania

<b>URL</b>	/task_actual
<b>Kontrolery</b>	views.py/task_actual

<b>Szablon</b>	task_actual.html
<b>Opis</b>	
Kontroler po otrzymaniu żądania od przeglądarki ( użytkownik wciśnie przycisk Moje Zadanie -> Aktualne Zadania na stronie głównej) metodą render generuje szablon task_actual.html. Do kontekstu wysyłane są zadania, które mają status różny od 100. Do wygenerowania tabeli wykorzystywana jest wtyczka jqGrid. Jego konfiguracja znajduje się w pliku task_actual.js	



Rysunek 5 Widok aktualne zadania

#### 2.1.2.6. Historia zadań

<b>URL</b>	/task_history
<b>Kontrolery</b>	views.py/task_history
<b>Szablon</b>	task_history.html
<b>Opis</b>	
Kontroler po otrzymaniu żądania od przeglądarki ( użytkownik wciśnie przycisk Moje Zadanie -> Historia Zadań na stronie głównej) metodą render generuje szablon task_actual.html. Do kontekstu wysyłane są zadania, które mają status równy 100. Do wygenerowania tabeli wykorzystywana jest wtyczka jqGrid. Jego konfiguracja znajduje się w pliku task_history.js	

#### 2.1.2.7. Szczegóły zadania

<b>URL</b>	/task_details/<?Pid>\w+
------------	-------------------------

<b>Kontrolery</b>	views.py/task_details
<b>Szablon</b>	task_details.html
<b>Opis</b>	
Kontroler po otrzymaniu żądania od przeglądarki ( użytkownik wciśnie przycisk Szczegóły obok wybranego zadania w tabeli) generuje okno typu pop-up przy wykorzystaniu klasy Dialog jQuery-UI. W urlu przyjmowany jest parametr id zadania, podawany automatycznie, pobierany z tabeli. Do okna przesyłane są dane dotyczące zadania. Następuje ajaxowe odświeżanie paska statusu. Co 3 sekundy kierowane jest zapytanie pod adres /get_status.	

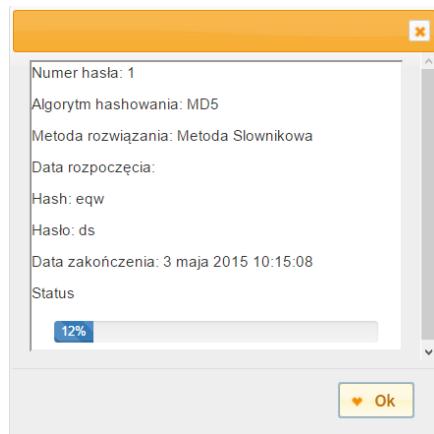


Rysunek 6 Szczegóły Zadania

#### 2.1.2.8. Szczegóły zadania dla pojedynczego hasła

<b>URL</b>	/password_details/<?Pid>\w+
<b>Kontrolery</b>	views.py/password_details
<b>Szablon</b>	password_details.html
<b>Opis</b>	

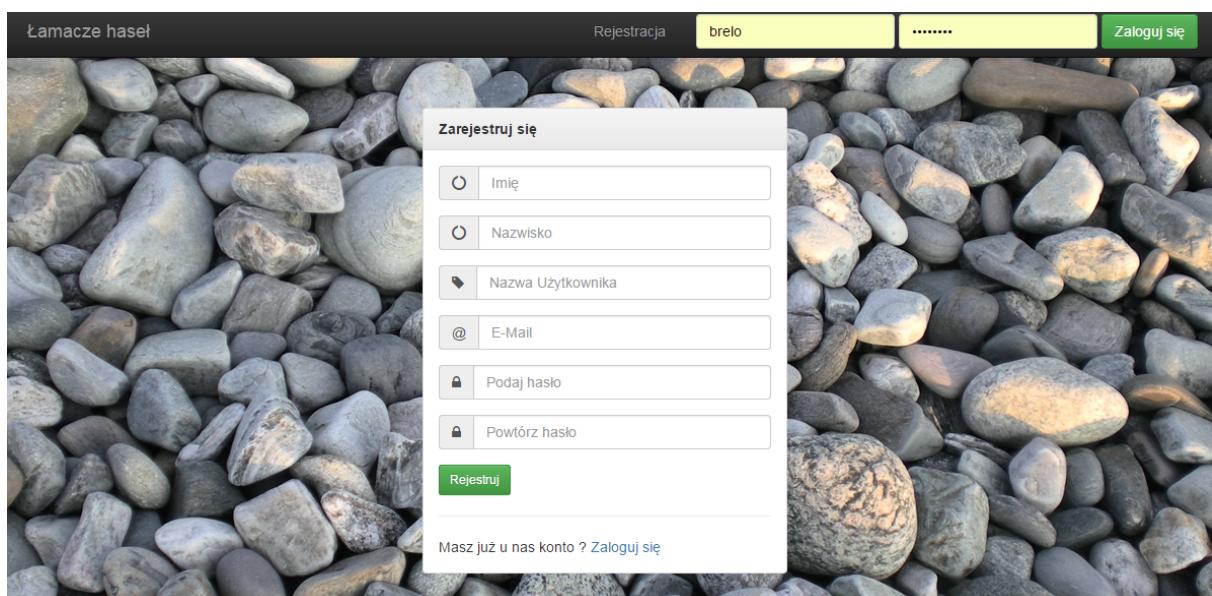
Kontroler po otrzymaniu żądania od przeglądarki ( użytkownik wcisnie przycisk Szczegóły obok wybranego hasła w tabeli po rozwinięciu zadania, do którego przynależy to hasło) generuje okno typu pop-up przy wykorzystaniu klasy Dialog jQuery-UI. W urlu przyjmowany jest parametr id hasła, podawany automatycznie, pobierany z tabeli. Do okna przesyłane są dane dotyczące rozwiązania hasła. Następuje ajaxowe odświeżanie paska statusu. Co 3 sekundy kierowane jest zapytanie pod adres /get\_status.



Rysunek 7 Szczegóły zadania dla pojedynczego hasła

### 2.1.2.9. Rejestracja

URL	/register/
Kontrolery	views.py/register
Szablon	register.html
<b>Opis</b>	
Kontroler po otrzymaniu żądania od przeglądarki (kliknięcie przycisku Rejestruj) pobiera dane z wypełnionych pól i sprawdza, czy wprowadzone dane są prawidłowe. (np. czy hasła pasują do siebie) Jeżeli formularz zawiera błędy są one wyświetlane na tej samej stronie, jeżeli jest wypełniony poprawnie następuje zapis użytkownika do bazy danych i przekierowanie na stronę potwierdzającą prawidłową rejestrację. (/register_success)	

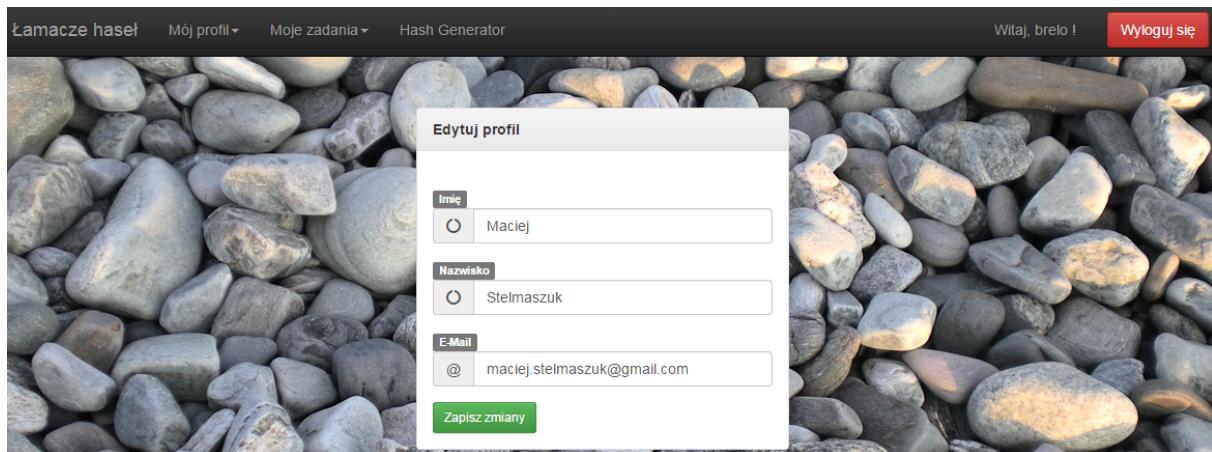


Rysunek 8 Rejestracja użytkownika

### 2.1.2.10. Edycja Profilu

URL	/edit_profile/
Kontrolery	views.py/edit_profile
Szablon	edit_profile.html
<b>Opis</b>	

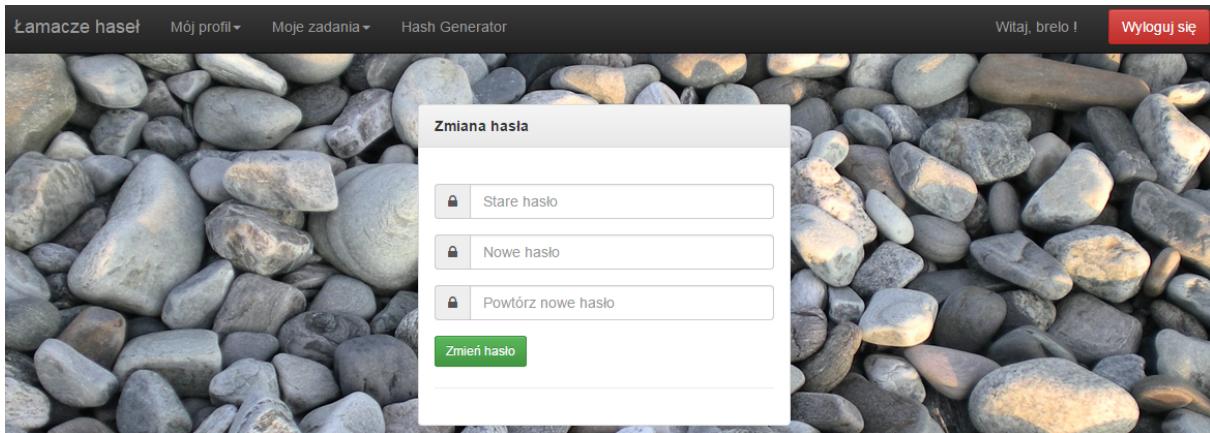
Kontroler po otrzymaniu żądania od przeglądarki (kliknięcie przycisku Zapisz zmiany) pobiera dane z wypełnionych pól i sprawdza, czy wprowadzone dane są prawidłowe. (np. czy adres e-mail jest poprawny) Jeżeli formularz zawiera błędy są one wyświetlane na tej samej stronie, jeżeli jest wypełniony poprawnie następuje aktualizacja danych w bazie danych.



Rysunek 9 Edycja profilu

#### 2.1.2.11.Zmiana Hasła

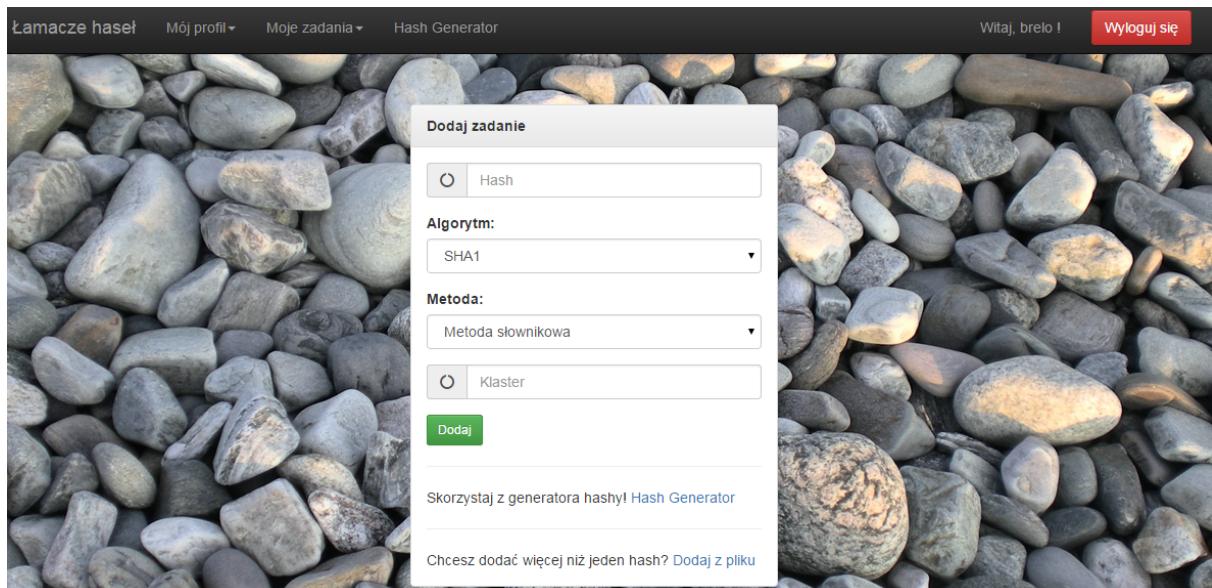
<b>URL</b>	/password_change/
<b>Kontrolery</b>	views.py/password_change
<b>Szablon</b>	password_change.html
<b>Opis</b>	
Kontroler po otrzymaniu żądania od przeglądarki (kliknięcie przycisku Zmień hasło) pobiera dane z wypełnionych pól i sprawdza, czy wprowadzone dane są prawidłowe. (czy stare hasło jest prawidłowe oraz czy nowe hasła pasują do siebie) Jeżeli formularz zawiera błędy są one wyświetlane na tej samej stronie, jeżeli jest wypełniony poprawnie następuje aktualizacja danych w bazie danych - zmiana hasła użytkownika.	



Rysunek 10 Widok zmiany hasła

### 2.1.2.12.Dodanie zadania

<b>URL</b>	/add_task/ , /mpi_send/
<b>Kontrolery</b>	views.py/add_task , mpi.py/mpi_send
<b>Szablon</b>	add_task.html
<b>Opis</b>	
Po wypełnieniu wszystkich pól i kliknięciu przycisku Dodaj uruchamiany jest skrypt, który weryfikuje czy wprowadzone dane są poprawne. (czy długość hashy jest odpowiednia w zależności od algorytmu hashowania) Jeżeli dane są niepoprawne zostaje wyświetlony komunikat błędu. Jeżeli dane są poprawne wtedy skrypt przetwarza je na format JSON i wysyła pod url: /mpi_send/. W dalszej części zadanie jest zapisywane do bazy danych. Po poprawnym dodaniu zadania następuje przekierowanie na stronę z aktualnymi zadaniami. (/task_actual/)	

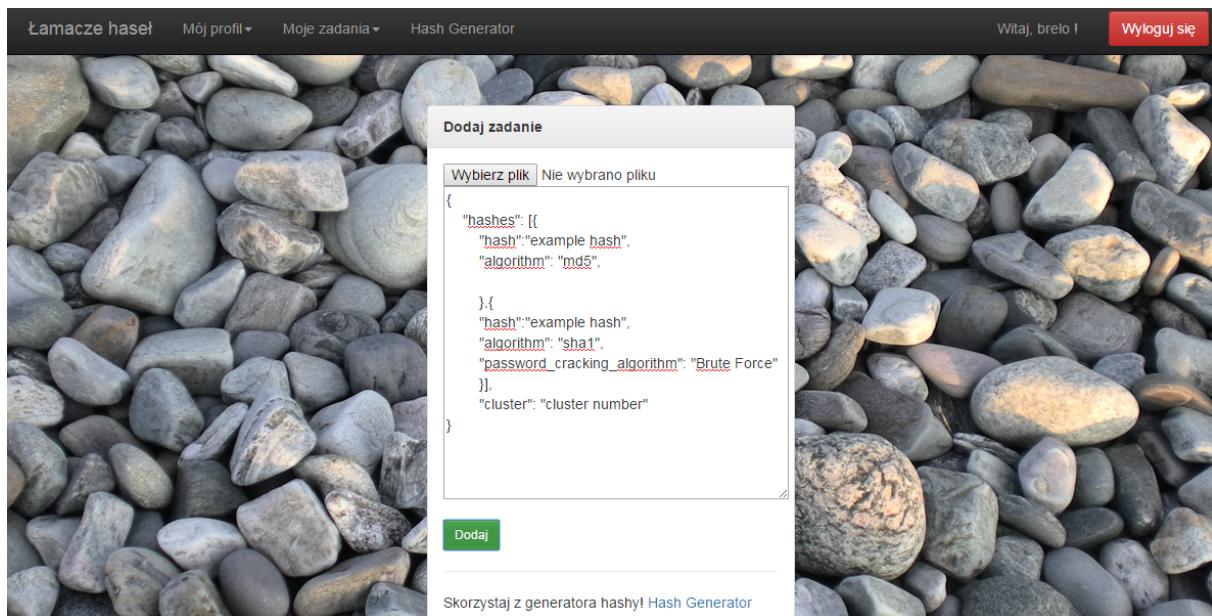


Rysunek 11 Dodawanie pojedynczego zadania

### 2.1.2.13.Dodanie zadania z pliku

<b>URL</b>	/add_tasks_from_file/ , /mpi_send/
------------	------------------------------------

<b>Kontrolery</b>	views.py/add_tasks_from_file , mpi.py/mpi_send
<b>Szablon</b>	add_tasks_from_file.html
<b>Opis</b>	
<p>Widok umożliwia wczytywanie zadań z pliku. Po kliknięciu przycisku Przeglądaj i wybraniu pliku w odpowiednim formacie (*.txt) jego zawartość zostanie załadowana do okna, gdzie dodatkowo można ją modyfikować. Po kliknięciu przycisku Dodaj wywoływany jest skrypt, który dokonuje konwersji zawartości pola tekstowego na format JSON. Wprowadzone dane nie są weryfikowane, po poprawnym dodaniu zadań użytkownik jest przekierowywany na stronę z aktualnymi zadaniami (/task_actual/). W przeciwnym wypadku użytkownik zostaje na tej samej stronie.</p>	



Rysunek 12 Dodawanie wielu zadań z pliku

#### 2.1.2.14. Strona Administratora

Pod adresem /admin dostępna jest witryna administratora. Została wykorzystana domyślna strona, którą oferuje nam framework Django. Tabele, które znajdują się na stronie oraz ich komórki zdefiniowane są w pliku admin.py.

Administracja Django

Witaj, **pchmiel**. Zmiana hasła / Wyloguj się

**Administracja stroną**

Aplikacja Kliencka	
Passwords	<a href="#">Dodaj</a> <a href="#">Zmień</a>
Tasks	<a href="#">Dodaj</a> <a href="#">Zmień</a>

Autentykacja i autoryzacja	
Grupy	<a href="#">Dodaj</a> <a href="#">Zmień</a>
Użytkownicy	<a href="#">Dodaj</a> <a href="#">Zmień</a>

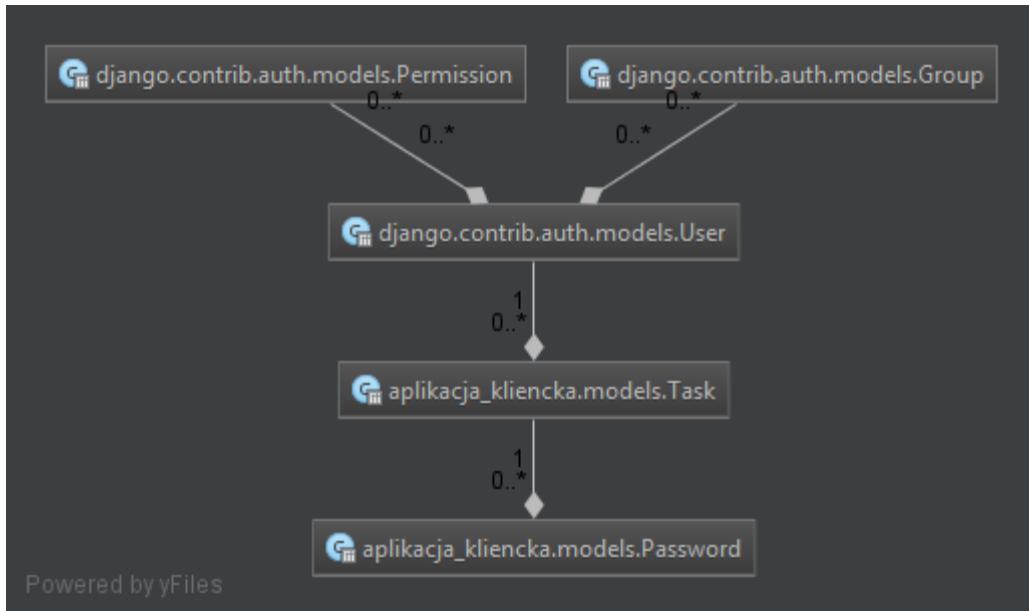
**Ostatnie akcje**

Moje akcje
<a href="#">Password object</a>
<a href="#">Hasło</a>
<a href="#">MPI</a>
<a href="#">Task</a>

Rysunek 13 Strona Administratora

## 2.1.3. Modele

### 2.1.3.1. Schemat Bazy Danych



Rysunek 14 Schemat bazy danych

### 2.1.3.2. Opis Bazy Danych

W projekcie wykorzystujemy oferowane przez Django modele User, Group oraz Permission, dzięki którym realizowany jest mechanizm logowania, wylogowywania oraz utrzymywania sesji. Oprócz tego w projekcie wykorzystujemy stworzone przez nas modele Task i Password.

### 2.1.3.3. Model Task

```
class Task(models.Model):
    user = models.ForeignKey(User)
    status = models.IntegerField()
    cluster = models.CharField(max_length=250)
    creation_date = models.DateTimeField(auto_now_add=True)
    end_time = models.DateTimeField(null=True, blank=True)
```

**User** - klucz obcy encji User, informuje o tym, który użytkownik zlecił zadanie.

**Status** - pole przechowujące wartości 0-100. Informuje o postępie w realizacji zadania.

**Cluster** - nazwa klastra, na którym realizowane jest zadanie.

**Creation Date** - data utworzenia zadania, wypełniany automatycznie w momencie dodawania zadania.

**End Time** - data zakończenia zadania, wypełniana w momencie złamania hasła lub „poddania się” algorytmów.

#### 2.1.3.4. Model Password

```
CRACKING_ALGORITHMS = (
    ('MS', 'Metoda Słownikowa'),
    ('TT', 'Tablice Teczowe'),
    ('BF', 'Brute Force'),
)
hash = models.CharField(max_length=250)
password = models.CharField(max_length=250, null=True, blank=True)
task = models.ForeignKey(Task, related_name='passwords')
status = models.IntegerField()
start_time = models.DateTimeField(null=True, blank=True)
end_time = models.DateTimeField(null=True, blank=True)
algorithm = models.CharField(max_length=30)
password_cracking_algorithm = models.CharField(max_length=2, choices=CRACKING_ALGORITHMS)
```

**Hash** - pole przechowujące hash.

**Password** - pole, w które wpisywane jest hasło po złamaniu.

**Task** - klucz obcy informujący o przynależności pojedynczego hasła do zadania (paczki haseł).

**Status** - pole o wartościach (0-100) informujące o postępie w łamaniu hasła.

**Start Time** - data rozpoczęcia łamania hasła.

**End Time** - data zakończenia łamania hasła.

**Algorithm** - pole informujące o tym jakiego algorytmu hashowania użyto.

**Password Cracking Algorithm** - pole informujące o tym jakiego algorytmu łamania hasła użyto.

### 2.2. Algorytmy łamania haseł

#### 2.2.1. Metoda słownikowa (atak słownikowy)

Atak słownikowy to technika siłowego odgadywania haseł, jest w pewien sposób zbliżona do brute force. W tej metodzie ważny jest słownik, który jest niezbędny. Każde słowo z takiego słownika jest porównywane z hasłem, po

wcześniejszym obliczeniu wartości funkcji skrótu. Ataki są skuteczne tam, gdzie systemy nie używają wielowyrazowych fraz. W naszej implementacji algorytmu kombinacje dużych i małych liter nie chronią przed atakiem. Wyższość ataku słownikowego nad metodą „Brute Force” to krótszy czas działania w systemach, gdzie ludzie ustawiają hasła. Z kolei atak słownikowy nie zadziała tam, gdzie są ustawione kombinacje znaków, których nie ma w słowniku.

#### 2.2.1.1. Sposób implementacji

1. Wczytanie pliku słownika.
2. Wczytanie jednego hasła ze słownika. Hasło zostaje przekazane do funkcji **hackify**, która z pomocą liczb zapisanych dwójkowo, modyfikuje hasło zmieniając wielkość liter. Przykład:

hasło: mail, reprezentacja bitowa: 0000

hasło: Mail, reprezentacja bitowa: 1000

hasło: mAiL, reprezentacja bitowa: 1010

3. Każde hasło zmodyfikowane w funkcji **hackify**, zostaje przekazane do funkcji **checkSuffixes**, gdzie najpierw się oblicza skrót MD5 lub SHA1, a następnie jeśli hasło nie jest złamane, zostają dodane przyrostki, które są wszystkimi znakami ASCII oprócz liter. Przykład:

Funkcja **checkSuffixes** dostała hasło "leMOn" od funkcji **hackify**:

leMOn1 -> leMOn2 -> leMOn3 -> ... -> leMOn! -> ... -> leMOn1! -> leMOn1@ -> ... -> leMOn!@

4. Każda kombinacja hasła jest sprawdzana, jeśli hasło nie jest złamane i wszystkie kombinacje zostały sprawdzone, algorytm pobiera następne hasło i rozpoczyna procedurę.

Obecnie prawie wszystkie systemy wymagają haseł z wielkimi literami, cyframi i znakiem specjalnym. Dzięki funkcjom **hackify** i **checkSuffixes** mamy większe prawdopodobieństwo złamania hasła. Bez nich wczytujemy hasła i tylko sprawdzamy czy jest takie same. Dzięki temu możemy ograniczyć słownik do

tych haseł, które składają się tylko z liter. W programie używamy słownika, który posiada ponad 300 000.

#### **2.2.1.2. Ważne fragmenty kodu programu**

## Funkcja modyfikująca wielkość liter (hackify):

```
std::string Attack::hackify(std::string pass)
{
    std::string s;
    std::string pass2;
    std::string result = "";
    int x = pass.length();
    for(int i = 0; i < pow(2, pass.length()); i++)
    {
        pass2 = pass;
        const boost::dynamic_bitset<> b2(x, i);
        boost::to_string(b2, s);
        for(int j = 0; j < s.length(); j++)
        {
            if(s[j] == '1')
            {
                pass2[j] = toupper(pass[j]);
            }
            else if (s[j] == '0')
            {
                pass2[j] = tolower(pass[j]);
            }
        }
        result = checkSuffixes(pass2, 0);
        if (result != "")
            break;
    }
    return result;
}
```

**Funkcja doklajająca przyrostki (checkSuffixes):**

```
std::string Attack::checkSuffixes(std::string pass, int level)
{
    const int max_level = 3;
    std::string result = "";
    if(level != max_level) {
        std::string pass2 = pass;
        for(int i = 0; i < suffixes.length(); i++) {
            Hash h;
            if (function == HashingFunction::SHA1) {
                Hash tmp = CryptoUtils::generateSHA1(pass2);
                h = tmp;
            }
            else if (function == HashingFunction::MD5) {
                Hash tmp = CryptoUtils::generateMD5(pass2);
                h = tmp;
            }
            if(CryptoUtils::convertHashToHexRep(h) == this->key) {
                result = pass2;
                break;
            }
            pass2 = pass;
            pass2 += suffixes[i];
            result = checkSuffixes(pass2, level+1);
            if (result != "") break;
        }
    }
    return result;
}
```

## **2.2.2.Brute Force**

Algorytm Brute Force realizuje idee przeglądu zupełnego dla problemu łamania haseł. Innymi słowy generuje oraz sprawdza funkcje skrótu dla wszystkich możliwe kombinacje znaków. Oprócz wartości funkcji skrótu poszukiwanego hasła, parametrami algorytmu są także początkowy oraz końcowy ciąg znaków. Określają one jednoznacznie granice zbioru haseł, w którym poszukiwane będzie hasło. Takie podejście umożliwia zastosowanie algorytmu w aplikacjach wielowątkowych.

### **2.2.2.1. Sposób implementacji**

1. Pobranie parametrów wywołania. Są to poszukiwana wartość funkcji skrótu, początkowy ciąg znaków, końcowy ciąg znaków. Poszukiwania hasła zaczynają się naturalnie od początkowego ciągu znaków.
2. Obliczenie wartości funkcji skrótu dla ciągu znaków. Do generowania wartości funkcji skrótu SHA-1 oraz MD5 wykorzystano bibliotekę OpenSSL.
3. Porównanie obliczonej wartości z wartością poszukiwaną.
4. Jeśli wynik porównania jest pozytywny, wtedy algorytm kończy pracę oraz zwraca znaleziony ciąg znaków.
5. W przeciwnym przypadku następuje wygenerowanie kolejnego ciągu znaków poprzez jego inkrementacje. Szczegóły implementacji tej funkcjonalności zostały opisane w kolejnej sekcji.
6. Algorytm kończy pracę, jeśli wygenerowany ciąg znaków jest równy końcowej wartości podanej w parametrze wywołania.

Przyjęto, że algorytm sprawdza możliwe kombinacje w podanym zakresie dla wszystkich drukowalnych znaków ASCII, łącznie jest ich 94.

Inkrementacje ciągu znaków można rozumieć jak inkrementacje liczby w takim systemie liczbowym, gdzie są 94 cyfry reprezentowane przez kolejne drukowalne znaki kodu ASCII. Cyfra o najniższej wartości w tym kodzie jest! (wykrzyknik), a o największej ~ (tylda). W operacji wzięto pod uwagę przeniesienie przy dodawaniu.



### 2.2.2.2. Ważne fragmenty kodu programu

Funkcja inkrementująca ciąg znaków (`incrementString`):

```
void CryptoUtils::incrementString(std::string &s, int stringPosition) {
    if(s.size() <= stringPosition) {
        s.append("!");
    } else {
        s[stringPosition] = s[stringPosition] + 1;
        if(s[stringPosition] > 126) {
            s[stringPosition] = '!';
            incrementString(s, stringPosition+1);
        }
    }
}
```

### 2.2.3. Tablice tęczowe

Łamanie haseł przy pomocy tablic tęczowych polega na połączeniu dwóch innych metod rozwiązywania tego problemu - brute force i słownikowej. Na podstawie obliczonych skrótów generowane są kolejne ciągi znaków aż do wygenerowania określonej liczby haseł zawartych w danym łańcuchu. Metoda tablic tęczowych pozwala na oszczędzenie mocy obliczeniowej w stosunku do algorytmu brute force - nie ma potrzeby generowania kolejnych możliwych kombinacji łańcuchów znakowych, zamiast tego wykorzystuje się wygenerowane wcześniej skróty. Kolejną z zalet metody tablic tęczowych jest znaczne ograniczenie powierzchni zajmowanej przez wygenerowane zbiory skrótów w stosunku do rozmiarów danych wykorzystywanych przez metodę tablic tęczowych. Do wygenerowania kolejnych haseł w metodzie tablic tęczowych wykorzystuje się tzw. funkcję redukującą. Generuje ona nowy ciąg znaków na podstawie bieżącego. Przy znajomości funkcji redukującej i jej parametrów, możliwe jest odtworzenie całego łańcucha haseł. Przy generowaniu tablic tęczowych istotne jest zapobieganie kolizjom hashy, co jest często realizowane poprzez na przykład wykorzystywanie funkcji redukującej, której wynik zależy od aktualnie analizowanego wiersza.

### 2.2.3.1. Sposób implementacji

Metoda łamania haseł przy wykorzystaniu haseł tęczowych została zaimplementowana w klasie RainbowCracker. Z uwagi na ograniczenia sprzętowe (wygenerowanie tablic tęczowych jest intensywne obliczeniowo) algorytm jest w stanie złamać stosunkowo krótkie (ok. 4 znaków) hasła składające się z cyfr i stanowi raczej "proof of concept" niż pełne rozwiązanie problemu. Redukcja danego ciągu znaków polega na pobraniu z niego ilości cyfr odpowiadającej aktualnie analizowanej długości hasła. Brakujące cyfry uzupełniane są zerami. Metoda tablic tęczowych potrafi złamać hasła, które zostały przetworzone za pomocą funkcji SHA-1 i MD5.

### 2.2.3.2. Ważne fragmenty kodu programu

#### Funkcja redukująca (reduce):

```
std::string RainbowCracker::reduce(Hash& hash, int position, int table_index, int password_length) {
    int extracted_digits = 0;
    std::string new_key;
    std::string hex_rep = CryptoUtils::convertHashToHexRep(hash);
    for (auto it = hex_rep.begin(); it != hex_rep.end(); ++it) {
        unsigned char tmp = *it;
        if (tmp >= '0' && tmp <= '9') {
            new_key += tmp;
            if (++extracted_digits == password_length) break;
        }
    }
    if (extracted_digits < password_length) {
        for (int i = 0; i < password_length - extracted_digits; ++i)
            new_key += "0";
    }
    return new_key;
}
```

### 2.3. Komunikacja między modułami

Moduły (aplikacja kliencka i węzły MPI) komunikują się poprzez URL /mpi\_send. Na ten adres wysyłane są komunikaty w formacie JSON.

**Format danych:**

```
{  
  "hashes": [{  
    "hash": "111",  
    "algorithm": "md5"  
  }, {  
    "hash": "2",  
    "algorithm": "sha1"  
  }],  
  "cluster": "0",  
  "graininess": 1/2/3  
}
```

**cluster** - identyfikator klastra, na którym chcemy wykonać zadanie

**graininess** - poziom ziarnistości dla zadania

Sockety przez jakie komunikują się Django i master klastra to sockety ZeroMQ. Twórcy tej biblioteki/frameworka nazywają swoje sockety socketami na sterydach. Wiadomości przesyłane są w architekturze PUB - SUB zarówno od Django jak i od MPI. Wiadomości to zwyczajne stringi jednak każdy SUB rejestruje, jakie wiadomości go interesują. Filtr ten odsiewa wiadomości niezaczynające się od zarejestrowanego znaku/stringa (cluster\_id). ClusterId ustawiany jest, jako zmienna środowiskowa przed uruchomieniem mpi przez połączenie ssh. Następnie w skrypcie startującym mpi zmienna ta przekazywana jest jako parametr wywołania mpirun. Ostateczna forma wiadomości kierowanej do klastra o id "1" wygląda następująco:

```
"1 {"id": "12", "hash": "21uwuh2rf1", "algorithm": "md5/sha1", "graininess": 1/2/3}"
```

Wiadomości zwrotne tagowane są literą D gdyż takich wiadomości nastuchuje Django. Wiadomość zwrotna:

"D {"id": "12", "password": "trudneHaslo", "start\_time": 123, "end\_time": 321}"

### **3. Konfiguracja i uruchomienie systemu**

#### **3.1. Środowisko pracy**

Wszystkie węzły klastra obliczeniowego wymagają do swojego działania systemu Linux. Poniższe wskazówki dot. wymaganych bibliotek i pakietów pozwalają na pełną konfigurację węzła z systemem operacyjnym Ubuntu 14.04.2-LTS z jądrem 3.13.0-53-generic. W wypadku korzystania z innych wersji/dystrybucji może zaistnieć potrzeba instalacji innych pakietów.

#### **3.2. Aplikacja wykonująca obliczenia (C++)**

##### **3.2.1. Kompilacja**

Aby poprawnie skompilować aplikację wykonującą obliczenia na danym węźle należy ściągnąć z repozytorium systemu Ubuntu następujące pakiety:

- openmpi-bin
- openmpi-common
- libopenmpi-dev
- libopenmpi1.6-dbg
- libopenmpi1.6
- build-essential
- libtool
- pkg-config
- autoconf
- automake
- cmake
- libssl-dev

Należy pobrać też niedostępna w repozytorium systemu Ubuntu bibliotekę ZeroMQ. Biblioteka dostępna jest na stronie <http://zeromq.org/> w formacie \*.tar.gz. Po ściągnięciu i rozpakowaniu ZeroMQ, należy odpowiednio wywołać komendy

"configure", "make" i "make install". Więcej informacji dot. konfiguracji ZeroMQ można odnaleźć na stronie twórców biblioteki. Potrzebne do poprawnego działania nagłówki dla języka C++ dostępne są w repozytorium GitHub: <https://raw.githubusercontent.com/zeromq/cppzmq/master/zmq.hpp>. Po pobraniu należy umieścić plik zmq.hpp w folderze /usr/local/include.

Kod źródłowy aplikacji wykonującej obliczenia (folder "mpi" w folderze głównym repozytorium) należy umieścić w katalogu /vagrant na węźle typu "master", a następnie zbudować wydając polecenie "make".

### **3.2.2.Urchomienie**

Aby zapewnić możliwość łatwej modyfikacji klastra, każdy z węzłów powinien posiadać oprogramowanie umożliwiające mu pracę zarówno w trybie "slave" jak i "master".

Wymagane pakiety:

- openssh-client
- openssh-server
- nfs-common
- nfs-kernel-server

### **3.2.3.Konfiguracja węzła jako część klastra**

#### **3.2.3.1. Węzeł typu „slave”**

- Utworzenie użytkownika "vagrant".
- Utworzenie katalogu /vagrant oraz nadanie rekurencyjnie właściciela (wcześniej stworzony użytkownik "vagrant").
- Dodanie do /etc/fstab wpisu montującego udostępniony przez węzeł typu "master" przez NFS katalogu /vagrant do lokalnego katalogu /vagrant.
- Import klucza publicznego wygenerowany na węźle master do pliku ~/.ssh/authorized\_keys (użytkownik "vagrant").
- Dodanie wpisu do konfiguracji SSH dla każdego hosta "Host \*\n\tStrictHostKeyChecking no".

### **3.2.3.2. Węzeł typu „master”**

- Utworzenie użytkownika "vagrant".
- Utworzenie katalogu /vagrant oraz nadanie rekurencyjnie właściciela (wcześniej stworzony użytkownik "vagrant").
- Utworzenie pliku .mpi\_np w katalogu domowym użytkownika "vagrant". Zapisanie w nim liczby procesów do uruchomienia przez klaster.
- Utworzenie pliku .mpi\_hostfile w katalogu domowym użytkownika "vagrant". Zapisanie w nim zgodnie dokumentacją OpenMPI nazw węzłów typu "slave" (jeżeli w sieci znajduje się server DNS z wpisami dot. węzłów typu "slave" bądź jeżeli nazwy rozwijane są w pliku /etc/hosts) lub ich adresy. Określenie ilości procesów, jaka ma być uruchomiona na danym węźle typu slave - maksymalnej oraz minimalnej liczby.
- Skopiowanie do katalogu /etc/init.d dostarczony skrypt mpi\_run\_script z nazwą mpi oraz nadanie prawa wykonywania dla użytkownika "vagrant" (skrypt uruchamiany jest przez Django, które łączy się z węzłem master przez SSH, w skrypcie eksportowane są ustawienia klastra dostarczone przez Django oraz aplikacja wykonująca obliczenia jest uruchamiana jako demon).
- Udostępnienie katalogu /vagrant poprzez NFS dla każdego hosta w sieci klastra.
- Wygenerowanie klucza publicznego i prywatnego.
- Dodanie wpisu do konfiguracji SSH dla każdego hosta "Host \*\n\tStrictHostKeyChecking no".

## **3.3. Aplikacja kliencka (Django)**

Serwer aplikacji klienckiej powstał w oparciu o język Python 3 (testowana dla wersji Python 3.4.0) oraz framework Django w wersji 1.7.2.

### **1. Wymagane biblioteki**

Do poprawnego działania aplikacji klienckiej wymagane są następujące pakiety z repozytorium Ubuntu:

- python3-pip

Pakiety instalowane za pośrednictwem pip:

- Django (1.7.2)
- paramiko (1.15.2)
- pyzmq (14.5.0)
- python3-pika (0.9.14)

Oprócz tego do poprawnego działania aplikacji potrzebne są następujące technologie:

- HTML5
- Bootstrap 3.3.4
- jQuery 2.1.3
- jqGrid 4.7.1

## 2. Konfiguracja

W pliku settings.py znajdującym się w folderze Lamacze\_Hasel/Lamacze\_Hasel należy zmodyfikować zawartość zmiennych MY\_SOCKET\_ADDRESS oraz CLUSTERS. W pierwszej powinien znaleźć się adres adres IP serwera aplikacji klienckiej widoczny w sieci klastra wraz z portem 5557. W drugiej natomiast należy umieścić dane dot. adresu węzła typu "master" klastra. W polu "address" jego IP w sieci klastra, a w polu socket\_address jego IP wraz z portem 5557. Przykładowa konfiguracja:

```
MY_SOCKET_ADDRESS = 'tcp://172.28.128.1:5557'
```

```
CLUSTERS = [{"id":'1', 'address': '172.28.128.3', 'username': 'vagrant', 'password': 'vagrant', 'socket_address':'tcp://172.28.128.3:5558'}]
```

## 3. Uruchomienie

Aby uruchomić serwer aplikacji klienckiej należy przejść do folderu Lamacze\_Hasel i wydać polecenie "python3 manage.py runserver". Uruchomienie serwera aplikacji klienckiej spowoduje uruchomienie całego klastra obliczeniowego zgodnie ze stworzoną konfiguracją.

## 4. Testy wydajnościowe

Testy zostały przeprowadzone na klastrze oraz na laptopie. Klaster obliczeniowy składał się z 5 slave'ów, z czego każdy slave używał 2 rdzeni, co łącznie dawało 10 rdzeni. Laptop używał tylko 2 rdzeni.

Poniżej przedstawiono wyniki testów w tabelce, oraz porównanie metody Brute Force dla klastra i laptopa.

Laptop		hasło: 123		
Hasło: aaa11	Metoda			
	Brute Force	Słownikowa	Tablice Tęczowe	
Ziarnistość	Czas			
mała	4	6	4	
średnia	6	7	3	
duża	8	7	3	
Klaster	5 slavów(każdy po 2 rdzenie)			
Hasło: AaM!!	Metoda			
	Brute Force	Słownikowa	Tablice Tęczowe	
Ziarnistość	Czas			
mała	1	2	2	
średnia	2	4	1	
duża	3	5	1	

