WOJSKOWA AKADEMIA TECHNICZNA

im. Jarosława Dąbrowskiego

WYDZIAŁ CYBERNETYKI

PRACA DYPLOMOWA

STUDIA I°

Temat pracy: **SPOŁECZNOŚCIOWY SERWIS HELPDESK**

INFORMATYKA

(kierunek studiów)

INŻYNIERIA SYSTEMÓW

(specjalność)

Wykonał:

Radosław RELIDZYŃSKI

Prowadzący:

prof. dr. hab. inż. Andrzej WALCZAK

Warszawa 2024

OŚWIADCZENIE

~~Wyrażam zgodę~~ / nie wyrażam[*] zgody
na udostępnianie mojej pracy w czytelni Archiwum WAT.

Dnia 22.01.2024 r.

Radosław Reliszyński
.....
(podpis)

[*]*Niepotrzebne skreślić*

Spis treści

Wstęp

Tematem mojej pracy inżynierskiej jest projekt i implementacja społecznościowego serwisu typu helpdesk, czyli aplikacji internetowej pozwalającej jej użytkownikom na tworzenie oraz przeglądanie wpisów zawierających poradniki i instrukcje. Serwis będzie prezentował użytkownikowi na jakie problemy może natrafić i jak je we własnym zakresie rozwiązywać nie posiadając większych kompetencji w zakresie obsługi komputera

Docelowo strona będzie zorientowana wokół instruktorów Związku Harcerstwa Polskiego, ale korzystać z niej będzie mógł każdy. Taki wybór grupy docelowej uwarunkowany jest tym, że sam jestem instruktorem tej organizacji i osobiście doświadczam sytuacji wskazujących na potrzebę powstania przestrzeni, w której ludzie nie posiadający obycia z komputerem czy Internetem będą mogli dowiedzieć się, w jaki sposób poradzić sobie w świecie cyfrowym.

Celem projektu jest wyjście naprzeciw tym potrzebom i stworzenie właśnie takiej przestrzeni. System będzie umożliwiał zebranie w jedno miejsce niezbędnych wskazówek, instrukcji oraz sugestii pozwalających na efektywniejszą pracę na komputerze oraz sprawniejsze odnajdowanie się w świecie różnych interfejsów programów, jakie są powszechnie używane.

Praca została podzielona na kilka rozdziałów, z których każdy koncentruje się na innym aspekcie projektu - od analizy potrzeb i wymagań, poprzez projektowanie systemu, aż po jego implementację i testowanie. Zakończenie pracy skupia się na podsumowaniu osiągnięć, wskazaniu potencjalnych dalszych kierunków rozwoju projektu oraz refleksji nad doświadczeniami wyniesionymi z realizacji projektu.

Rozdział I. Opis dziedziny przedmiotu

I.1. Opis organizacji

Rys. 1. Logo Związku Harcerstwa Polskiego

„Związek Harcerstwa Polskiego (ZHP) – największa polska organizacja harcerska. Powstała na zjeździe 1–2 listopada 1918 z połączenia wszystkich wcześniej działających organizacji harcerskich i skautowych (scalenie organizacji z trzech zaborów przebiegało w trudnym okresie formowania się II Rzeczypospolitej i zakończyło pod koniec 1920). Jest wychowawczym, patriotycznym, dobrowolnym i samorządnym stowarzyszeniem, otwartym dla wszystkich bez względu na pochodzenie, rasę czy wyznanie. W 2022 roku członkami ZHP było 136 626 osób.” [1]

„Misją Związku Harcerstwa Polskiego jest wychowywanie młodzieży oraz wspieranie rozwoju i kształtowanie charakteru człowieka poprzez stawianie wyzwań. Podstawowe wartości, na których opiera się wychowanie zawarte są w prawie harcerskim.” [1]

Rys. 2. Struktura Związku Harcerstwa Polskiego

Na podstawie strony samej organizacji [2] Związek Harcerstwa Polskiego posiada zhierarchizowaną strukturę opartą na powyższym schemacie (odpowiednio od dołu reprezentują one odpowiednik zespołu zadaniowego, grupę zespołów działającą najczęściej

przy szkole, osiedle mieszkaniowe / miejscowość, dzielnicę / powiat, województwo lub Warszawę – chorągiew Stołeczna).

Z perspektywy państwa ciałami prawnymi są Chorągwie, to one odpowiadają za zadania administracyjne organizacji. Dodatkowo, obowiązki prawne posiadają również wszyscy instruktorzy, niezależnie od szczebla, na jakim działają. Podpisują oni umowę wolontariacką oraz klauzulę RODO.

Struktura organizacji będzie więc tutaj niezbędna przy wdrażaniu całego projektu do użytkowania, ponieważ w ramach różnych elementów struktury będą różne zadania do wykonania, czy to w zakresie promowania serwisu, czy jego dalszego rozwijania.

I.2. Rozwiązania informatyczne zastosowane w organizacji

W celu stałego rozwoju organizacji, jej władze naczelne regularnie wprowadzają w życie projekty mające na celu zwiększenie innowacyjności organizacji. Narzędzia do tego stosowane nie są w pełni intuicyjne, przez co użytkownicy potrzebują wsparcia z zewnątrz do ich obsługi. Tymi narzędziami są między innymi:

- Microsoft 365 – narzędzie do sprawnej komunikacji, zbierania dokumentacji oraz planowania działań. Pozwala to na sprawne grupowanie ludzi, zarządzenie dostępem do informacji oraz bezpieczne ich przechowywanie.
- Tipi – oficjalna ewidencja członków wraz z ich historią członkostwa (posiadane odznaczenia, przynależność do jednostek organizacji)
- Harcerski Serwis Szkoleniowy – platforma do nauki podstawowych kompetencji instruktorów, takich jak znajomość RODO czy organizacja obozu (wszyscy instruktorzy organizacji zobowiązani są do podpisania konkretnej umowy)
- Strony na Facebooku – przestrzeń do promocji organizacji oraz jej działań
- Konta bankowe – od niedawna dostępne nawet na poziomie drużyn, pozwalają na łatwiejszy dostęp do środków oraz sprawniejsze zarządzanie budżetem.

I.3. Problemy organizacji związane z zastosowanymi rozwiązaniami informatycznymi

Pomimo licznych korzyści wynikających z implementacji nowoczesnych technologii informatycznych, instruktorzy Związku Harcerstwa Polskiego (ZHP) napotykają na szereg wyzwań, które utrudniają efektywną pracę oraz pełne wykorzystanie dostępnych narzędzi. Problemy te można zgrupować w kilka głównych kategorii:

- **Problemy z Logowaniem i Dostępem do Platformy Microsoft 365**: Wiele zgłoszeń instruktorów dotyczy trudności z logowaniem się do systemu Microsoft 365. Użytkownicy często zgłaszają problemy z zapomnianymi hasłami, błędami autentykacji czy brakiem dostępu do niektórych funkcji. Te kwestie nie tylko

ograniczają dostęp do kluczowych zasobów, ale także powodują frustrację i opóźnienia w realizacji codziennych zadań.

- **Niewystarczająca Ochrona Danych Osobowych**: Instruktorzy, przechowując dane wrażliwe na prywatnych komputerach i dyskach wirtualnych, nieświadomie narażają organizację na ryzyko naruszeń zasad RODO. Jest to szczególnie niepokojące w świetle coraz większej świadomości dotyczącej prywatności i bezpieczeństwa danych.
- **Brak Umiejętności Podstawowej Obsługi Komputera**: Niektórzy użytkownicy, szczególnie ci nowi lub mniej zaawansowani technologicznie, wykazują braki w podstawowej obsłudze komputera. Ogranicza to ich zdolność do skutecznego wykorzystania narzędzi informatycznych, co wpływa na ogólną wydajność pracy.
- **Niewystarczające Umiejętności Obsługi Narzędzi Biurowych**: Wiele trudności wynika także z ograniczonej znajomości narzędzi biurowych takich jak Word, Excel, Outlook, czy Teams. Brak wprawy w korzystaniu z tych podstawowych narzędzi informatycznych prowadzi do utrudnień w codziennej pracy, obniża efektywność oraz zaburza wewnętrzną komunikację.
- **Niezawodność i Stabilność Platformy**: Problemy z niestabilnością działania platformy Microsoft 365, takie jak przerywane sesje, wolne ładowanie lub błędy systemowe, dodatkowo komplikują pracę instruktorów, co może prowadzić do utraty ważnych danych lub niemożności wykonania pracy w kluczowych momentach.

Te wyzwania, mimo że mogą wydawać się problemami jednostkowymi, mają szeroki wpływ na pracę całej organizacji. Przekładają się one na zmniejszoną efektywność działań, frustrację wśród pracowników oraz potencjalne ryzyko naruszenia zasad ochrony danych osobowych. Mój projekt jest wyjściem naprzeciw tym wyzwaniom biorąc sobie na cel wsparcie instruktorów oraz wszystkich, którzy z takimi problemami się spotykają.

I.4. Analiza SWOT

Poniżej przedstawiona jest tabela zawierająca analizę SWOT będącą ewaluacją rozwiązań informatycznych w organizacji.

Tab. 1. Analiza SWOT wpływu rozwiązań informatycznych na działania w ZHP

S – mocne strony	W – słabe strony
• Duże bezpieczeństwo danych • Łatwy dostęp na wielu urządzeniach • Możliwość współpracy użytkowników • Automatyzacja istotnych procesów w organizacji	• Problemy z logowaniem do usług Microsoft 365 • Narzędzia nie są dostosowane do osób nie wprawionych w obsługę narzędzi informatycznych

• Możliwość prowadzenia łatwej w przeglądaniu i precyzyjnej dokumentacji	
O – szanse	T - zagrożenia
• Dalszy rozwój pozwala na znaczną eliminację zagrożeń wynikających z nieautoryzowanym dostępem do danych lub ich nieoczekiwaną utratą • Wraz z kolejnymi udogodnieniami instruktorzy mogą mniej skupiać się na zadaniach w zakresie logistki, a bardziej na wartości merytorycznej zajęć przygotowywanych dla harcerzy	• Wielu użytkowników rezygnuje z tych rozwiązań na rzecz mniej bezpiecznych i bardziej ograniczonych, ale takich, z którymi są zaznajomieni • Użytkownicy, którzy nie mają wprawy w podstawowej obsłudze komputera nie radzą sobie z dostępnymi dla nich narzędziami • Użytkownicy nie wiedzą, skąd mogą czerpać wiedzę o tym, jak obsługiwać dostępne im narzędzia

I.5. Charakterystyka procesu, który będzie realizował serwis

Zadaniem serwisu będzie udostępnianie platformy do tworzenia i przeglądania wpisów z poradami pomagającymi w rozwiązaniu podstawowych problemów użytkownika w zakresie wykorzystywania różnych narzędzi dostępnych na komputerze.

Funkcjonalności serwisu będą reagować na wyniki analizy SWOT poprzez wpieranie mocnych stron i szans oraz odpowiadanie na potrzeby wynikające ze słabych stron i zagrożeń. Z tego powodu powinien on działać w następujących kontekstach:

- Zwiększać kompetencje użytkowników w zakresie korzystania z komputera i Internetu, niezależnie od poziomu zaawansowania
- Wychodzić naprzeciw problemom, z jakimi spotyka się użytkownik
- Być łatwy w obsłudze
- Umożliwiać tworzenie oraz przeglądanie wszelkich porad, instrukcji czy wskazówek
- Posiadać porady dla użytkowników o różnym poziomie obeznania z komputerem
- Zachęcać użytkowników do wybierania profesjonalnych i bezpiecznych rozwiązań informatycznych
- Gromadzić wszelkie porady w zakresie obsługi popularnych narzędzi komputerowych
- Być skierowanym do grupy docelowej, którą stanowią za równo członkowie ZHP jak i wszyscy zainteresowani zwiększaniem swoich kompetencji komputerowych.
- Zapewniać walidację zamieszczanych treści pod względem poprawności oraz jakości zamieszczonych porad.
- Dawać przestrzeń do informacji zwrotnej użytkowników w przypadku dowolnych uwag, jakie mogą oni mieć do zamieszczanych treści

Proces wdrażania serwisu będzie obejmował kilka etapów, począwszy od wersji pilotażowej lokalnej, przez udostępnienie projektu do testów manualnych, po pełne wdrożenie i integrację z narzędziami ZHP. Regularnie będą zbierane opinie i wnioski użytkowników, aby dostosowywać serwis do ich potrzeb i oczekiwań.

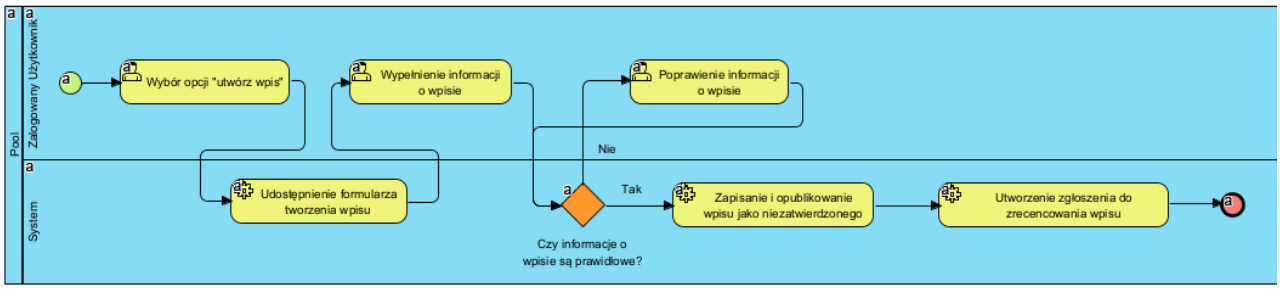
I.6. Diagramy BPMN elementów procesu

„Notacja i model procesu biznesowego (BPMN) to metoda schematu blokowego, która modeluje kroki planowanego procesu biznesowego od końca do końca. Co kluczowe, notacja i model procesu biznesowego wizualnie przedstawia szczegółową sekwencję działań biznesowych i przepływów informacji potrzebnych do ukończenia procesu.” [9]

Poniżej przedstawione są diagramy prezentujące przebieg podstawowych operacji w systemie:

I.6.1. Diagram BPMN dla tworzenia nowego wpisu.

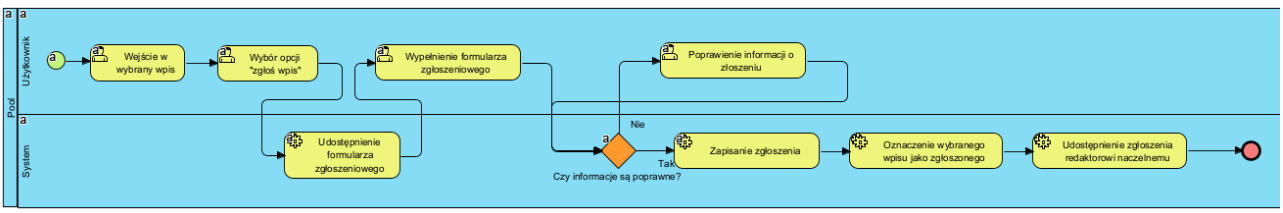
Użytkownik tworzy nowy wpis przez poprawne wypełnienie formularza. Po udanym utworzeniu wpisu zostaje utworzone zgłoszenie informujące o tym, że nowo utworzony wpis należy sprawdzić pod kątem poprawności.



Rys. 3. Diagram BPMN dla tworzenia nowego wpisu.

I.6.2. Diagram BPMN dla zgłaszania wpisu.

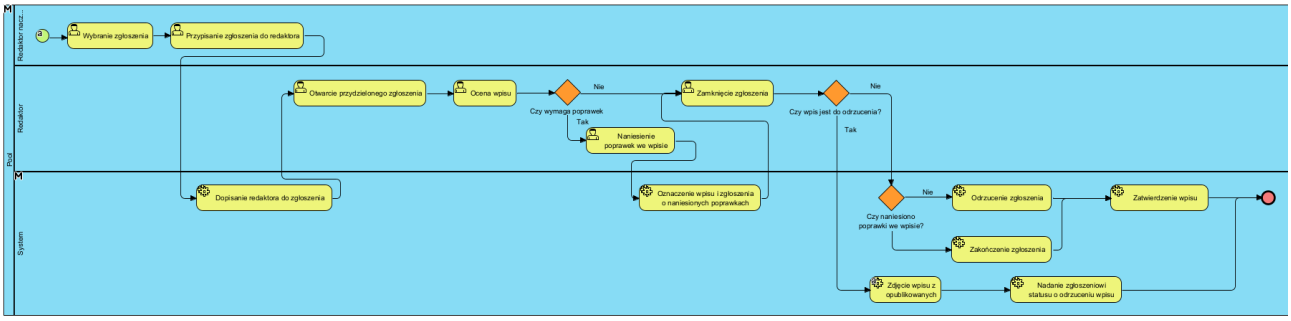
Użytkownik zgłaszając wpis informuje o tym, że należy go oznaczyć jako potencjalnie zawierający błędy, których dotyczy zgłoszenie. Trafia ono do redaktora naczelnego, który może przejrzeć treść zgłoszenia.



Rys. 4. Diagram BPMN dla zgłaszania wpisu.

I.6.3. Diagram BPMN dla rozwiązywania zgłoszenia.

Rozwiązanie zgłoszenia zaczyna się od tego, że redaktor naczelny przydziela je do wybranego redaktora. On następnie może wprowadzić poprawki we wpisie, a następnie zamknąć zgłoszenie poprzez kompletne odrzucenie wpisu (przestaje ono być widoczne dla użytkowników) lub zamknięcie zgłoszenia (które w zależności od naniesionych poprawek oznaczane jest jako zamknięte lub odrzucone).



Rys. 5. Diagram BPMN dla rozwiązywania zgłoszenia.

Rozdział II. Specyfikacja wymagań oraz specyfikacja przypadków użycia

II.1. Wymagania funkcjonalne

„Wymagania funkcjonalne opisują funkcje, możliwości, które w systemie będziemy potrzebować, a system ma realizować. Istotne jest używanie przy określaniu wymagań konkretnych, precyzyjnych, jednoznacznych i dających się następnie zweryfikować stwierdzeń. Na przykład formatowanie tekstu, modulowanie sygnału, formaty zapisów danych, możliwości wydruku, przeglądanie faktur za dany okres.” [4]

Tab. 2. Spis wymagań funkcjonalnych

L.p.	Wymaganie	Warunek	Opis wymagania
1	Rejestracja użytkowników	Obligatoryjne	System powinien umożliwiać rejestrację nowych użytkowników przy pomocy nazwy użytkownika oraz hasła.
2	Logowanie użytkowników	Obligatoryjne	System powinien umożliwiać logowanie zarejestrowanych użytkowników za pomocą nazwy użytkownika oraz hasła.
3	Dostęp dla niezalogowanych	Opcjonalne	System powinien oferować niezalogowanym użytkownikom ograniczony dostęp do wybranych treści i funkcjonalności serwisu, uniemożliwiając dodawanie czy zgłaszanie wpisów.

4	Zarządzanie kontem	Obligatoryjne	System powinien udostępniać wszystkie funkcje CRUD w zakresie zarządzania danymi profilu użytkownika.
5	Zarządzanie autorskimi wpisami	Obligatoryjne	System powinien udostępniać wszystkie funkcje CRUD w zakresie autorskich wpisów użytkownika.
6	Publikowanie wpisów	Obligatoryjne	System powinien umożliwiać publikację autorskiego wpisu użytkownika.
7	Przechowywanie wpisów	Obligatoryjne	System powinien przechowywać wszystkie utworzone wpisy jako niezależne od siebie obiekty wraz z ich datą i godziną utworzenia oraz publikacji.
8	Przeglądanie wpisów	Obligatoryjne	System powinien udostępniać możliwość przeglądania, wyszukiwania i sortowania wpisów wszystkich użytkowników.
9	Zarządzanie uprawnieniami	Obligatoryjne	System powinien umożliwiać administratorowi zarządzanie poziomami uprawnień użytkowników.
10	Zarządzanie statusami	Obligatoryjne	System powinien przyporządkowywać do każdego wpisu i zgłoszenia status odpowiedni dla stanu, w jakim się znajduje.
11	Zgłaszanie błędu we wpisie	Obligatoryjne	System powinien umożliwiać użytkownikowi zgłaszanie uwag lub propozycji poprawek do wpisów, z opcją dołączania szczegółowego opisu problemu i ewentualnych zrzutów ekranu dla lepszego zrozumienia zgłoszenia. Zgłoszenie ma przechowywać datę i godzinę jego przesłania.
12	Kategorie wpisów	Opcjonalne	System powinien oferować system kategoryzowania wpisów poprzez tagi, umożliwiając użytkownikom ich przypisywanie oraz wyszukiwanie wpisów według tych tagów.
13	Zróżnicowana zawartość wpisów	Opcjonalne	System powinien umożliwiać tworzenie wpisów z różną zawartością (tekstem, grafikami) z możliwością jej układania dla poprawy atrakcyjności prezentowanych treści.
14	Rola redaktora	Obligatoryjne	System powinien umożliwiać nadanie zalogowanemu użytkownikowi statusu

			redaktora. Redaktor to zaufany użytkownik z kompetencjami w zakresie obsługi komputera. Otrzymuje on uprawnienia do redagowania wpisów i obsługi zgłoszeń.
15	Redaktor – ocena wpisów	Obligatoryjne	Redaktor powinien mieć możliwość oceniania niezatwierdzonego wpisu przydzielonego przez redaktora naczelnego, w celu jego zatwierdzenia lub odrzucenia.
16	Redaktor – obsługa zgłoszeń	Obligatoryjne	Redaktor powinien mieć możliwość obsługi zgłoszenia przydzielonego przez redaktora w celu podjęcia decyzji o dokonaniu sugerowanych poprawek lub ich odrzucenia.
17	Informacja zwrotna dla użytkownika	Opcjonalne	System powinien podawać informację zwrotną użytkownikowi o decyzji w sprawie zatwierdzenia jego wpisu lub rozpatrzenia zgłoszenia.
18	Redaktor naczelny	Obligatoryjne	System powinien umożliwiać nadanie doświadczonemu redaktorowi roli redaktora naczelnego, który będzie odpowiadał za przydzielanie zgłoszeń do redaktorów. Posiadać tą rolę będzie miała bardzo wąska grupa redaktorów, najbardziej zaufanych, pozwalając im na zarządzanie wszystkimi zgłoszeniami dostępnymi w systemie.

II.2. Wymagania pozafunkcjonalne

„Wymagania pozafunkcjonalne inaczej wymagania dotyczące jakości usług. Zwykle szczegółowe stwierdzenia warunków, w których rozwiązanie musi pozostać skuteczne, cechy, które musi posiadać rozwiązanie, lub ograniczenia, w których musi działać. Przykłady obejmują: niezawodność, łatwość konserwacji, testowalność oraz dostępność.” [4]

Tab. 3. Spis wymagań pozafunkcjonalnych

L.p.	Wymaganie	Opis wymagania
1	Bezpieczeństwo kont	System powinien zapewniać autoryzację i uwierzytelnianie użytkowników zachowujący bezpieczeństwo danych o użytkownikach.

2	Bezpieczeństwo informacji	System powinien zawierać zabezpieczenia przed różnymi atakami, między innymi SQL injection, Dot Dot Slash, XSS czy CSRF.
3	Niezawodność działania	System powinien być dostępny dla użytkowników przez 99% czasu w ciągu roku.
4	Użyteczność interfejsu	Interfejs użytkownika powinien być prosty, intuicyjny i zgodny z zasadami projektowania User Experience. Narzędzia dostępne w danym interfejsie powinny być jednoznacznie opisane.
5	Użyteczność dla różnych przeglądarek	System powinien obsługiwać różne przeglądarki internetowe, między innymi Google Chrome, Microsoft Edge, Firefox.
6	Użyteczność dla różnych urządzeń	Interfejs użytkownika powinien być dostosowany do urządzenia, z którego korzysta użytkownik. Strona powinna być przystępnie wyświetlana zarówno na dużych ekranach komputerów, jak i na mniejszych ekranach tabletu czy smartfonu.
7	Niezawodność działania wielu użytkowników	System powinien być w stanie obsługiwać wielu użytkowników w jednym momencie bez występowania konfliktów w dostępie do funkcjonalności.
8	Zgodność z RODO	System powinien spełniać wymogi normy RODO (Rozporządzenia o Ochronie Danych Osobowych). Dane osobowe, jeśli w ogóle to będą przechowywane w konkretnym celu oraz zabezpieczone przez nieautoryzowanym dostępem.
9	Zgodność ISO	System powinien być zgodny z normami bezpieczeństwa informacji, takimi jak ISO 27001.
10	Wydajność dostępu	System powinien umożliwiać szybki dostęp do danych, z czasem odpowiedzi poniżej 5 sekund.
11	Monitorowanie działania systemu	System powinien posiadać mechanizmy monitorowania i zarządzania zasobami poprzez logi oraz pogląd stanu systemu dla administratora.
12	Odporność na awarie	System powinien być odporny na awarie oprogramowania i sieciowe poprzez przewidywanie i zapobieganie potencjalnych błędów użytkownika.

II.3. Wymagania systemowe

„Wymagania systemowe (ang. system requirements, także wymagania sprzętowe) – minimalne możliwe właściwości systemu komputerowego lub sprzętu niezbędne do uruchomienia danego programu. Najczęściej dotyczą minimalnej prędkości procesora, minimalnej wielkości pamięci RAM, pojemności dysku twardego oraz wersji systemu operacyjnego. Często dotyczą także parametrów karty graficznej.” [6]

Tab. 4. Spis wymagań systemowych

L.p.	Wymaganie	Opis wymagania
1	System operacyjny	System powinien być kompatybilny z systemem operacyjnym Linux, Windows 10 lub nowszym, Android 10 lub nowszym.
2	Przeglądarka	System powinien być zgodny z najnowszymi wersjami przeglądarek internetowych, takich jak Google Chrome, Mozilla Firefox, Safari, Microsoft Edge itp.
3	Język programowania	System powinien być oparty na języku programowania Python wraz z frameworkiem Django.
4	Baza danych	System powinien korzystać z bazy danych umożliwiające przechowywanie wpisów o bardzo różnym formacie i wadze.
5	Rozdzielczość	System powinien być dostosowany do różnych rozdzielczości ekranów, takich jak ekran komputera, tablet, smartfon itp.

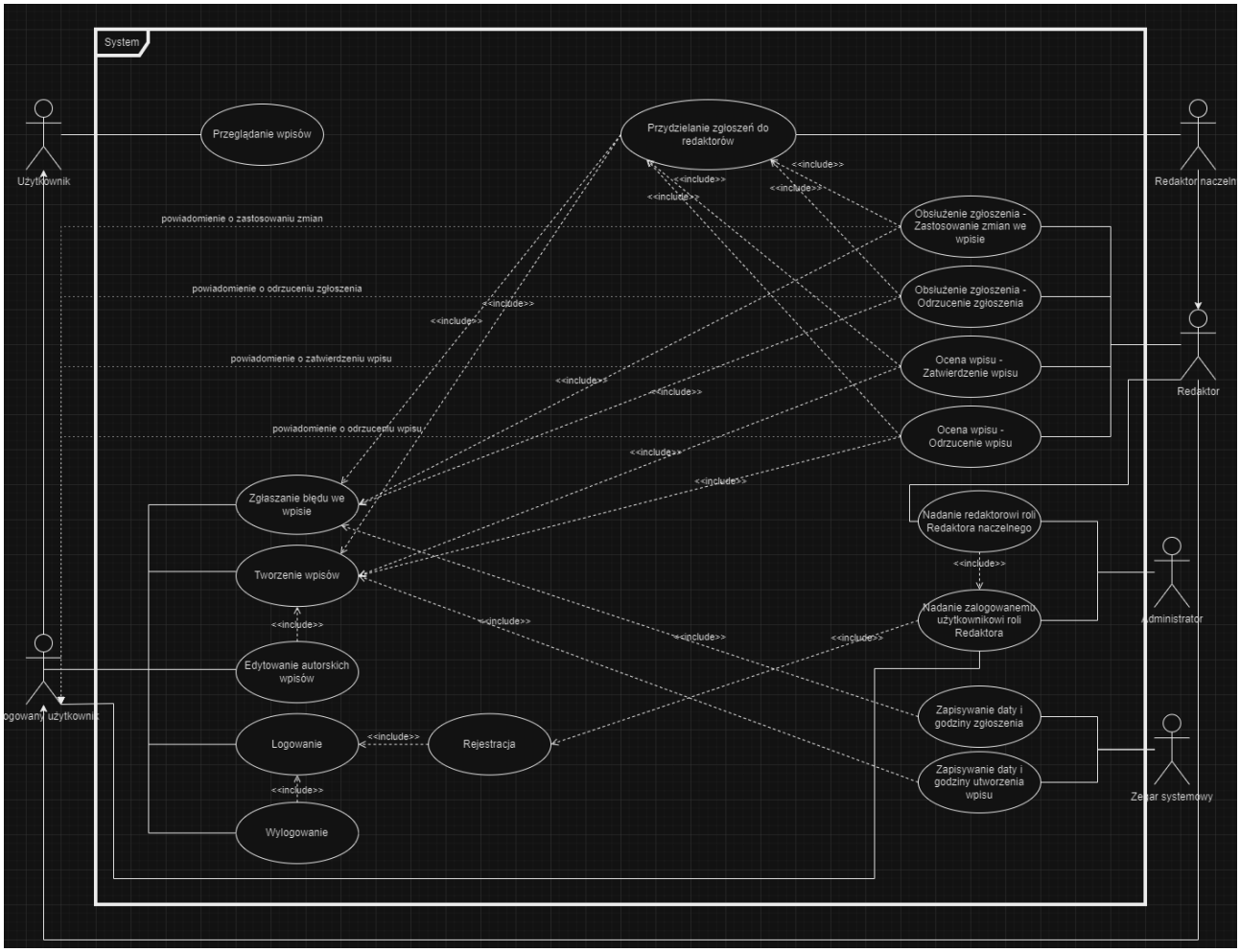
II.4. Diagram przypadków użycia

„Diagram przypadków użycia (ang. use case diagram) jest diagramem, który przedstawia funkcjonalność systemu wraz z jego otoczeniem.

Diagramy przypadków użycia pozwalają na graficzne zaprezentowanie własności systemu tak, jak są one widziane po stronie użytkownika.

Diagramy przypadków użycia służą do zobrazowania usług, które są widoczne z zewnątrz systemu.” [10]

Poniżej przedstawiony jest diagram przypadków użycia projektu:



Rys. 6. Diagram przypadków użycia systemu

II.5. Specyfikacja przypadków użycia

W tym rozdziale przedstawiono tabele z opisem przypadków użycia oraz ich aktorów. Każdy przypadek użycia został oceniony pod względem poziomu skomplikowania przy pomocy metody punktów przypadków użycia (Use Case Points), której klasyfikacja opisana jest w źródle numer [3]. Po specyfikacji przedstawione są przykładowe scenariusze przypadków użycia.

Tab. 5. Opis przypadków użycia z diagramu przypadków użycia UML

L.p.	Przypadek użycia	Opis przypadku użycia	Aktor	UCP
1	Przeglądanie wpisów	Użytkownik ma możliwość przeglądania dostępnych wpisów w systemie między innymi za pomocą fragmentu tytułu czy tagów.	Użytkownik	5
2	Logowanie	Użytkownik może zalogować się do systemu, uzyskując dostęp do dodatkowych jego funkcjonalności.	Zalogowany użytkownik	2
3	Wylogowanie	Zalogowany użytkownik może wylogować się zamykając sesję.	Zalogowany użytkownik	1
4	Rejestracja	Użytkownik może zarejestrować się tworząc nowe konto.	Zalogowany użytkownik	2
5	Tworzenie wpisów	Zalogowany użytkownik może tworzyć nowe wpisy, wprowadzając treść i informacje o krokach instrukcji.	Zalogowany użytkownik	5
6	Edytowanie autorskich wpisów	Zalogowany użytkownik może edytować treść autorskich wpisów.	Zalogowany użytkownik	2
7	Zgłaszanie błędu we wpisie	Zalogowany użytkownik może zgłosić błąd we wpisie, informując o tym redaktorów.	Zalogowany użytkownik	3
8	Przydzielanie zgłoszeń do redaktorów	Redaktor naczelny może przydzielić zgłoszenia dotyczące zmian w wpisach innym redaktorom w celu ich obsługi.	Redaktor naczelny	2
9	Obsłużenie zgłoszenia – zastosowanie zmian we wpisie	Redaktor ma możliwość wprowadzenia zalecanych zmian zgłoszonych w wpisie i zatwierdzić zaktualizowany wpis.	Redaktor	3
10	Obsłużenie zgłoszenia – odrzucenie zgłoszenia	Redaktor może odrzucić zgłoszenie zmian w wpisie i nie nanieść żadnych poprawek.	Redaktor	2

11	Ocena wpisu – zatwierdzenie wpisu	Redaktor może zatwierdzić wpis, uznając go za odpowiedni i gotowy do publikacji.	Redaktor	3
12	Ocena wpisu – odrzucenie wpisu	Redaktor może odrzucić wpis, podając uzasadnienie swojej decyzji.	Redaktor	2
13	Zapisywanie daty i godziny zgłoszenia	Zegar systemowy zapisuje datę i godzinę zgłoszenia konkretnego wpisu lub żądania.	Zegar systemowy	1
14	Zapisywanie daty i godziny utworzenia wpisu	Zegar systemowy rejestruje datę i godzinę publikacji wpisu, co umożliwia śledzenie chronologii treści.	Zegar systemowy	1
15	Nadanie zalogowanemu użytkownikowi roli Redaktora	System umożliwia nadanie zalogowanemu użytkownikowi roli redaktora.	Administrator	1
16	Nadanie redaktorowi roli Redaktora naczelnego	System umożliwia nadanie zalogowanemu użytkownikowi roli redaktora naczelnego.	Administrator	1

Tab. 6. Opis aktorów z diagramu przypadków użycia UML

L.p.	Aktor	Opis aktora
1	Użytkownik	Osoba korzystająca z systemu, która ma możliwość jedynie przeglądania wpisów. Nie ma dostępu do zaawansowanych funkcji, takich jak tworzenie, edytowanie lub publikowanie wpisów.
2	Zalogowany użytkownik	Osoba, która posiada aktywne konto w systemie i zalogowała się. Zalogowany użytkownik ma dostęp do pełnej funkcjonalności systemu, w tym tworzenia, edytowania i publikowania wpisów. Może on korzystać z innych zaawansowanych funkcji, takich jak edytowanie autorskich wpisów czy przeglądanie jego zgłoszeń.
3	Redaktor naczelny	Osoba odpowiedzialna za zarządzanie zgłoszeniami w systemie. Redaktor naczelny ma uprawnienia do przydzielania zgłoszeń do redaktorów, nadzorowania obsługi zgłoszeń oraz podejmowania decyzji dotyczących zatwierdzenia lub odrzucenia wpisów. Ma on wgląd do wszystkich dostępnych w systemie wpisów.

4	Redaktor	Osoba odpowiedzialna za redagowanie i ocenę wpisów zgłaszanych przez użytkowników. Redaktor ma możliwość obsługi zgłoszeń, wprowadzania zalecanych zmian we wpisach oraz podejmowania decyzji dotyczących zatwierdzenia lub odrzucenia wpisów. Pracuje pod nadzorem redaktora naczelnego oraz dba o utrzymanie wysokiego standardu treści w systemie.
5	Zegar systemowy	Komponent systemu odpowiedzialny za śledzenie daty i godziny. Zegar systemowy jest używany do rejestrowania czasu zgłoszenia wpisów, daty publikacji oraz daty i godziny utworzenia wpisów. Zapewnia dokładność danych czasowych w systemie i umożliwia monitorowanie chronologii działań.
6	Administrator	Administrator nadzoruje cały system, ma dostęp do logów, bazy danych oraz wszystkich możliwych funkcjonalności systemu. W procesie odpowiada za nadawanie ról redaktora oraz redaktora naczelnego.

Opis scenariuszy przypadków użycia:

Scenariusz 1. Tworzenie nowego wpisu

- Aktorzy: Zalogowany użytkownik, Zegar systemowy
- Przypadki użycia: Logowanie, Tworzenie wpisów
- Scenariusz:
  1) Użytkownik loguje się do systemu
  2) Zalogowany użytkownik klika opcję tworzenia nowego wpisu
  3) System otwiera formularz tworzenia wpisu
  4) Zalogowany użytkownik wprowadza tytuł, tagi oraz kroki wpisu.
  5) Zalogowany użytkownik zatwierdza formularz wpisu.
  6) Zegar systemowy zapisuje nowy wpis z datą i godziną jego utworzenia.
  7) System tworzy zgłoszenie dla redaktorów w sprawie oceny wpisu.

Scenariusz 2. Zatwierdzanie nowego wpisu

- Aktorzy: Redaktor, Redaktor naczelny, Zegar Systemowy
- Przypadki użycia: Przydzielanie zgłoszeń do redaktorów, Ocena wpisu – zatwierdzenie wpisu
- Scenariusz:
  1) Redaktor naczelny loguje się do systemu.
  2) Redaktor naczelny wchodzi do panelu ze wszystkimi zgłoszeniami.
  3) Redaktor naczelny znajduje nieprzydzielone zgłoszenie.
  4) Redaktor naczelny przypisuje znalezione zgłoszenie do wybranego redaktora.

5) Redaktor naczelny wylogowuje się z systemu.
6) Redaktor loguje się do systemu.
7) Redaktor wchodzi do panelu z przydzielonymi mu zgłoszeniami.
8) Redaktor znajduje nowo przydzielone zgłoszenie
9) Redaktor zatwierdza zgłoszenie.
10) Status spisu, którego dotyczy zgłoszenie zmienia się na zatwierdzony.

Scenariusz 3. Zgłaszanie błędu we wpisie

- Aktorzy: Zalogowany użytkownik
- Przypadki użycia: Zgłaszanie błędu we wpisie, Przeglądanie wpisów
- Scenariusz:
    1) Użytkownik przegląda istniejące wpisy w systemie.
    2) Użytkownik wchodzi w wybrany wpis.
    3) Użytkownik zauważa błąd we wpisie.
    4) Użytkownik klika opcję zgłoszenia błędu przy danym wpisie.
    5) System przenosi użytkownika do formularza zgłoszeniowego.
    6) Użytkownik wprowadza szczegóły dotyczące błędu, opisując problem.
    7) Użytkownik zatwierdza zgłoszenie błędu.
    8) Zegar systemowy zapisuje nowe zgłoszenie z datą i godziną jego utworzenia.

Rozdział III. Projekt aplikacji

III.1. Środowisko wytwórcze

III.1.1. IDE – PyCharm

„PyCharm – zintegrowane środowisko programistyczne (IDE) dla języka programowania Python firmy JetBrains. Zapewnia m.in.: edycję i analizę kodu źródłowego, graficzny debugger, uruchamianie testów jednostkowych, integrację z systemem kontroli wersji. Wspiera także programowanie i tworzenie aplikacji internetowych w Django.” [20]

Dzięki wbudowanym narzędziom oceny jakości kodu, debugowania oraz automatycznego testowania umożliwiona będzie sprawniejsza produkcja kodu.

III.1.2. Przeglądarka – Google Chrome

Google Chrome – „darmowa przeglądarka internetowa rozwijana przez Google. Jej kod został napisany w oparciu o rozwiązania open source częściowo oparte na innych aplikacjach (m.in. WebKit i Mozilla), z wyjątkiem wersji na iOS. Od wydania wersji 28 przeglądarka wykorzystuje silnik Blink. Przeglądarkę tę charakteryzuje oszczędność miejsca, jakie zajmuje interfejs.” [21]

Wbudowane funkcjonalności przeglądarki co do analizy struktury i źródeł strony pozwoli na sprawną analizę rezultatów prac nad projektem.

III.1.3. System Operacyjny – Windows10

„Windows to system operacyjny stworzony przez firmę Microsoft, który jest używany na komputerach osobistych na całym świecie.” [22]

„Windows 10 – wersja systemu operacyjnego Microsoft Windows, która została wydana 29 lipca 2015 roku. Do 29 lipca 2016 roku możliwa była darmowa aktualizacja systemu Windows 7 lub 8.1 do Windowsa 10.” [23]

Ze względu na popularność systemu testowanie aplikacji zapewni dużą kompatybilność z wieloma urządzeniami zarówno w roli klienta, jak i serwera.

III.1.4. System kontroli wersji – git + GitHub

„Git – rozproszony system kontroli wersji. Stworzył go Linus Torvalds jako narzędzie wspomagające rozwój jądra Linux. Git stanowi wolne oprogramowanie i został opublikowany na licencji GNU GPL w wersji 2.” [24]

System kontroli wersji będzie bardzo istotnym narzędziem przy realizacji projektu, przede wszystkim z uwagi na czas tej realizacji.

III.1.5. Platforma chmurowa do wdrożenia – Heroku

„Heroku – platforma chmurowa stworzona w modelu PaaS (Platform as a Service) obsługująca kilka języków programowania. Heroku jest jedną z pierwszych tego typu platform. Rozwijana była od czerwca 2007, kiedy to udostępniała tylko język Ruby. Aktualnie obsługuje języki Java, JavaScript (Node.js), Scala, Clojure, Python i PHP oraz nieudokumentowany Perl. Podstawowym systemem operacyjnym jest Debian lub bazujący na Debianie Ubuntu.” [51]

Ta platforma zapewnia proste i sprawne w zarządzaniu wdrażanie oraz publikację projektu.

III.2. Stos technologiczny

III.2.1. Język – Python

„Python – język programowania wysokiego poziomu ogólnego przeznaczenia, o rozbudowanym pakiecie bibliotek standardowych, którego ideą przewodnią jest czytelność i klarowność kodu źródłowego. Jego składnia cechuje się przejrzystością i zwięzłością.” [27]

Język Python będzie stanowił podstawę implementacji projektu.

III.2.2. Framework - Django

„Django – wolny i otwarty framework przeznaczony do tworzenia aplikacji internetowych, napisany w Pythonie. Powstał pod koniec 2003 roku jako ewolucyjne rozwinięcie aplikacji internetowych, tworzonych przez grupę programistów związanych z Lawrence Journal-World. W 2005 roku kod Django został wydany na licencji BSD. Nazwa frameworku pochodzi od gitarzysty Django Reinhardta.” [28]

Framework Django pozwoli na kompleksową implementację systemu webowego.

III.2.3. Baza danych – PostgreSQL

„PostgreSQL, także Postgres – obok MySQL i SQLite, jeden z najpopularniejszych otwartych systemów zarządzania relacyjnymi bazami danych. Początkowo opracowywany na Uniwersytecie Kalifornijskim w Berkeley i opublikowany pod nazwą Ingres. W miarę rozwoju i zwiększania funkcjonalności, baza danych otrzymała nazwy Postgres95 i ostatecznie PostgreSQL, aby upamiętnić pierwowzór oraz zaznaczyć zgodność ze standardem SQL.” [29]

Wybór bazy danych został oparty o jego duże bezpieczeństwo i przystępność w obsłudze.

III.2.4. Tworzenie szablonów – HTML + CSS

„HTML (z ang. HyperText Markup Language) to hipertekstowy język znaczników, czyli nazwa języka stosowanego do opracowywania dokumentów hipertekstowych. Aktualnie HTML stosujemy przede wszystkim do tworzenia stron internetowych – pozwala on dodać do kodu źródłowego istotne elementy tj.: wyróżnienia, ramki, akapity.” [30]

„CSS to tak zwane kaskadowe arkusze stylów (ang. Cascading Style Sheets). Style służą do opisania wyglądu elementów witryny, zdefiniowanych uprzednio w HTML (np. jakie mają kolory, rozmiary, marginesy, a nawet jak są względem siebie rozmieszczone).” [31]

Narzędzia te zostaną wykorzystane do tworzenia szablonów frameworka Django.

III.2.5. Komunikacja widoków z szablonami – Jinja2

„Jinja2 − silnik szablonów dla języka programowania Python pozwalający na separację logiki aplikacji (Python) od jej warstwy prezentacyjnej (HTML). […] Zasada działania Jinja2 polega na umieszczaniu w plikach źródłowych (np. z rozszerzeniem .html) znaczników, które następnie są zastępowane generowaną przez aplikację treścią. System umożliwia stosowanie struktur kontrolnych (testów (if), pętli (for), itp.).” [33]

Silnik ten stanowić będzie niezbędne dopełnienie szablonów.

III.2.6. Warstwa wizualna szablonów – CSS + Bootstap

„Bootstrap – biblioteka CSS, rozwijana przez programistów Twittera, wydawana na licencji MIT. Zawiera zestaw przydatnych narzędzi ułatwiających tworzenie interfejsu graficznego stron oraz aplikacji internetowych. Bazuje głównie na gotowych rozwiązaniach HTML oraz CSS (kompilowanych z plików Less) i może być stosowany m.in. do stylizacji takich elementów jak teksty, formularze, przyciski, wykresy, nawigacje i innych komponentów wyświetlanych na stronie. Biblioteka korzysta także z języka JavaScript.” [35]

Gotowe komponenty bootstrapa pozwolą szybciej tworzyć warstwę wizualną szablonów.

III.2.7. Konteneryzacja i zarządzanie środowiskiem – Docker

„Docker – otwarte oprogramowanie służące do konteneryzacji, działające jako „platforma dla programistów i administratorów do tworzenia, wdrażania i uruchamiania aplikacji rozproszonych”. Docker jest określany jako narzędzie, które pozwala umieścić program oraz jego zależności (biblioteki, pliki konfiguracyjne, lokalne bazy danych itp.) w lekkim, przenośnym, wirtualnym kontenerze, który można uruchomić na prawie każdym serwerze z systemem opartym na jądrze Linux” [55]

Konteneryzacja projektu zapewni lepsze zarządzanie procesem uruchamiania projektu wraz z całym procesem tworzenia bazy danych czy generowania plików statycznych.

III.2.8. Testy end-to-end – Cypress

„Są to testy, podczas których wcielasz się w użytkownika końcowego i starasz się sprawdzić „ścieżki", jakimi ten użytkownik może przechodzić od początku do końca procesu w aplikacji." [64]

„Cypress jest narzędziem wykorzystywanym do automatyzacji testów. Pozwala automatyzować testy interfejsu użytkownika, umożliwia automatyzowanie testów integracyjnych […] sprawdzając, czy poprawne dane wyświetlają się użytkownikowi na interfejsie graficznym." [65]

Testy end-to-end będą stanowić dopełnienie fazy testowania przez ostateczną weryfikację skutecznego działania szablonów i najważniejszych funkcjonalności systemu.

III.3. Podejście do tworzenia oprogramowania

III.3.1. Podejście zwinne

W ramach realizacji projektu zastosowane zostanie podejście zwinne oraz iteracyjne.

Podejście zwinne w tworzeniu oprogramowania zakłada adaptacyjność i elastyczność wobec zmieniających się wymagań i warunków projektowych.

Manifest zwinności: [11]

1) ludzie i interakcje ponad procesy i narzędzia
2) działające oprogramowanie ponad szczegółową dokumentację
3) współpracę z klientem ponad negocjacje umów
4) reagowanie na zmiany ponad realizację założonego planu

Realizując projekt duży nacisk będzie kładziony na dużą współpracę z potencjalnymi użytkownikami, którzy będą testować system. Dzięki temu będzie można wykryć potencjalne błędy, zebrać opinie dotyczące przejrzystości interfejsu, czy poznać więcej pomysłów na urozmaicenie projektu.

Projekt tworzony będzie w porcjach (iteracjach), tworzone modele, widoki czy szablony będą powstawały po kolei, dzieląc je głównie pod względem realizowanych przypadków użycia.

III.3.2. Test-driven development

„Test Driven Development to technika programowania odwracająca naturalną kolejność. Najpierw piszemy test definiujący wymagania stawiane naszemu programowi, a dopiero potem dopisujemy implementację przechodzącą ten test. Cały cykl dzielimy na krótkie iteracje, gdzie najpierw piszemy jeden test, potem dodajemy kod przechodzący wszystkie testy, a następnie poprawiamy strukturę kodu tak, aby testy dalej przechodziły." [43]

W przypadku tego projektu tworzone będą testy automatyczne sprawdzające dane funkcjonalności pod różnymi względami, określając różne prawdopodobne zachowania użytkownika oraz to, jak system ma na nie reagować. Następnie po implementacji danych funkcjonalności będą uruchamiane testy sprawdzające, czy system działa zgodnie z oczekiwaniami.

III.3.3. Self-documenting code

Tworząc kod wprowadzone zostanie podejście kodowania, które poprzez swoją strukturę będzie łatwe do czytania, analizy i wnioskowania tylko poprzez jego czytanie. Jest do podejście służące poprawy jakości kodu, wspomaga zarządzanie nim oraz ułatwia dalszy jego rozwój.
Zastosowane zostaną zasady samodokumentującego się kodu:

1. Precyzyjne nazewnictwo (struktur, zmiennych, etc.).
2. Niepowielanie kodu, wielokrotne użycie danych elementów.
3. Klarowna struktura komponentów.
4. Nieużywanie instrukcji „break”.
5. Pomijanie stosowania „magicznych liczb” (przypisywanie twardo wpisanych liczb do odpowiednio nazwanych zmiennych).

Zasady te określone są na podstawie źródła numer [45]

III.3.4. Comment-free code

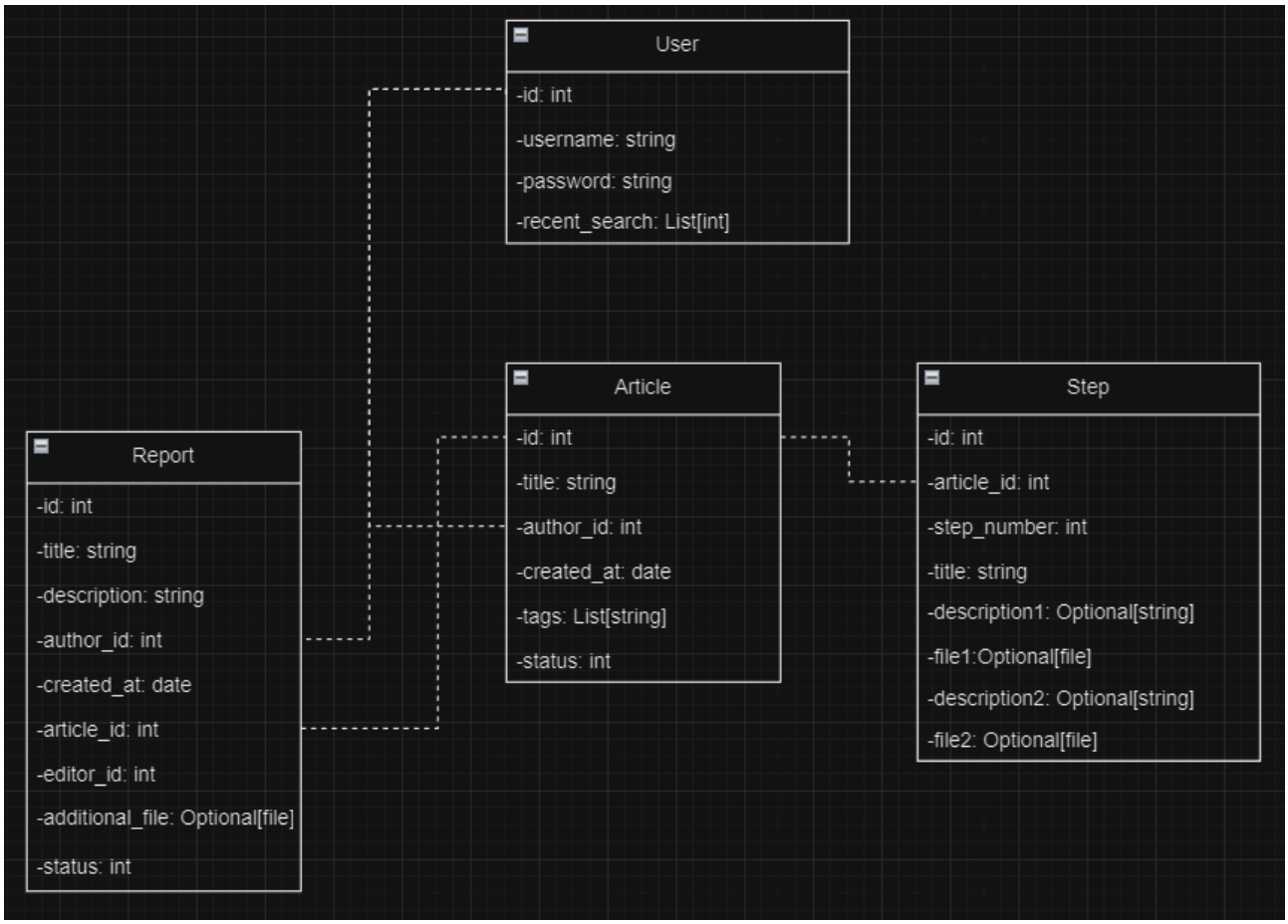
Na podstawie źródła numer [47], cel tego podejścia rozszerza tak naprawdę podejście samodokumentującego kodu. Skupia się minimalizacji komentarzy w taki sposób, żeby wpłynęło to pozytywnie na poprawę przejrzystości kodu. Pomimo tego, że komentarze służą do opisu tego, co programujemy, to są sytuacje, w którym są one sprzeczne z tym jak program działa. Dlatego zaleca się ich pomijanie, a w ramach rekompensaty dokonać takich zmian w kodzie, żeby pozwalały na wywnioskowanie samemu jak to działa.

Sztandarowym przykładem jest stosowanie wartości odmierzających czas. Bardzo często sekunda ma wartość 1, więc minuta 60. Zatem jeśli chcemy w kodzie zastosować wartość 3 minut, to zamiast pisać wartość 180 można po pierwsze przypisać ją do zmiennej o nazwie „THREE_MINUTES”, a wartość zapisać jako „3 * 60”.

Dzięki podejściom takim jak na powyższym przykładzie można uniknąć niejasności mogących wyniknąć z różnic między kodem, a dotyczącym jego komentarzem. Warto jednak dodać, że podejście to dopuszcza stosowanie komentarzy w niektórych miejscach (na przykład przy złożonych operacjach matematycznych).

III.4. Diagram klas

Poniższy rysunek zawiera diagram klas UML, zawierający informacje o tym jakie obiekty będą zawierały się w systemie oraz jakie są dodatkowe zaimplementowane mechanizmy służące optymalizacji kodu i przepływu danych.



Rys. 7. Diagram klas

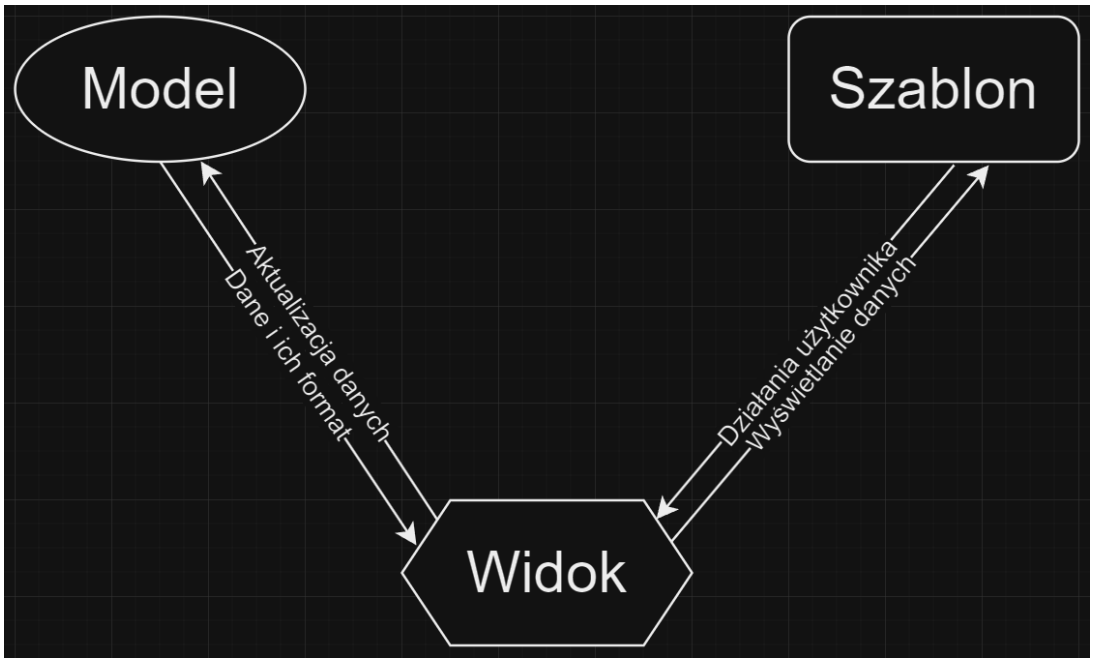
III.5. Architektura

W ramach projektu zastosowana zostanie architektura Model-Widok-Szablon (Model-View-Template, MVT) przy użyciu frameworka Django. Na podstawie źródeł [15], [16] oraz [17], jest to wzorzec architektury stosowany do tworzenia aplikacji internetowych będący wariacją popularnego wzorca Model-View-Controller (MVC). Jego atutami są duża modularność, elastyczność oraz wyraziste oddzielenie logiki biznesowej od interfejsu użytkownika.

Opis komponentów architektury MVT:

- Model – reprezentacja i zarządzanie danymi, definiowanie struktury bazy danych.
- Widok – obsługa logiki aplikacji, przetwarzanie żądań http.
- Szablon – warstwa prezentacji, wygląd strony internetowej, pliki html z dynamicznie generowanymi elementami (poprzez technologię jinja2).

Schemat architektury przedstawiony jest na poniższym rysunku:



Rys. 8. Schemat architektury klient-serwer systemu

Uzasadnienie decyzji:

Architektura MVT sprawdza się bardzo dobrze przy tworzeniu systemów webowych. Poprzez oddzielne tworzenie warstw aplikacji zapewniona jest większa czytelność kodu oraz łatwiejsza skalowalność aplikacji. Mając dobrze zarządzoną strukturę wiadomo dokładnie gdzie znajdują się konkretne elementy architektury oraz gdzie zamieszczać nowe.

Dużym atutem dla przejrzystości są modele, które reprezentują strukturę i sposób przechowywania danych, a wykorzystując mapowanie obiektowo-relacyjne (ORM) można znacznie uprościć implementację komunikacji aplikacji z bazą danych. Zaletą jest również wykorzystanie widoków i szablonów, oddzielających logikę od sposobu jej wyświetlania.

Cały system jest również łatwy w testowaniu z możliwością niezależnego testowania różnych elementów systemu. Można stosować zarówno testy jednostkowe jak i integracyjne, testując niezależnie aplikacje Django czy elementy architektury MVT.

MVT ma również bardzo duże znaczenie jeśli chodzi o bezpieczeństwo danych. Izolacja danych od logiki pozwala na efektywne zarządzenie bezpieczeństwem danych i upraszcza walidację dostępu do nich.

Podsumowując, dobór architektury w połączeniu ze stosem technologicznym określonym we wcześniejszym rozdziale daje bardzo dużo korzyści zarówno pod względem tworzenia jakościowego systemu (bezpiecznego, niezawodnego) jak i pod względem wygody realizacji projektu (elastyczność, czytelność, łatwa testowalność).

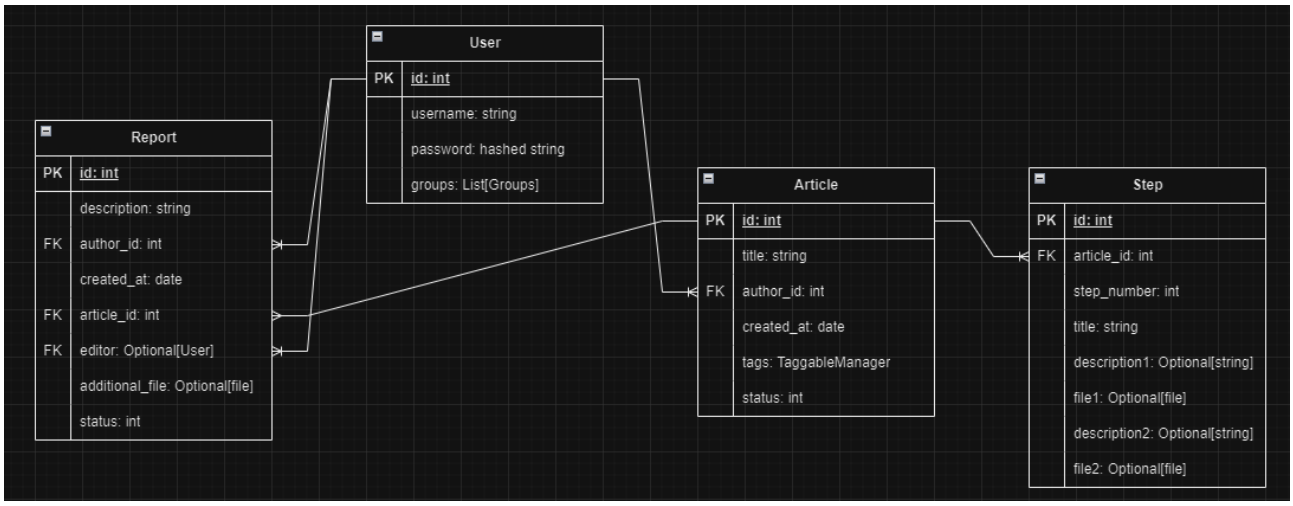
III.6. Projekt bazy danych

W ramach projektu zastosowana będzie relacyjna baza danych PostgreSQL. Przechowywać ona będzie wszystkie najważniejsze informacje o użytkowniku, wpisach czy zgłoszeniach.

PostgreSQL bardzo dobrze łączy się z frameworkiem Django. Wykorzystując silnik „django.db.backends.postgresql” można w łatwy sposób stworzyć połączenie z bazą danych w projekcie, a dzięki mapowaniu relacyjno-obiektowemu (ORM) przeglądanie i zarządzanie danymi jest proste i wydajne.

III.6.1. Schemat bazy danych

Poniższy rysunek przedstawia schemat relacyjny bazy danych:



Rys. 9. Schemat bazy danych

III.6.2. Opis bazy danych

Poniżej opisana jest każda tabela bazodanowa oraz ich atrybuty.

User (użytkownicy):

- id: identyfikator użytkownika
- username: nazwa użytkownika
- password: hasło użytkownika (przechowywane w postaci haszu)

Article (do wpisów):

- id: identyfikator wpisu
- title: tytuł wpisu
- author_id: identyfikator autora wpisu (klucz obcy do tabeli „User”)
- created_at: data utworzenia wpisu
- tags: lista tagów przypisanych do wpisu
- status: numer odpowiedniego statusu wpisu mówiący o tym na jakim etapie weryfikacji się znajduje (sposób numeracji wyjaśniony jest w dalszym opracowaniu)

Step (kroki w artykułach):

- id: identyfikator kroku
- article_id: identyfikator artykułu dla kroku (klucz obcy do tabeli „Article”)
- step_number: numer porządkowy kroku
- title: nazwa kroku
- description1: ewentualny opis kroku
- file1: ewentualne zdjęcie uzupełniające opis kroku
- description2: ewentualny opis kroku
- file2: drugie ewentualne zdjęcie uzupełniające opis kroku

Report (do zgłoszeń):

- id: identyfikator zgłoszenia
- description: opis zgłoszenia
- author_id: identyfikator autora zgłoszenia (klucz obcy do tabeli „User”)
- article_id: identyfikator zgłaszanego wpisu (klucz obcy do tabeli „Article”)
- editor_id: identyfikator redaktora obsługującego zgłoszenie (klucz obcy do tabeli „User”)
- additional_file: ewentualny plik ze zrzutem ekranu uzupełniający opis
- status: numer odpowiedniego statusu wpisu mówiący o tym na jakim etapie weryfikacji się znajduje (sposób numeracji wyjaśniony jest w dalszym opracowaniu)

III.7. Projekt interfejsu oraz uprawnienia

Ze względu na wskazaną wcześniej grupę docelową projektu interfejs graficzny musi być prosty w obsłudze oraz intuicyjny. Projekt uwzględniać musi zarówno aspekt estetyki i czytelności aplikacji, jak i wszelkie potrzeby co do funkcjonalności. Poniżej zestawione są w tabeli szablony wraz z ich zawartością oraz informacjami co do uprawnień dostępu dla każdej roli.

Tab. 7. Wykaz uprawnień do elementów szablonu dla każdego rodzaju użytkownika

L.p.	Szablon	Element	Niez. użytkownik	Zal. użytkownik	Redaktor	Redaktor naczelny
1	**Base**	**Główny dostęp**	**Tak**	**Tak**	**Tak**	**Tak**
2	Base	Navbar - home	Tak	Tak	Tak	Tak
3	Base	Navbar - search	Tak	Tak	Tak	Tak
4	Base	Navbar - user panel	Nie	Tak	Tak	Tak
5	Base	Navbar - editor panel	Nie	Nie	Tak	Tak
6	Base	Navbar - master editor panel	Nie	Nie	Nie	Tak
7	Base	Navbar - register	Tak	Nie	Nie	Nie
8	Base	Navbar - login	Tak	Nie	Nie	Nie
9	Base	Navbar - logout	Nie	Tak	Tak	Tak
10	Base	Stopka	Tak	Tak	Tak	Tak
11	**Home**	**Główny dostęp**	**Tak**	**Tak**	**Tak**	**Tak**
12	Home	Przekierowanie do tworzenia wpisu	Nie	Tak	Tak	Tak
13	Home	Przekierowanie do wyszukiwarki	Tak	Tak	Tak	Tak
14	Home	Przekierowanie do panelu użytkownika	Nie	Tak	Tak	Tak
15	Home	Przekierowanie do panelu redaktora	Nie	Nie	Tak	Tak
16	Home	Przekierowanie do panelu redaktora naczelnego	Nie	Nie	Nie	Tak
17	**Search**	**Główny dostęp**	**Tak**	**Tak**	**Tak**	**Tak**
18	Search	Szukaj po tytule	Tak	Tak	Tak	Tak
19	Search	Szukaj po frazie	Tak	Tak	Tak	Tak
20	Search	Szukaj po tagach	Tak	Tak	Tak	Tak

21	Search	Wyszukaj autorskie	Nie	Tak	Tak	Tak
22	Search	Przycisk "View" dla każdego wyniku wyszukania	Tak	Tak	Tak	Tak
23	**View**	**Główny dostęp**	**Tak**	**Tak**	**Tak**	**Tak**
24	View	Zawartość artykułu wraz z krokami	Tak	Tak	Tak	Tak
25	View	Przycisk edycji wpisu	Nie	Tylko jeśli jest autorem wpisu	Jeśli jest autorem wpisu lub posiada raport na temat tego wpisu	Jeśli jest autorem wpisu lub posiada raport na temat tego wpisu
26	View	Przycisk zgłoszenia wpisu	Nie	Tak	Tak	Tak
27	**Edit**	**Główny dostęp**	**Nie**	**Tylko jeśli jest autorem wpisu**	**Jeśli jest autorem wpisu lub posiada raport na temat tego wpisu**	**Jeśli jest autorem wpisu lub posiada raport na temat tego wpisu**
28	Edit	Formularz zmieniający wartości	Nie	Tylko jeśli jest autorem wpisu	Jeśli jest autorem wpisu lub posiada raport na temat tego wpisu	Jeśli jest autorem wpisu lub posiada raport na temat tego wpisu
29	Edit	Przyciski konfiguracji liczby kroków	Nie	Tylko jeśli jest autorem wpisu	Jeśli jest autorem wpisu lub posiada raport na temat tego wpisu	Jeśli jest autorem wpisu lub posiada raport na temat tego wpisu
30	**Create**	**Główny dostęp**	**Nie**	**Tak**	**Tak**	**Tak**
31	Create	Formularz tworzący wpis	Nie	Tak	Tak	Tak
32	Create	Przyciski konfiguracji liczby kroków	Nie	Tak	Tak	Tak
33	**Report Article**	**Główny dostęp**	**Nie**	**Tak**	**Tak**	**Tak**
34	Report Article	Formularz tworzący raport	Nie	Tak	Tak	Tak
35	Report Article	Podgląd wpisu	Nie	Tak	Tak	Tak
36	**View Report**	**Główny dostęp**	**Nie**	**Tylko jeśli jest autorem wpisu**	**Tylko jeśli jest autorem wpisu**	**Tylko jeśli jest autorem wpisu**
37	View Report	Podgląd raportu	Nie	Tylko jeśli jest autorem wpisu	Tylko jeśli jest autorem wpisu	Tylko jeśli jest autorem wpisu
38	View Report	Przycisk do poglądu wpisu	Nie	Tylko jeśli jest	Tylko jeśli jest autorem wpisu	Tylko jeśli jest autorem wpisu

				autorem wpisu		
39	**User panel**	**Główny dostęp**	**Nie**	**Tak**	**Tak**	**Tak**
40	User panel	Lista autorskich wpisów	Nie	Tak	Tak	Tak
41	User panel	Przycisk podglądu wpisu	Nie	Tak	Tak	Tak
42	User panel	Lista własnych zgłoszeń ze statusami	Nie	Tak	Tak	Tak
43	User panel	Przycisk podglądu zgłoszenia	Nie	Tak	Tak	Tak
44	**Editor panel**	**Główny dostęp**	**Nie**	**Nie**	**Tak**	**Tak**
45	Editor panel	Lista przydzielonych zgłoszeń ze statusami	Nie	Nie	Tak	Tak
46	Editor panel	Przycisk do zarządzania zgłoszeniem	Nie	Nie	Tak	Tak
47	Editor panel	Filtrowanie zgłoszeń po statusach i autorstwie	Nie	Nie	Nie	Tak
48	**Master Editor Panel**	**Główny dostęp**	**Nie**	**Nie**	**Nie**	**Tak**
45	Master Editor Panel	Lista wszystkich zgłoszeń	Nie	Nie	Nie	Tak
49	Master Editor Panel	Przycisk do zarządzania zgłoszeniem	Nie	Nie	Nie	Tak
50	Master Editor Panel	Filtrowanie zgłoszeń po statusach i autorstwie	Nie	Nie	Nie	Tak
51	**Manage Report**	**Główny dostęp**	**Nie**	**Nie**	**Tylko jeśli jest on mu przydzielony**	**Tak**
52	Manage Report	Podgląd raportu	Nie	Nie	Tylko jeśli jest on mu przydzielony	Tak
53	Manage Report	Formularz przypisania redaktora	Nie	Nie	Nie	Tak
54	Manage Report	Przycisk podglądu wpisu	Nie	Nie	Tylko jeśli jest on mu przydzielony	Tak
55	Manage Report	Przycisk odrzucenia wpisu	Nie	Nie	Tylko jeśli jest on mu przydzielony	Tak

56	Manage Report	Przycisk zamknięcia zgłoszenia	Nie	Nie	Tylko jeśli jest on mu przydzielony	Tak

III.8. Cykl życia statusów

Wpisy oraz zgłoszenia mają zamkniętą listę statusów, jakie mogą posiadać. Poniżej przedstawiam to jak w zależności od sytuacji w systemie zmieniają się statusy.

Tab. 8. Cykl życia statusów przy utworzeniu nowego wpisu

L.p.	Sytuacja	Zgłoszenie	Wpis	Widoczność artykułu
1	Stworzono artykuł	na opened	unapproved	Tak
2	Redaktor N. przypisuje zgłoszenie	na assigned	unapproved	Tak
3	(opcjonalnie) Redaktor nanosi poprawki	na changes applied	changes during report	tak
4	Redaktor odrzuca wpis	article rejected	rejected	Nie
5	Redaktor zamyka zgłoszenie	concluded	approved	Tak

Tab. 9. Cykl życia statusów przy zgłoszeniu już istniejącego wpisu:

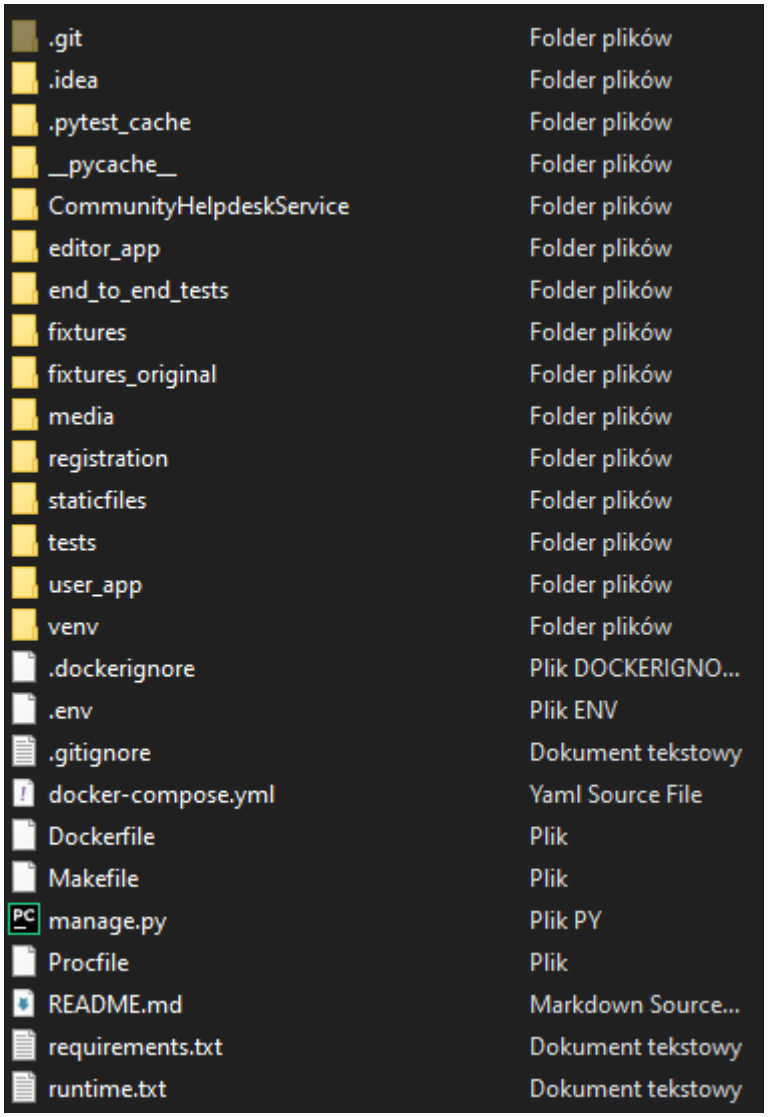
L.p.	Sytuacja	Zgłoszenie	Wpis	Widoczność artykułu
1	Stworzono artykuł	opened	changes requested	Tak
2	Redaktor N. przypisuje zgłoszenie	assigned	changes requested	Tak
3	(opcjonalnie) Redaktor nanosi poprawki	changes applied	changes during report	Tak
4	Redaktor odrzuca wpis	concluded	rejected	Nie
5	Redaktor zamyka zgłoszenie	-	-	-
5.1	Jeśli naniósł zmiany	concluded	approved	Tak
5.2.	Jeśli nie naniósł zmian	rejected	approved	Tak

Rozdział IV. Implementacja

IV.1. Struktura projektu:

IV.1.1. Struktura plików folderu głównego

Poniżej przedstawiona jest struktura plików projektu. Składa się ona ze skryptu zarządzania projektem, folderu głównego projektu, folderów aplikacji frameworka i innych istotnych elementów, które zostaną dalej opisane.

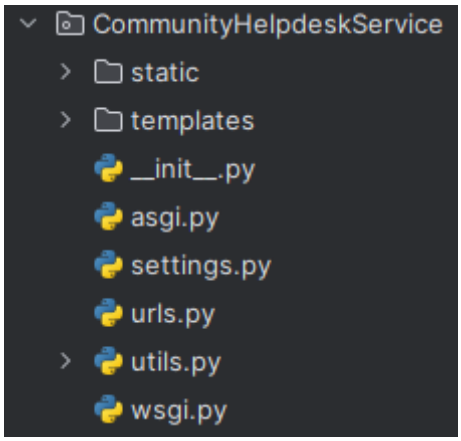


Rys. 10. Struktura plików projektu w głównym folderze

Skrypt „**manage.py**” w frameworku Django pełni rolę narzędzia wiersza poleceń, które umożliwia zarządzanie aplikacjami Django. Jest on lokalnym interfejsem dla django-admin, dostarczając dodatkowe ustawienia specyficzne dla danego projektu. Umożliwia wykonywanie wielu zadań, takich jak migracje bazy danych, tworzenie aplikacji, tworzenie konta administratora, a także testowanie i interakcję z projektem Django (chociażby poprzez możliwość uruchomienia konsoli wewnątrz projektu).

Folder główny projektu "**CommunityHelpdeskService**", utworzony przez Django, służy jako centralny punkt zarządzania i konfiguracji całego projektu. Jego kluczowym elementem jest plik settings.py, który zawiera wszystkie konfiguracje niezbędne do działania aplikacji. Cały folder obejmuje takie ustawienia jak:

- Definicja aplikacji będących częścią projektu
- Informacje o ścieżkach do szablonów, plików i innych niezbędnych dla projektu elementów
- Połączenie z bazą danych wraz z silnikiem i danymi uwierzytelniającymi
- Komunikację serwera z aplikacjami webowymi WSGI (wsgi.py)
- Komunikację asynchroniczną z wieloma użytkownikami ASGI (asgi.py)
- Podstawowe ścieżki projektu wraz z przekierowaniami do aplikacji Django (urls.py)
- Dodatkowe funkcjonalności dla projektu (utils.py)
- Dodatkowe szablony do rozszerzania szablonów dla widoków, między innymi szablon bazowy „base.html" z konfiguracją przeznaczoną dla wszystkich szablonów (foldery css oraz templates).



Rys. 11. Zawartość folderu „CommunityHelpdeskService"

Folder środowiska projektu „**venv**" oraz plik ze zmiennymi środowiskowymi „**.env**" zawierają informacje o lokalnych ustawieniach dla projektu. Przechowują zainstalowane biblioteki, dostosowane do systemu operacyjnego konfiguracje pozwalające na odpowiednie interpretowanie i uruchamianie projektu oraz wszelkie zmienne potrzebne projektowi do działania.

Folder „**.git**" oraz plik „**.gitignore**" konfigurują i przechowują informacje o kontroli wersji, określają jakie pliki są śledzone pod względem zmian i jakie zmiany następowały wraz z rozwojem projektu.

Folder „**media**" stanowi miejsce przechowywania plików wprowadzanych do systemu. Tam są zapisywane i przechowywane, a adresy dostępu do nich są przechowywane w bazie danych. Dodatkowo istnieje folder „**staticfiles**", który służy aplikacji do pobierania zdjęć, skryptów czy plików stylu, które są tam pomocniczo przechowywane.

Foldery „**.pytest_cache**" oraz „**__pycache__**" zawierają dane pomocnicze (np. pliki bajtowe .pyc) dla szybszego uruchomienia testów lub modułów systemu.

Folder „**.idea**” to plik stworzony przez IDE, zawiera konfiguracje dla PyCharma takie jak preferencje formatowania czy ustawienia dotyczące kontroli jakości kodu.

Folder „**tests**” zawiera pakiet testów automatycznych dla całego projektu. Każdy plik to osobna klasa testowa, wszystkie klasy tworzą strukturę dziedziczenia do sprawniejszego tworzenia testów. Folder „**end_to_end_tests**” natomiast zgodnie ze swoją nazwą posiada testy end-to-end napisane przy użyciu narzędzia „Cypress”.

Folder „**fixtures**” przechowuje pliki JSON zawierający bazowe dane wgrywane do systemu przy jego uruchomieniu poprzez dockera.

Pliki „**Dockerfile**”, „**docker-compose.yml**”, „**.dockerignore**” oraz „**requirement.txt**” służą do uruchamiania systemu poprzez konteneryzację.

Pliki „**Procfile**”, „**runtime.txt**” oraz „**requirements**.txt” służą do uruchomienia systemu na platformie Heroku. Plik „**Makefile**” z kolei zawiera instrukcje służące do uruchomienia systemu poprzez konsolę w systemie Windows.

Foldery „**registration**”, „**user_app**” oraz „**editor_app**” to foldery aplikacji implementujących wszystkie niezbędne funkcjonalności projektu i zostaną opisane w kolejnych rozdziałach. To właśnie one zawierają realizację architektury, definiują modele, widoki i szablony oraz w ramach nich we współpracy ze skryptem „urls.py” w folderze „CommunityHelpdeskService” odbywa się mechanizm przechwytywania i przekierowywania do odpowiednich elementów systemu zapytań http.

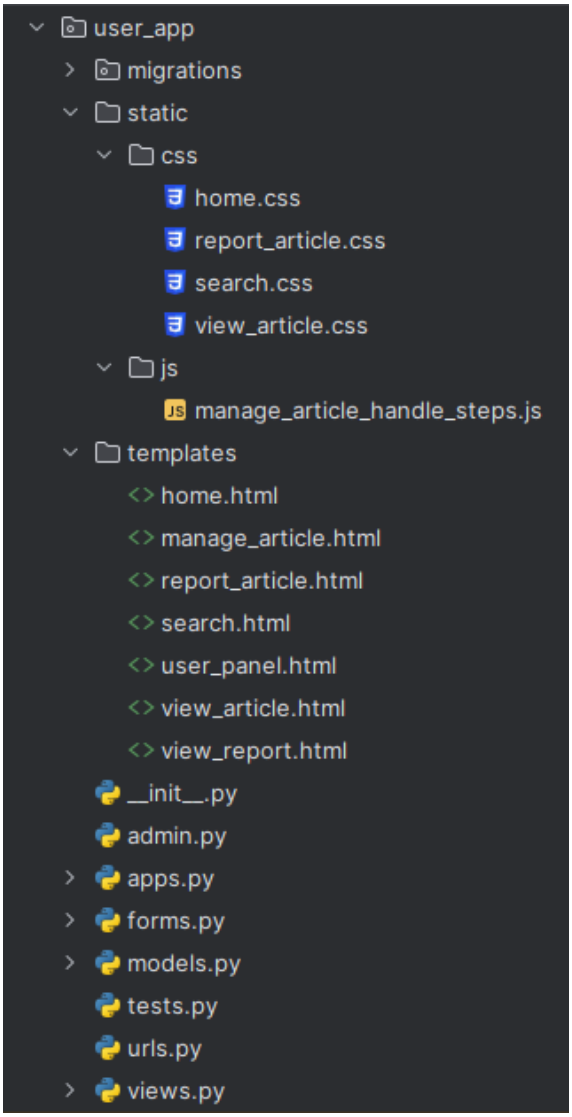
IV.1.2. Struktura zaimplementowanych aplikacji

Wszystkie zaimplementowane aplikacje składają się z dostosowaniem do indywidualnych potrzeb każdej aplikacji z takiej listy zawartości:

- Folder „**__pycache__**”, tak jak w folderze głównym, zawiera dane pomocnicze do szybszego ładowania modułów
- Folder „**migrations**” przechowuje pliki migracji, które są używane do zarządzania zmianami w schemacie bazy danych, pozwalając na kontrolę i ewolucję struktury danych projektu.
- Folder „**static**” zawiera statyczne pliki, takie jak CSS, JavaScript i obrazy, które są wykorzystywane do projektowania i funkcjonalności interfejsu użytkownika.
- Folder „**templates**” przechowuje pliki HTML szablonów, które definiują strukturę i wygląd stron internetowych generowanych przez aplikację.
- Skrypt „**__init__.py**” oznacza folder jako pakiet Pythona, umożliwiając importowanie modułów i pakietów w obrębie aplikacji.
- Skrypt „**admin.py**” definiuje ustawienia dla panelu administracyjnego Django, umożliwiając konfigurację sposobu wyświetlania i zarządzania modelami w interfejsie administratora.
- Skrypt „**apps.py**” zawiera konfigurację danej aplikacji, umożliwiając Django identyfikację jej jako modułu oraz konfigurację specyficznych właściwości aplikacji.

- Skrypt „**forms.py**” definiuje formularze Django, które są wykorzystywane do tworzenia i przetwarzania danych wejściowych od użytkowników. Część z nich tworzona jest na podstawie zdefiniowanych modeli.
- Skrypt „**models.py**” definiuje struktury modeli danych, które są używane do interakcji z bazą danych, reprezentując tabele i relacje między nimi.
- Skrypt „tests**.py**” zawiera testy jednostkowe i integracyjne, które pomagają w zapewnieniu poprawności i stabilności działania aplikacji.
- Skrypt „**urls.py**” określa mapowanie URL-ów na widoki aplikacji, definiując w jaki sposób aplikacja odpowiada na różne żądania HTTP.
- Skrypt „**views.py**” zawiera logikę obsługi widoków, definiując, jakie dane są prezentowane użytkownikowi oraz jak są one przetwarzane.

Przykład struktury aplikacji został zaprezentowany poniżej na podstawie zawartości folderu aplikacji „user_app”:



Rys. 12. Zawartość folderu aplikacji na przykładzie „user_app”

Następnie zaprezentowane są poniżej wszystkie aplikacje wraz z tym jakie implementują przypadki użycia i jakie elementy architektury realizują. Sposób przydzielenia przypadków użycia do danej aplikacji wynika bezpośrednio z nazw oraz przeznaczenia aplikacji.

Aplikacja „registration”

Odpowiada za zarządzanie rejestracją, logowaniem i sesją użytkownika. Wykorzystuje model użytkownika z frameworka Django do zarządzania sesją i odpowiedniego dodawania/usuwania danych logowania.

Zaimplementowane przypadki użycia:

- Logowanie
- Rejestracja
- Wylogowanie

Realizacja architektury:

- Modele: wykorzystuje model użytkownika zaimplementowany w ramach frameworka Django, który realizuje zadania klasy „User” z diagramu klas.
- Widoki: implementuje widoki tworzące nowego użytkownika w przypadku rejestracji, logowania użytkownika przy pomocy mechanizmu uwierzytelniania czy wylogowywania.
- Szablony: Udostępnia pliki html wsparte plikami stylów css udostępniające

Aplikacja „user_app”

Element projektu odpowiedzialny za implementację wszystkich działań związanych z wpisami i zgłoszeniami w zakresie użytkownika. Dostarcza model wpisu, formularze potrzebne do tworzenia oraz wyszukiwania wpisów, widoki z szablonami do wyświetlania i obsługi żądań użytkownika.

Zaimplementowane przypadki użycia:

- Przeglądanie wpisów
- Zgłaszanie błędu we wpisie
- Tworzenie wpisów
- Edytowanie autorskich wpisów
- Zapisywanie daty i godziny zgłoszenia
- Zapisywanie daty i godziny utworzenia wpisu

Realizacja architektury:

- Modele: implementuje 2 modele, jeden dla wpisów, drugi dla kroków wpisu. Klasa dla wpisów przechowuje podstawowe informacje takie jak tytuł czy autora, a kroki przechowują jedynie treść wpisu.

- Widoki: tworzy widok strony głównej wyszukiwarki do wpisów, tworzenia wpisów oraz innych elementów. Obsługuje żądania wyświetlając lub konfigurując zawartość bazy danych.
- Szablony: Udostępnia szablony pozwalające użytkownikom na realizację działań na opisanych widokach, wyświetlając przyjazny dla nich interfejs.

Aplikacja „editor_app”

Aplikacja udostępniająca wszystkie narzędzia dostępne dla redaktorów i redaktorów naczelnych. Służy do zapewniania redaktorom przestrzeni do obsługi zgłoszeń utworzonych przez użytkowników. Widoki tej aplikacji są niedostępne dla zwykłego użytkownika, wymagana jest co najmniej przynależność do redaktorów.

Zaimplementowane przypadki użycia:

- Przydzielanie zgłoszeń do redaktorów
- Obsłużenie zgłoszenia – zastosowanie zmian we wpisie
- Obsłużenie zgłoszenia – odrzucenie zgłoszenia
- Ocena wpisu – zatwierdzenie wpisu
- Ocena wpisu – odrzucenie wpisu

Realizacja architektury:

- Modele: implementuje model zgłoszenia wykorzystywany zarówno przez użytkowników do tworzenia zgłoszeń jak i redaktorów do ich obsługi.
- Widoki: zawiera widoki pozwalające zarówno na przeglądanie, analizowanie jak i obsługiwanie zgłoszeń.
- Szablony: Udostępnia szablony do realizacji zadań widoków, z mechanizmami ułatwiającymi przeglądanie zgłoszeń.

IV.2. Grupy użytkowników:

IV.2.1. Users (Użytkownicy)

Grupa "Users" skupia zalogowanych użytkowników serwisu "CommunityHelpdeskService". Użytkownicy w tej grupie mają dostęp do funkcjonalności w zakresie zarządzania wpisami czy zgłaszania problemów. Ta grupa stanowi podstawę społeczności serwisu i jej członkowie są aktywnymi odbiorcami oraz twórcami treści w ramach serwisu.

IV.2.2. Editors (Redaktorzy)

Grupa "Editors" to użytkownicy pełniący rolę redaktorów w serwisie. Redaktorzy są odpowiedzialni za moderowanie treści wprowadzanych przez użytkowników oraz zarządzanie zgłoszeniami.

IV.2.3. MasterEditors (Redaktorzy Naczelni)

Grupa "MasterEditors" obejmuje redaktorów naczelnych, którzy posiadają rozszerzone uprawnienia w zakresie zarządzania treścią i użytkownikami serwisu. Są oni odpowiedzialni za nadzór nad pracą redaktorów poprzez przydzielanie zgłoszeń.

IV.3. Zarządzanie dostępem

W systemie zastosowane są różne mechanizmy zarządzania dostępem do treści. Opisane są one poniżej:

IV.3.1. Dekorator „login_required”

W projekcie wykorzystany jest dekorator „login_required” z modułu „django.contrib.auth.decorators”. Dla danego widoku sprawdza on, czy użytkownik jest zalogowany i jeśli nie to przekierowuje go do strony logowania

Przykład użycia dekoratora:

```
@login_required
def user_panel_view(request):
    """ all user's articles and reports """

    user_articles = [{
        'article': article, 'steps_amount':
len(Step.objects.filter(article=article)),
    } for article in Article.objects.filter(author=request.user)]
```

W momencie, w którym niezalogowany użytkownik będzie chciał wejść na stronę z widoku „user_panel_view”, zostanie przekierowany na stronę logowania. Przekierowanie to jest zdefiniowane w ustawieniach przy pomocy specjalnej zmiennej:

```
LOGIN_URL = '/registration/login/'
```

IV.3.2. Sprawdzanie przynależności do grupy

Poziom uprawnień w systemie jest bezpośrednio wyrażany przynależnością do danej grupy użytkowników. Więc aby sprawdzić, czy użytkownik jest redaktorem odpowiedniego szczebla to należy sprawdzić, czy należy do odpowiedniej grupy.

Przykład sprawdzania przynależności:

```
is_master_editor = request.user.groups.filter(name='MasterEditors').exists()
```

IV.3.3. Sprawdzanie sesji

Zmienne sesji użytkownika są dostępne dla szablonów, dlatego jest to dobry sposób na przekazywanie informacji. Szablon „base.html” generuje pasek nawigacyjny w oparciu o przynależność użytkownika do grup, dlatego do sesji dodane są flagi „is_editor” oraz „is_master_editor”, dzięki czemu w połączeniu z flagą informującą o autoryzacji można wygenerować pasek nawigacyjny z listą linków dostosowaną do aktualnego poziomu uprawnień użytkownika.

Dodawanie elementów do sesji:

```
is_master_editor = request.user.groups.values_list('name',
flat=True).filter(name='MasterEditors').exists()
request.session['is_master_editor'] = bool(is_master_editor)
is_editor = request.user.groups.values_list('name',
flat=True).filter(name='Editors').exists()
request.session['is_editor'] = bool(is_editor)
```

Poniżej przedstawiona jest implementacja paska nawigacyjnego w szablonie „base.html”, gdzie widać warunki jakie muszą zostać spełnione, żeby dany element paska był dostępny w interfejsie użytkownika. Dzieje się to na podstawie zmiennych dodanych do sesji oraz flagi mówiącej czy użytkownik został uwierzytelniony.

```
{% if user.is_authenticated %}
    <li class='nav-item active'>
        <a class='nav-link' href='/user_app/user_panel'>User Panel</a>
    </li>
{% endif %}
{% if user.is_authenticated and request.session.is_editor %}
    <li class='nav-item active'>
        <a class='nav-link' href='/editor_app/editor_panel'>Editor Panel</a>
    </li>
{% endif %}
{% if user.is_authenticated and request.session.is_master_editor %}
    <li class='nav-item active'>
        <a class='nav-link' href='/editor_app/master_editor_panel'>Master Editor
Panel</a>
    </li>
{% endif %}
{% if not user.is_authenticated %}
    <li class='nav-item active'>
        <a class='nav-link' href='/registration/register'>Register</a>
    </li>
{% endif %}
{% if not user.is_authenticated %}
    <li class='nav-item active'>
        <a class='nav-link' href='/registration/login'>Login</a>
    </li>
{% endif %}
{% if user.is_authenticated %}
    <li class='nav-item active'>
        <a class='nav-link' href='/registration/logout'>Logout</a>
    </li>
{% endif %}
```

IV.4. Komunikacja z bazą danych

Poniżej przedstawiona jest lista instrukcji służących do komunikacji z bazą danych w zakresie ich pobierania, aktualizacji czy tworzenia jako przykłady zastosowania. Instrukcje te są bezpośrednio tłumaczone na zapytania SQL i wysyłane do bazy danych.

IV.4.1. Pobranie obiektu o danym id

Pobranie wpisu:

```
article = Article.objects.get(id=report.article.id)
```

Pobranie zgłoszenia:

```
report = Report.objects.get(id=report_id)
```

IV.4.2. Pobranie listy obiektów z filtrowaniem

Pobranie listy zgłoszeń użytkownika o danym id:

```
user_reports = Report.objects.filter(author_id=user_id)
```

Pobranie listy zgłoszeń przypisanych do danego redaktora:

```
editor_reports = Report.objects.filter(editor=editor)
```

Pobieranie listy wpisów z danym zestawem tagów:

```
searched_articles = Article.objects.filter(tags__in=tag_objects)
```

Pobieranie listy wpisów z danym fragmentem tytułu:

```
searched_articles = Article.objects.filter(
    title__icontains=search_text, status__in=search_permitted_statuses)
```

IV.4.3. Sprawdzanie czy obiekt istnieje

```
request.user.groups.filter(name='MasterEditors').exists()
```

IV.4.4. Zliczanie obiektów

```
count_articles = Article.objects.filter(title__contains='Test Article').count()
```

IV.4.5. Sortowanie wyników

```
steps = Step.objects.filter(article=article).order_by('ordinal_number')
```

IV.4.6. Zwracanie unikatowych wyników

```
authors = User.objects.filter(report_author__in=reports).distinct()
```

IV.4.7. Zwracanie unikatowej listy wartości danego atrybutu w liście obiektów:

```
statuses = [{'status_number': n, 'status_name': ReportStatus.get_status_name(n)}
            for n in reports.values_list('status', flat=True).distinct()]
```

IV.4.8. Tworzenie nowego użytkownika na podstawie danych z formularza

```
form = UserCreationForm(request.POST)
if form.is_valid():
    form.save()
    return redirect('login')
```

IV.4.9. Tworzenie nowego wpisu

```
form = ArticleForm(request.POST)
if form.is_valid():
    new_article = form.save(commit=False)
    new_article.author = request.user
    new_article.status = ArticleStatus.UNAPPROVED.n
    new_article.save()
```

IV.4.10. Tworzenie zbioru kroków tworzonego wpisu

```
step_form_set = StepFormSetCreate(request.POST, request.FILES)
if step_form_set.is_valid():
    save_files(request.FILES)
    ordinal_number = 1
    for step_form in step_form_set:
        step = step_form.save(commit=False)
        step.article = new_article
        step.ordinal_number = ordinal_number
        step.save()
        ordinal_number += 1
```

IV.4.11. Tworzenie nowego zgłoszenia

```
report_form = ReportForm(request.POST, request.FILES)
if report_form.is_valid():
    save_files(request.FILES)
    report = report_form.save(commit=False)
    report.title = generate_report_title(report.description)
    report.author = request.user
    report.article = article
    report.status = ReportStatus.OPENED.n
    report.save()
```

IV.4.12. Obsługa zmiany lub przydzielania redaktora do zgłoszenia

```
if request.method == 'POST':
    if is_master_editor and 'editor_assign_id' in request.POST:
        report = Report.objects.get(id=request.POST.get('report_id'))
        new_editor = User.objects.get(id=request.POST.get('editor_assign_id'))
        report.editor = new_editor
        report.status += 1  # change from '(na) opened' to '(na) assigned' for
both types of report
        report.save()
        return redirect('home')
```

IV.5. Zapytania HTTP

Zapytania HTTP w projekcie są kluczowym elementem interakcji między klientem (przeglądarką użytkownika) a serwerem, na którym działa aplikacja. Django jako framework webowy, zapewnia rozbudowane narzędzia do obsługi tych zapytań.

IV.5.1. Zapytania GET

Zapytania GET nie posiadają żadnego szyfrowania przekazywanych informacji, a ich zawartość jest widoczna w przeglądarce w ramach adresu URL. W projekcie służą do pobierania danych i do tworzenia przycisków przekierowujących między stronami. W Django, zapytania GET są łatwo dostępne w widokach poprzez obiekt request.GET, który jest podobnym do słownika obiektem zawierającym wszystkie parametry zapytania.

Przykład formularzu tworzącego zapytanie GET w szablonie Django:

```
<form method='get' action='{% url 'user_panel' %}'>
    <button type='submit' class='btn btn-dark form-control form-control-sm'>
        Go to user panel
    </button>
</form>
```

Obsługa tego zapytania dzieje się w ramach funkcjonalności frameworka Django, który przy użyciu systemu routingu zdefiniowanego w skryptach „urls.py” mapuje zapytanie na odpowiedni widok.

IV.5.2. Zapytania POST

Zapytania POST są bardziej złożone, gdyż dane przekazywane tym zapytaniem są szyfrowane i przesyłane w sposób niejawny. W projekcie służą głównie do przesyłania danych z formularzy. W Django dostęp do danych POST odbywa się poprzez obiekt request.POST, który podobnie jak request.GET, jest słownikiem lub podobnym do słownika obiektem.

Przykład formularza tworzącego zapytanie POST dla logowania:

```
<form action='#' method='post' class='form-group'>
    {% csrf_token %}
    {{ form.as_p }}
    <button type='submit' class='btn btn-dark
registration_button'>Login</button>
</form>
```

Przykład obsługi zapytania POST w widoku dla logowania:

```
if request.method == 'POST':
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(request, username=username, password=password)
```

IV.5.3. Przekazywanie plików

Framework Django w zapytaniach http przekazuje dwie struktury: „request.POST” jak i „request.FILES”. Oby można było przekazać tą drugą, formularzowi dodaje się instrukcję „enctype='multipart/form-data'”.

Jeśli chodzi o zapisywanie plików, znajdują się one pod ścieżką określoną w ustawieniach, a baza danych przechowuje nazwę pliku oraz wspomnianą ścieżkę

Przedstawiony poniżej formularz dla zgłaszania błędu we wpisie zawiera załączanie pliku:

```
<form class='report_form' method='post' enctype='multipart/form-data'>
    {% csrf_token %}
    {{ report_form.as_p }}
    <button type='submit' class='btn btn-dark image_button submit_report_i_b
let_hover' name='submit_report'>
        <img src='{% static 'img/submit_report_icon.png' %}'
alt='submit_report'/>
        <span class='hover_text'>submit report</span>
    </button>
</form>
```

Obsługa formularza z plikiem z zapisywaniem pliku do systemu dla zgłaszania błędu we wpisie:

```
report_form = ReportForm(request.POST, request.FILES)
if report_form.is_valid():
    save_files(request.FILES)
    report = report_form.save(commit=False)
    report.title = generate_report_title(report.description)
    report.author = request.user
    report.article = article
    report.status = ReportStatus.OPENED.n
    report.save()
```

Wywołana jest tutaj funkcja zapisująca plik do systemu, jej implementacja wygląda następująco:

```
def save_files(requested_files):
    """ file saving function """
    fs = FileSystemStorage()
    for file_name in requested_files:
```

```
        file = requested_files[file_name]
        fs.save(file.name, file)
```

Wykorzystana jest klasa „FileSystemStorage” z biblioteki „django.core.files.storage”, to właśnie za jej pomocą zapisywane są pliki poprzez metodę „save()” podając nazwę pliku oraz sam plik. Miejsce zapisu pliku definiuje zmienna z ustawień w skrypcie „settings.py”:

```
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
MEDIA_URL = '/media/'
```

IV.6. Uwierzytelnianie, autoryzacja oraz rejestracja

„Uwierzytelnianie to potwierdzenie, że jesteśmy tym, za kogo się podajemy, czyli inaczej to ujmując, potwierdzenie swojej tożsamości. Najczęściej możemy je spotkać podczas logowania” [37]

„Autoryzacja to proces określania uprawnień danego podmiotu. Inaczej mówiąc autoryzacja pozwala na stwierdzenie czy dany podmiot (np. osoba) posiada dostęp do danego zasobu” [37]

IV.6.1. Uwierzytelnianie

Uwierzytelnianie odbywa się poprzez widok „login_view” w aplikacji „registration”. Użytkownik poprzez ten widok ma udostępniany formularz logowania, po którego poprawnym wypełnieniu wywoływana jest instrukcja z frameworka Django do dokowania uwierzytelniania na podstawie wprowadzonych danych. Po udanym uwierzytelnieniu następuje zalogowanie użytkownika przez dodanie jego unikalnego klucza do sesji (przechowywanego pod nazwą „SESSION_KEY”). Oprócz tego dodawane są zmienne „BACKEND_SESSION_KEY” (aplikacja backendu używanego do uwierzytelniania) oraz „HASH_SESSION_KEY” (unikalny hash służący do zachowania integralności danych sesji oraz zabezpieczający przed nieautoryzowaną manipulacją identyfikatora sesji).

Widok login_view z instrukcjami „authenticate” oraz login (z biblioteki “django.contrib.auth”):

```
def login_view(request):
    """ login user to session """
    if request.method == 'POST':
        user = authenticate(request, username=request.POST['username'],
password=request.POST['password'])
        if user is not None:
            login(request, user)
            request.session['is_master_editor'] =
request.user.groups.filter(name='MasterEditors').exists()
            request.session['is_editor'] =
request.user.groups.filter(name='Editors').exists()
            return redirect(request.GET.get('next', 'home'))

    form = AuthenticationForm()
    return render(request, 'login.html', {'form': form})
```

Implementacja logowania w bibliotece Django:

```
def login(request, user, backend=None):
    # […]
    request.session[SESSION_KEY] = user._meta.pk.value_to_string(user)
    request.session[BACKEND_SESSION_KEY] = backend
    request.session[HASH_SESSION_KEY] = session_auth_hash
```

Fragment skopiowany bezpośrednio z zainstalowanej biblioteki, można go znaleźć w internecie w repozytorium github [68].

Po udanym zalogowaniu użyte jest przekierowanie do wartości „next”. Jest to wykorzystanie funkcjonalności frameworka Django, który przy próbie wejścia na stronę wymagającą bycia zalogowanym przez nieuwierzytelnionego użytkownika przekieruje go na stronę logowania zapamiętując adres strony, do której chciano się dostać w pierwszej kolejności. Tak wygląda adres url przy takim przekierowaniu:

```
http://127.0.0.1:8000/registration/login/?next=/user_app/create_article/
```

IV.6.2. Autoryzacja

Autoryzacja odbywa się w wielu miejscach systemu. W projekcie dzieje się to przy weryfikacji dostępu do danych widoków i szablonów. Istnieją 2 najistotniejsze zastosowania autoryzacji:

Sprawdzenie czy użytkownik jest zalogowany

```
@login_required
def create_article_view(request):
```

Przedstawiona powyżej deklaracja widoku zawiera dekorator „@login_required” określająca, że tylko zalogowany użytkownik ma dostęp do tego widoku. Jeśli niezalogowany użytkownik wywoła zapytanie ze ścieżką do tego widoku, wtedy nastąpi przekierowanie do adresu podanego w ustawieniach systemu pod nazwą „LOGIN_REDIRECT_URL”.

Sprawdzenie czy użytkownik należy do grupy z danymi prawami dostępu

```
@login_required
def manage_report_view(request, report_id):
    """ view the content of the report and allow to manage it """
    report = Report.objects.get(id=report_id)
    is_master_editor = request.user.groups.filter(name='MasterEditors').exists()
    is_editor = request.user.groups.filter(name='Editors').exists()
```

W tym przypadku sprawdzane jest czy użytkownik należy do odpowiedniej grupy użytkowników. Na podstawie tej przynależności widok wykona odpowiednie operacje i odmówi tych, które użytkownikowi nie przysługują.

IV.6.3. Rejestracja

Rejestracja odbywa się w widoku „register_view”, gdzie udostępniany jest formularz rejestracji, a wypełniając go przechwytuje on nowe dane użytkownika i zapisuje do bazy. Formularz jest wzięty z biblioteki „django.contrib.auth.forms” i zawiera wbudowaną weryfikację poprawności wprowadzanych danych w zakresie poprawności i unikalności nazwy użytkownika oraz złożoności hasła.

Widok rejestracji:

```
def register_view(request):
    """ parse and handle registration form """
    user_already_exists = False
    if request.method == 'POST':
        form = UserCreationForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('login')
        else:
            user_already_exists = True

    form = UserCreationForm()
    return render(request, 'register.html', {'form': form,
'user_already_exists': user_already_exists})
```

IV.7. Formularze

Dane wprowadzane przez użytkowników w celu obsługi logowania, operacji na wpisach czy zgłoszeniach i innych odbywa się poprzez implementację formularzy. Framework django dostarcza dużo funkcjonalności pozwalających na łatwą implementację formularzy wraz z tym jak są prezentowane na stronach.

IV.7.1. Definiowanie formularzy

Formularze definiowane są wewnątrz aplikacji w ramach plików “forms.py”. Powstają na podstawie implementacji klas dziedziczących po różnych klasach znajdujących się wewnątrz biblioteki “django.forms”. Najważniejsze to “Form” do tworzenia formularzy od zera oraz “ModelForm” wykorzystującego jako bazę utworzone modele.

Definiowanie elementów formularza odbywa się na 2 sposoby. Pierwszy bazuje na wykorzystaniu modelu określonego w klasie “Meta” wewnątrz klasy formularza (które konkretnie atrybuty są brane pod uwagę definiowane są przy pomocy atrybutów “exclude” lub “include”). Drugim sposobem natomiast jest samodzielne opisanie atrybutu tak jak na przykładzie poniżej:

```
class SearchByTagsForm(forms.Form):
    search_tags = TagField(label='Search by Tags',
widget=TagWidget(attrs={'class': 'form-control form-control-sm'}))
```

IV.7.2. Definiowanie stylów wewnątrz formularzy

Wewnątrz definicji formularzy jest możliwość określania stylu za pomocą zmiennej attrs wewnątrz definicji poszczególnych elementów szablonu. Można tam zamieszczać instrukcje css (w ramach słowa kluczowego “style”) oraz nazwy klas dla obiektów html, które wyświetlają dany element (w ramach słowa kluczowego “class”), do których stosowane są odpowiednie style na podstawie Bootstrapa lub samodzielnie zdefiniowanych stylów. Poniżej kilka przykładów w jaki sposób w projekcie są te style i nazwy klas definiowane:

```
class ArticleForm(forms.ModelForm):
    class Meta:
        model = Article
        exclude = ['author', 'created_at', 'status']
        widgets = {
            'title': forms.TextInput(attrs={'required': True, 'class': 'form-
control form-control-sm'}),
            'tags': TagWidget(
                attrs={'class': 'form-control form-control-sm'}),
        }
```

```
class SearchByNameForm(forms.Form):
    search_title = forms.CharField(label='Search by Title', max_length=255,
widget=forms.TextInput(
        attrs={'required': True, 'class': 'form-control form-control-sm'}))
```

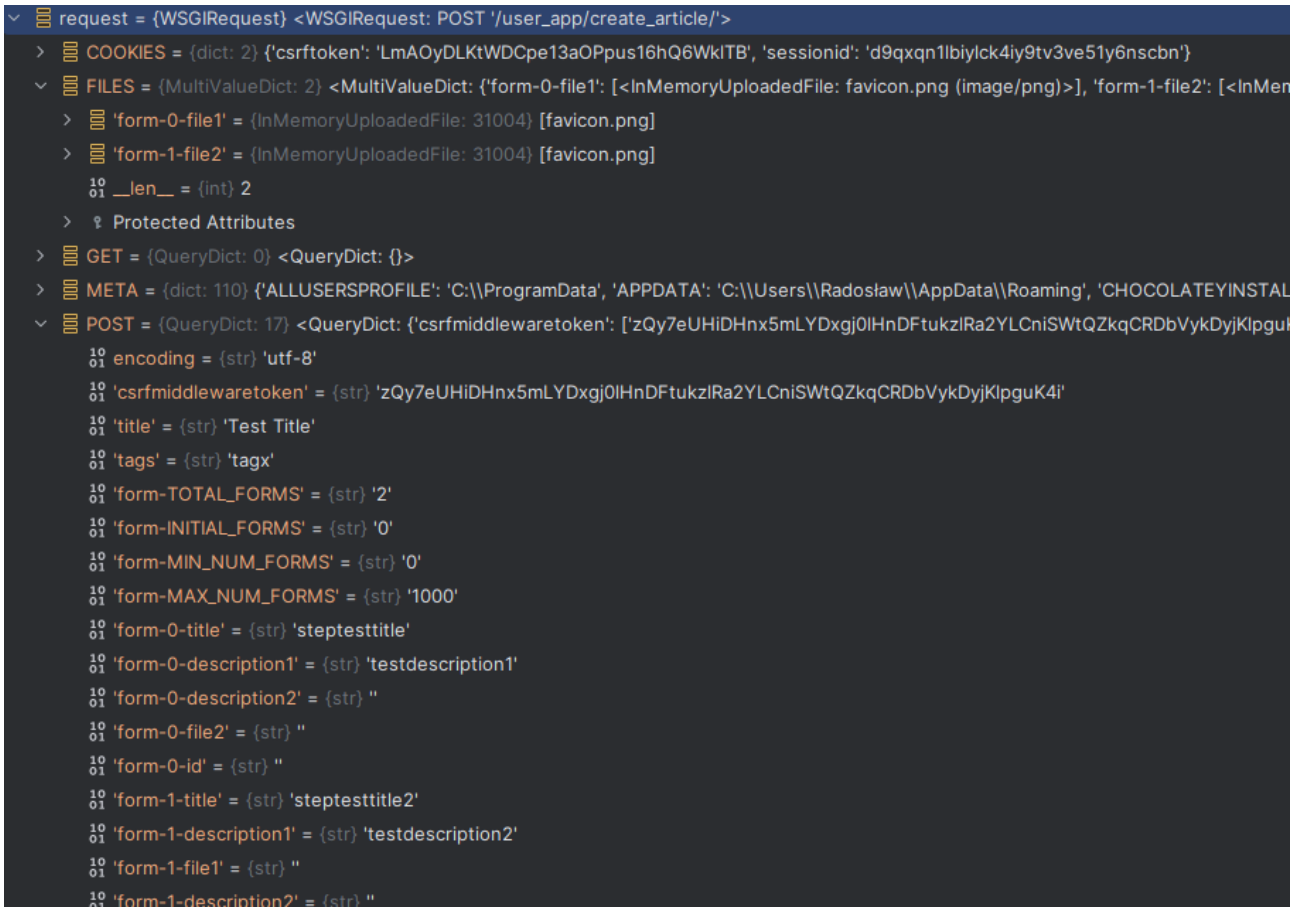
IV.7.3. Przesyłanie formularzy w systemie

Formularze udostępniane są w ramach szablonów html. Przesyłane są poprzez widoki w momencie renderowania strony przy pomocy technologii Jinja2. Posiadają wbudowany mechanizm pozwalający na wyświetlanie ich jako wybrany rodzaj elementu html (p, div, ul, table). Poniżej pokazany jest przykład przesłania formularza oraz jego wyświetlenia w szablonie:

```
return render(request, 'report_article.html', {'report_form': report_form,
'article': article, 'steps': steps})
```

```
<form class='report_form' method='post' enctype='multipart/form-data'>
    {% csrf_token %}
    {{ report_form.as_p }}
    <button type='submit' class='btn btn-dark image_button submit_report_i_b
let_hover' name='submit_report'>
        <img src='{% static 'img/submit_report_icon.png' %}'
alt='submit_report'/>
        <span class='hover_text'>submit report</span>
    </button>
</form>
```

Po przesłaniu rozwiązanego formularza przez użytkownika wysyłane jest zapytanie do serwera wraz ze strukturą słownikową zawierającą wszystkie wypełnione dane. Tak wyglądają przykładowe struktury przekazywanych danych wewnątrz zapytania (na podstawie debuggera wewnątrz IDE PyCharm):



Rys. 13. Zrzut ekranu debuggera od IDE PyCharm dla zapytania POST tworzenia wpisu.

FILES zawiera niezbędne informacje o plikach wstawionych do formularza, natomiast POST zawiera informacje o pozostałych elementach formularza oraz unikalny token CSRF zapewniający bezpieczeństwo przy przesyłaniu formularza.

IV.7.4. Obsługa formularzy

Ze względu na różne przeznaczenie formularzy istnieją w projekcie różne sposoby ich obsługi.

Dla małych formularzy służących tylko do przekierowań między stronami, takich jak ten:

```
<form method='get' action='{% url 'user_panel' %}'>
    <button type='submit' class='btn btn-dark form-control form-control-sm'>
        Go to user panel
    </button>
</form>
```

Obsługa dzieje się wewnątrz frameworka i nie wymaga żadnej obsługi poza posiadaniem odpowiedniej ścieżki do przekierowania.

Dla odpowiedniej reakcji na wywołany formularz taki jak poniżej:

```
<form method='post'>
    {% csrf_token %}
    <button type='submit' class='btn btn-dark image_button let_hover'
```

```
name='close_report'>
        <img src='{% static 'img/close_report_icon.png' %}'
alt='close_report_icon'/>
        <span class='hover_text'>close the report</span>
    </button>
</form>
```

Obsłużony jest poprzez sprawdzenie, czy w zapytaniu występuje nazwa wciśniętego przycisku i jeśli tak, to wywoływane jest odpowiednie działanie, dla powyższego formularzu jest to ten element odpowiedniego widoku:

```
elif 'close_report' in request.POST:
    if ReportStatus.is_about_new_article(report.status):
        report.status = ReportStatus.CONCLUDED.n
    […]  # pozostałe instrukcje
```

Formularze na podstawie modeli posiadają metodę “save” pozwalającą na automatyczną konwersję wprowadzonych danych do obiektu modelu. Nie zmieniając żadnych flag wraz z tą konwersją odbywa się zapisanie obiektu do bazy danych, co dzieje się w przypadku formularzu do rejestracji w poniższy sposób:

```
form = UserCreationForm(request.POST)
if form.is_valid():
    form.save()
```

Natomiast ustawiając flagę “commit” na fałsz, można przypisać obiekt do zmiennej i dokonać zmian jeszcze przed zapisaniem, co odbywa się na przykład przy zapisywaniu zgłoszeń:

```
report_form = ReportForm(request.POST, request.FILES)
if report_form.is_valid():
    save_files(request.FILES)
    report = report_form.save(commit=False)
    report.title = generate_report_title(report.description)
    report.author = request.user
    report.article = article
    report.status = ReportStatus.OPENED.n
    report.save()
```

Drugie wywołanie metody “save” zapisuje aktualny stan obiektu do bazy danych.

IV.7.5. Obsługa grup formularzy

W przypadku tworzenia lub edycji wpisów zaimplementowany jest nieco bardziej zaawansowany mechanizm wymagający dodatkowego opisania. Jest to grupa formularzy (formset), gdzie każdy formularz z tej grupy odpowiada jednemu krokowi we wpisie.

Tak wygląda definicja formularzu dla kroku:

```
class StepForm(forms.ModelForm):
    class Meta:
        model = Step
        exclude = ['article', 'ordinal_number']
        widgets = {
            'title': forms.TextInput(
                attrs={'class': 'form-control form-control-sm', 'required':
True, 'placeholder': 'Title of the step'}),
            'description1': forms.Textarea(attrs={'class': 'form-control form-
control-sm', 'rows': 2, 'cols': 40,
                                                  'placeholder': 'First
description for the step'}),
            'file1': forms.ClearableFileInput(attrs={'class': 'form-control
form-control-sm', 'type': 'file',
                                                     'accept': 'image/*',
'placeholder': 'First image for the step'}),
            'description2': forms.Textarea(attrs={'class': 'form-control form-
control-sm', 'rows': 2, 'cols': 40,
                                                  'placeholder': 'Second
description for the step'}),
            'file2': forms.ClearableFileInput(attrs={'class': 'form-control
form-control-sm', 'type': 'file',
                                                     'accept': 'image/*',
'placeholder': 'Second image for the step'}),
        }
        labels = {key: '' for key in ('title', 'description1', 'file1',
'description2', 'file2')}
```

A tak wygląda definicja grupy formularzy:

```
StepFormSetCreate = forms.modelformset_factory(Step, form=StepForm)
StepFormSetEdit = forms.modelformset_factory(Step, form=StepForm, extra=0)
```

Wykorzystanie grupy formularzy w wdioku odbywa się poprzez stworzenie zmiennej na podstawie powyższego kreatora grupy formularzy oraz listy kroków, które ma zawierać (w przypadku tworzenia wpisu jest to lista bez obiektów, a w przypadku edycji są to aktualnie istniejące kroki tego wpisu):

```
step_form_set = StepFormSetCreate(queryset=Step.objects.none())
step_form_set = StepFormSetEdit(queryset=Step.objects.filter(article=article))
```

W szablonie grupa formularzy wypisywana jest poprzez technologię jinja2 z dodaniem 2 flag pozwalających na bezpieczne ich wyświetlenie:

```
{{ step_form_set.management_form }}
{{ step_form_set.non_form_errors }}
<div id='steps-formset'>
    {% for step_form in step_form_set %}
```

```
        <div class='step-form'>
            <h4>Step {{ forloop.counter }}</h4>
            {{ step_form.as_p }}
            <br>
        </div>
    {% endfor %}
</div>
```

Wypełniona grupa formularzy przekazywana jest tak samo jak zwykły formularz ale z zachowaniem odpowiedniej konstrukcji. Tak prezentuje się przykładowy fragment zapytania POST przedstawiający iformacje o grupie formularzy:
'form-TOTAL_FORMS': ['2'],
'form-INITIAL_FORMS': ['0'],
'form-MIN_NUM_FORMS': ['0'],
'form-MAX_NUM_FORMS': ['1000'],
'form-0-title': ['steptesttitle'],
'form-0-description1': ['testdescription1'],
'form-0-description2': [''],
'form-0-file2': [''],
'form-0-id': [''],
'form-1-title': ['steptesttitle2'],
'form-1-description1': ['testdescription2'],
'form-1-file1': [''],
'form-1-description2': [''],
'form-1-id': ['']

Zatem identyfikacja elementów w grupie odbywa się poprzez używanie wspólnego słowa kluczowego (w tym przypadku „form") jako prefiksu. Każdy formularz w zestawie posiada swój numer identyfikacyjny, dzięki czemu w ramach słownika można dokładnie określić który atrybut należy do danego formularza.

Słownik ten jest tłumaczony na kroki wewnątrz funkcjonalności grupy formularzy i dzięki temu można łatwo obsłużyć je w widoku, co zostało rozwiązane w sposób następujący:

```
step_form_set = StepFormSetCreate(request.POST, request.FILES)
if step_form_set.is_valid():
    save_files(request.FILES)
    ordinal_number = 1
    for step_form in step_form_set:
        step = step_form.save(commit=False)
        step.article = new_article
        step.ordinal_number = ordinal_number
        step.save()
        ordinal_number += 1
```

step_form_set jest tutaj listą zbiórów elementów każdego kroku przekazywaną do step_form, dzięki czemu dzięki metody save do zmiennej step trafia już gotowy obiekt kroku z wypełnionymi danymi.

IV.8. Zastosowane zabezpieczenia

Framework Django zapewnia wiele funkcjonalności posiadających wbudowane zabezpieczenia przed złośliwymi działaniami użytkowników. Poniżej wylistowane są najważniejsze zagrożenia i to jak Django zabezpiecza się przed nimi. Zabezpieczenia te są opisane w odniesieniu do publikacji stworzonych przez popularną organizację zbierającą i popularyzującą wiedzę o cyberbezpieczeństwie OWASP (Open Web Application Security Project).

IV.8.1. Ochrona przed atakami XSS (Cross-Site Scripting):

Ataki Cross-Site Scripting (XSS) polegają na wstrzykiwaniu szkodliwych skryptów do stron internetowych w celu ich wywołania przez aplikację. Prowadzi to potencjalnie do nieautoryzowanego dostępu do danych lub funkcjonalności oraz innych złośliwych działań.

Django radzi sobie z zagrożeniem XSS poprzez automatyczne neutralizowanie znaków w szablonach, które mogłyby być interpretowane jako kod JavaScript. OWASP zaleca również unikanie używania wbudowanych filtrów „safe”, „mark_safe”, lub „json_script”, które wyłączają automatyczne neutralizowanie.

Przykładowy kod, który pokazuje automatyczne neutralizowanie w Django, może wyglądać tak:

```
Step {{ step.ordinal_number }}. {{ step.title }}
```

Są to zmienne przekazane do szablonu, które w przypadku posiadania znaków charakterystycznych dla języka JavaScript konwertuje je na znaki neutralne uniemożliwiając ich uruchomienie jako skryptu (na przykład „<” zamienia na „&lt”).

Żeby bezpiecznie dodać kod JavaScript do szablonu, należy umieścić go w osobnym pliku i odwołać się do niego tak jak na poniższym przykładzie:

```
<script src='{% static 'js/manage_article_handle_steps.js' %}'></script>
```

IV.8.2. Ochrona przed atakami CSRF (Cross-Site Request Forgery):

Zagrożenie Cross-Site Request Forgery (CSRF) polega na wykorzystaniu uwierzytelnionej sesji użytkownika przez osobę trzecią do wykonania nieautoryzowanych działań. Atakujący mogą zmusić przeglądarkę użytkownika do wysłania żądań do aplikacji, z którymi użytkownik jest już uwierzytelniony, potencjalnie prowadząc do nieautoryzowanych działań.

Django chroni przed atakami CSRF poprzez użycie „CsrfViewMiddleware", które dodaje związane z CSRF nagłówki do odpowiedzi aplikacji. W formularzach należy używać tagu szablonu {% csrf_token %}, aby dołączyć token CSRF do zapytania (jest to kluczowe przy zapytaniach POST). Przesyłany on jest wraz z formularzem dla weryfikacji zgodności tokenu z tym zapisanym w sesji użytkownika.

Ten mechanizm jest dodawany do projektu w pliku „settings.py":

```
MIDDLEWARE = [
    # […]
    'django.middleware.csrf.CsrfViewMiddleware',
    # […]
]
```

IV.8.3. Ochrona przed atakami SQL Injection:

SQL Injection jest rodzajem ataku, w którym złośliwy użytkownik może poprzez wprowadzenie odpowiedniego ciągu znaków manipulować zapytaniami SQL uzyskując nieautoryzowany dostęp do danych lub manipulując nimi wbrew woli właściciela systemu.

W celu zapobiegania atakom SQL Injection framework Django stosuje parametryzację zapytań. Kod SQL zapytania jest definiowany oddzielnie od jego parametrów, a wszelkie dane dostarczone przez użytkownika są automatycznie neutralizowanie tak samo jak w przypadku ochrony przed atakami XSS.

Poniżej przykład instrukcji tworzącej zapytanie biorącej parametry do zapytania, które przed wstawieniem do kodu SQL zostaną zneutralizowanie:

```
steps_to_delete = Step.objects.filter(article=article,
ordinal_number__gte=ordinal_number)
```

IV.8.4. Ochrona przed atakami Clickjacking:

Clickjacking jest rodzajem ataku, w którym złośliwa strona internetowa oszukuje użytkownika, aby kliknął na elementy strony docelowej, które są ładowane w ukrytej ramce lub iframe. Atakujący mogą wykorzystać to, aby doprowadzić do niezamierzonego kliknięcia na elementach interfejsu strony docelowej, a to może prowadzić do nieautoryzowanych działań.

Django chroni przed atakami clickjacking poprzez wykorzystanie nagłówka HTTP X-Frame-Options. Ten nagłówek określa, czy zasób może być ładowany w ramce (innymi słowy osadzone wewnątrz „<iframe>”, „<frame>”, „<embed>” lub „<object>”). W projekcie zastosowany jest middleware „XFrameOptionsMiddleware”, który ustawia nagłówek X-Frame-Options dla wszystkich odpowiedzi HTTP na „DENY” (blokowanie ładowania zasobu w ramce) lub „SAMEORIGIN” (ładowanie tylko dla żądania z tej samej witryny). W przypadku tego projektu jest to wartość domyślna, czyli „DENY”.

IV.8.5. Walidacja nagłówków Host:

Zagrożenie związane z nagłówkiem Host polega na możliwości wykorzystania fałszywej wartości Host do przeprowadzenia ataków typu Cross-Site Request Forgery (CSRF), zatrucia pamięci podręcznej (cache poisoning) lub zatrucia linków w e-mailach. Fałszywy nagłówek Host może być używany do konstruowania błędnych URL-i, które mogą prowadzić do niebezpiecznych miejsc lub wykonywać nieautoryzowane działania.

Django radzi sobie z tym zagrożeniem poprzez walidację nagłówków Host względem ustawienia ALLOWED_HOSTS w metodzie django.http.HttpRequest.get_host(). Ta walidacja jest stosowana tylko przez metodę get_host(), więc jeśli kod dostępuje bezpośrednio do nagłówka Host poprzez request.META, oznacza to ominięcie tej ochrony.

IV.8.6. Session Security:

Bezpieczeństwo sesji w Django jest kluczowe dla ochrony danych użytkowników i ich tożsamości. Zagrożenie związane z sesjami polega na możliwości przechwycenia, manipulacji lub kradzieży danych sesji, co może prowadzić do przejęcia kont użytkowników lub wycieku informacji.

Zgodnie z dokumentacją Django, jednym z podstawowych zastosowanych zabezpieczeń jest użycie django.contrib.sessions. Działa ono podobnie do ograniczeń CSRF, wymagających rozmieszczenia witryny w taki sposób, aby niezaufani użytkownicy nie mieli dostępu do żadnych subdomen. W tym celu zastosowane są takie zmienne jak „BACKEND_SESSION_KEY” oraz „HASH_SESSION_KEY” służące do walidacji poprawności sesji oraz zapewniają, że dane sesji są izolowane, a tym samym nie mogą być wykorzystane przez nieuprawnione podmioty.

IV.8.7. Bezpieczeństwo treści przesyłanych przez użytkowników:

Treści przesyłane przez użytkowników mogą stanowić zagrożenie bezpieczeństwa, ponieważ nieufni użytkownicy mogą przesyłać szkodliwe pliki lub skrypty.

Django wychodzi naprzeciw tym zagrożeniom poprzez oferowanie wbudowanych narzędzi do bezpiecznego obsługiwania przesyłanych treści. Dobrym przykładem są tutaj formularze z polem FileInput, które można wykorzystać do bezpiecznego przesyłania plików. Przesłane pliki są dostępne w request.FILES, które jest słownikiem zawierającym klucze dla każdego pola FileInput w formularzu.

Przykład przechwytywania plików z formularza z zapytania:

```
report_form = ReportForm(request.POST, request.FILES)
```

Bezpieczeństwo wynikające z rozdzielenia danych od plików wynika z tego, że rozdzielając je można uniknąć dodatkowego ryzyka wywołania szkodliwego kodu (który w przypadku pliku mógłby nie być zneutralizowany). Oprócz tego następuje rozdzielenie miejsc, gdzie te dane są przechowywane, dzięki czemu zawartość pliku nie trafia do bazy danych, co sprzyja bezpieczeństwu.

IV.9. Sposób przekazywania statusów w projekcie

Do przechowywania statusów wykorzystywana jest popularna struktura, którą jest typ wyliczeniowy „Enum”. W ramach niego przechowywane są również informacje o tym jakie statusy odgrywają rolę przy przeszukiwaniu ich w ramach projektu. Implementacja typu wyliczeniowego dla statusów wpisów oraz zgłoszeń są podobne, dlatego analiza zostanie wykonana na jednym z nich:

```
class ArticleStatus(Enum):
    """ Enum for article statuses """

    def __init__(self, n: int, phrase: str, search_permitted: bool = False):
        self.n = n
        self.phrase = phrase
        self.search_permitted = search_permitted

    @classmethod
    def get_status_name(cls, n: int) -> str:
        """ get status phrase value to display it """
        for status in cls:
            if status.n == n:
                return status.phrase
        raise ValueError('Bad status number')

    APPROVED = (1, 'approved', True)
    UNAPPROVED = (2, 'unapproved', True)
    CHANGES_REQUESTED = (3, 'changes requested', True)
    CHANGES_DURING_REPORT = (4, 'changes during report', True)
    REJECTED = (5, 'rejected')
```

Klasa posiada wewnątrz swój konstruktor, który wykorzystują elementy do zadeklarowania ich dodatkowych atrybutów. Dodatkowo stworzona jest zmienna klasowa, która służy do zwracania odpowiedniej frazy dla statusu w szablonie, wyznaczona na podstawie przekazywanego numeru statusu.

Implementacja ta zapobiega błędnemu wstawianiu nazw statusów (poprzez błędy w zapisie czy nieoczekiwane nazwy wprowadzone przez twórcę oprogramowania). Dodatkowo porządkuje to cały mechanizm statusów, ponieważ ich deklaracje wraz ze szczegółowymi informacjami znajdują się w jednym miejscu kodu.

IV.10. Wsparcie sztucznej inteligencji w zakresie nadawania tytułów zgłoszeń

Użytkownicy opisują zgłoszenie, ale nie tworzą jego tytułu. Za to odpowiedzialna jest funkcjonalność systemu, generująca tytuł we współpracy z modelem opartym na uczeniu maszynowym. Przekazywana jest do niej treść zgłoszenia, a ona zwraca tytuł będący syntezą zgłoszenia.

Poniżej przedstawiona jest funkcja odpowiadająca za tworzenie tytułu:

```
def generate_report_title(description):
    """ generate report title with chatGPT help """
    client = OpenAI(api_key=config('OPENAI_API_KEY'))
    language = config('SYS_LANG')
    if language == 'pl':
        chat_content = (f'Stwórz tytuł do zgłoszenia o takim opisie:
"{description}".'
                        f'Tytuł ma być zwięzły, kilkusłowny oraz pragmatyczny,
bez upiększeń')
    elif language == 'en':
        chat_content = (f'Create a title of the report that has this
description: "{description}".'
                        f'Title must be short, with few words and pragmatic,
without additional exclamations')
    else:
        raise ValueError(f'Language "{language}" not supported')

    chat_completion = client.chat.completions.create(
        messages=[{'role': 'user', 'content': chat_content}],
        model= config('AI_MODEL'),
    )

    response = chat_completion.choices[0].message.content

    if response[0] == '"' and response[-1] == '"':
        response = response[1:-1]

    return response
```

Klucz autoryzacyjny java jest ukryty w zmiennych środowiskowych, tak samo jak nazwa modelu (przykładowa nazwa takiego modelu: „gpt-3.5-turbo”). Po kilku różnych testach okazało się, że w przypadku prośby o tytuł model językowy nie potrafi zwrócić wyniku bez cudzysłowów, więc są one usuwane wewnątrz funkcji.

IV.11. Sposób wyszukiwania wpisów

W projekcie istnieją 4 sposoby wyszukiwania, widoczne są na poniższym zrzucie ekranu:

Rys. 14. Zrzut ekranu widoku wyszukiwarki

Wyszukiwanie po tytule odbywa się przez sprawdzenie, czy w tytule występuje wpisana fraza. Zaimplementowane jest to w następujący sposób:

```
if 'search_by_title' in request.POST:
    search_title_form = SearchByTitleForm(request.POST)
    if search_title_form.is_valid():
        search_text = search_title_form.cleaned_data['search_title']
        search_permitted_statuses = [status.n for status in ArticleStatus if
status.search_permitted]
        searched_articles = Article.objects.filter(
            title__icontains=search_text, status__in=search_permitted_statuses)
    else:
        return HttpResponseBadRequest(f'form not valid: {report_form.errors}')
```

Wyszukiwanie po frazie jest nieco bardziej skomplikowane, bo przeszukuje po tytule oraz po opisach występujących w krokach:

```
if 'search_by_phrase' in request.POST:
    search_phrase_form = SearchByPhraseForm(request.POST)
```

```
    if search_phrase_form.is_valid():
        search_text = search_phrase_form.cleaned_data['search_phrase']
        search_permitted_statuses = [status.n for status in ArticleStatus if
status.search_permitted]

        searched_articles = Article.objects.filter(
            Q(status__in=search_permitted_statuses) & (
                    Q(title__icontains=search_text) |
                    Q(step__title__icontains=search_text) |
                    Q(step__description1__icontains=search_text) |
                    Q(step__description2__icontains=search_text)
            )
        )

    else:
        return HttpResponseBadRequest(f'form not valid: {report_form.errors}')
```

Wyszukiwanie po tagach wykorzystuje zainstalowaną bibliotekę i przy jej pomocy wyszukuje odpowiednie tagi:

```
if 'search_by_tags' in request.POST:
    search_tags_form = SearchByTagsForm(request.POST)
    if search_tags_form.is_valid():
        tags_to_search = search_tags_form.cleaned_data['search_tags']
        tag_objects = Tag.objects.filter(name__in=tags_to_search)
        searched_articles = Article.objects.filter(tags__in=tag_objects)
    else:
        return HttpResponseBadRequest(f'form not valid: {report_form.errors}')
```

Wyszukiwanie po autorstwie pobiera wszystkie wpisy należące do autora:

```
if 'search_by_ownership' in request.POST:
    if request.user.is_authenticated:
        searched_articles = Article.objects.filter(Q(author=request.user))
    else:
        return HttpResponse('HTTP Unauthorized', status=401)
```

IV.12. Wsparcie mechanizmu wyświetlania grup formularzy w szablonie

Grupy formularzy nie posiadają wbudowanych funkcjonalności pozwalających na sprawną konfigurację wyświetlania grupy formularzy w zakresie ich ilości. Dlatego do zapewnienia działającego interfejsu pozwalającego dodawać lub usuwać kroki we wpisie potrzebny jest skrypt, który to zrealizuje. Będą to więc 2 mechanizmy nasłuchujące odpowiednich przycisków i wywołujących odpowiednie instrukcje.

Usunięcie elementu jest prostsze w implementacji, gdyż musi ono pobrać ostatni formularz z grupy i usunąć go (z ograniczeniem, że nie może usunąć pierwszego). Implementacja wygląda następująco:

```
let addStepButton = document.getElementById('add-step-button');
let removeStepButton = document.getElementById('remove-step-button');
let stepsFormset = document.getElementById('steps-formset');
let totalFormsInput = document.querySelector('input[name$="TOTAL_FORMS"]');
```

```
removeStepButton.addEventListener('click', function (e) {
    if (e.target && e.target.classList.contains('remove-step-button')) {
```

```
        e.preventDefault();
        if (totalFormsInput.value > 1) {
            stepsFormset.lastElementChild.innerHTML = '';
            stepsFormset.removeChild(stepsFormset.lastElementChild);
            totalFormsInput.value = parseInt(totalFormsInput.value, 10) - 1;
        } else {
            alert('You have to have at least 1 step');
        }
    }
});
```

W przypadku dodania formularza do grupy funkcja pobiera ostatni element grupy formularzy, następnie przygotowuje ten formularz przez wyczyszczenie bazowej zawartości do stanu bez żadnych elementów, a następnie dodaje go jako ostatni element.

Czyszczenie zawartości pól formularza odbywa się poprzez wstawienie pustego łańcucha znaków do ich zawartości:

```
for (const newStepFormElement of newStepForm.querySelectorAll('input,
textarea')) {
    newStepFormElement.value = '';
    newStepFormElement.defaultValue = '';
}
```

Dodatkowo, z uwagi na typ pola do wprowadzania plików (ClearableFileInput) należy usunąć ewentualne dodatkowe elementy dotyczące starego pliku, który tam się znajdował. Zaimplementowane jest to w taki sposób, że z pierwotnego zestawu elementów pobiera się jedynie to pole odpowiedzialne za wprowadzenie pliku i podmienia się na niego cały zestaw:

```
Array.from(newStepForm.children).forEach(stepFormElement => {
    var fileInputElement = stepFormElement.querySelector('p
input[type="file"]');
    if (fileInputElement) {
        const newContainer = document.createElement('p');
        newContainer.appendChild(fileInputElement.cloneNode(true));
        newStepForm.replaceChild(newContainer, stepFormElement);
    }
});
```

Na koniec należy zamienić numer porządkowy kroku, który realizuje ten formularz i wstawić cały element:

```
newStepForm.innerHTML = newStepForm.innerHTML.replace(/form-\d+/g, 'form-' +
totalFormsInput.value);
totalFormsInput.value = parseInt(totalFormsInput.value, 10) + 1;
newStepForm.innerHTML = newStepForm.innerHTML.replace(/Step \d+/g, 'Step ' +
totalFormsInput.value);

stepsFormset.appendChild(newStepForm);
```

IV.13. Instalacja projektu na platformie heroku

Aby uruchomić aplikację na platformie heroku, należy skonfigurować odpowiednio projekt tak, żeby platforma była w stanie rozpoznać jak przygotować środowisko do uruchomienia systemu. Do tego prowadzono takie zmiany:

1. Zgrano wszystkie zainstalowane biblioteki do pliku „requirements.txt” przy pomocy poniższej komendy:

```
pip freeze > requirements.txt
```

2. Wywołano automatyczną konfigurację w pliku „setting.py”. Do tego wykorzystano funkcję „locals()” zwracającą słownik lokalnych zmiennych Django.

```
django_heroku.settings(locals())
```

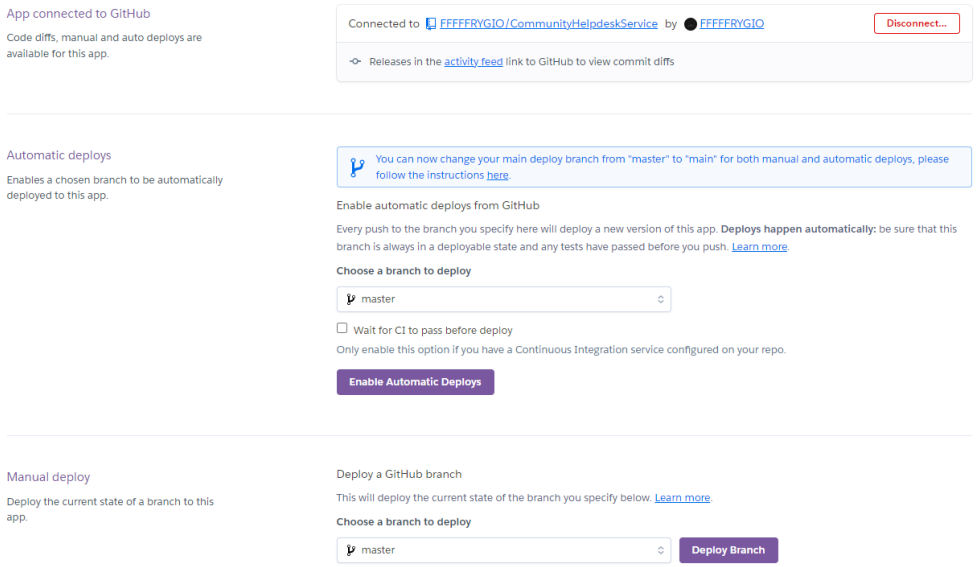
3. Stworzono plik „Procfile” definiujący polecenia, które powinny być wykonane przy uruchamianiu aplikacji:

```
web gunicorn CommunityHelpdeskService.wsgi:application --log-file -
```

4. Stworzono plik „runtime.txt” określający wersję środowiska uruchomieniowego z podaniem konkretnej wersji oprogramowania:

```
python-3.11.3
```

Do udostępnienia projektu platformie należy go uprzednio opublikować na platformie GitHub, a następnie połączyć aplikację heroku do tego repozytorium (określając również branch, na którym jest odpowiednia wersja projektu). Pokazane jest to na poniższym fragmencie strony, gdzie połączenie z platformą jest w ramach segmentu „App connected to GitHub”, a wybrany branch obok przycisku „Deploy Branch”:



Rys. 15. Zrzut ekranu z panelu wdrożeniowego platformy Heroku

Dodatkowo, przez wdrożeniem należy stworzyć i zadeklarować bazę danych oraz wszystkie potrzebne zmienne środowiskowe. Dane do połączenia z bazą danych należy pobrać na podstawie tych podanych w stworzonej bazie danych, pozostałe można nanieść bezpośrednio z pliku „.env”, z zachowaniem ostrożności jeśli chodzi o klucze dostępu lub inne dane wrażliwe.

Kiedy wszystko jest już prawidłowo skonfigurowane, aplikacja jest przygotowana do wdrożenia. Uruchomienie procesu wdrożenia odbywa się przez wciśnięcie przycisku „Deploy Branch”, tego samego, który widoczny jest na zrzucie ekranu powyżej.

IV.14. Konteneryzacja projektu

Poniżej przedstawione są kolejne kroki podjęte do skonfigurowania projektu oraz stworzenia gotowego pakietu instalacyjnego pozwalającego na wdrożenie projektu na różnych urządzeniach.

IV.14.1. Plik „Dockerfile” dla frameworka Django

Plik odpowiada za pobranie kodu i zainstalowanie bibliotek do projektu. Posaiada instrukcje w oparciu o takie słowa kluczowe:

- “FROM” – Wskazuje bazowy obraz, z którego Docker powinien zacząć budowę kontenera
- “WORKDIR” – Ustawia katalog roboczy w kontenerze
- “COPY” – Kopiuje pliki z lokalnego systemu do kontenera
- “RUN” – Wykonuje polecenie w konsoli w kontenerze

Tak wygląda gotowy plik “Dockerfile”

```
FROM python:3.11

WORKDIR /app

COPY . /app/
RUN pip install -r requirements.txt
```

IV.14.2. Kontener z aplikacją webową

Kontener uruchamia aplikację django określając polecenie uruchomieniowe, adres oraz port połączenia z systemem. W ramach procesu wdrożeniowego, ten kontener zostanie uruchomiony dopiero po tym jak kontener bazy danych będzie gotowy na przyjmowanie połączenia, a kontenery od migracji, zbierania plików statycznych oraz dodawania obiektów zakończą swoje działanie z sukcesem. Wszystkie te elementy są wyrażone w deklaracji kontenera:

```
web:
  build: .
  command: python manage.py runserver 0.0.0.0:8000
```

```
  volumes:
    - .:/app
  ports:
    - "8000:8000"
  depends_on:
    db:
      condition: service_healthy
    migration:
      condition: service_completed_successfully
    static_collector:
      condition: service_completed_successfully
    populate:
      condition: service_completed_successfully
```

IV.14.3. Kontener z bazą danych

Kontener ten tworzy bazę danych typu „postgres” na podstawie danych uwierzytelniających zawartych w zmiennych środowiskowych. Dodatkowo, posiada implementację mechanizmu „healthcheck” opisującego sposób, w jakim określa się, że baza danych jest skutecznie stworzona i gotowa na obsługę połączeń (implementacja na podstawie źródła [66]). Deklaracja tego kontenera wygląda następująco:

```
db:
  image: postgres:13-alpine
  environment:
    POSTGRES_DB: ${PG_DB_NAME}
    POSTGRES_USER: ${PG_DB_USER}
    POSTGRES_PASSWORD: ${PG_DB_PASSWORD}
  volumes:
    - postgres_data:/var/lib/postgresql/data
  ports:
    - "${PG_DB_PORT}:${PG_DB_PORT}"
  healthcheck:
    test: [ "CMD-SHELL", "sh -c 'pg_isready -U ${PG_DB_USER} -d ${PG_DB_NAME}'"
]
    interval: 10s
    timeout: 3s
    retries: 3
```

IV.14.4. Kontener odpowiedzialny za migracje

Niewielki kontener, odpowiedzialny za uruchomienie migracji bazodanowych. Swoje działanie rozpoczyna nie wcześniej niż w momencie, którym baza danych będzie gotowa:

```
migration:
  build: .
  command: python manage.py migrate
  depends_on:
    db:
      condition: service_healthy
```

IV.14.5. Kontener odpowiedzialny za wczytywanie danych do systemu

Ten kontener odpowiada za wczytanie danych do systemu na podstawie plików JSON. Pliki przekazane są w takiej kolejności, żeby obiekty wczytywane zostały w momencie, gdy wszystkie jego klucze obce znajdą swoje klucze główne w bazie danych. Uruchamia się dopiero jak zostaną wykonane wszystkie migracje:

```
populate:
  build: .
  command: python manage.py loaddata fixtures/groups.json fixtures/users.json
fixtures/articles.json fixtures/steps.json fixtures/reports.json
  depends_on:
    migration:
      condition: service_completed_successfully
```

Wczytanie danych z projektu do pliku JSON odbywa się poprzez instrukcję „dumpdata” przedstawioną na przykładzie wpisów:

```
python manage.py dumpdata user_app.Article --indent 4 > fixtures/articles.json
```

Tak utworzony plik ma typ kodowania „UTF-16”, którego nie przyjmują instrukcje „loaddata”, stąd należy zamienić konwersje na „UTF-8”. Do tego służy taki zestaw instrukcji dla konsoli PowerShell:

```
$content = Get-Content fixtures/articles.json -Raw
$utf8NoBomEncoding = New-Object System.Text.UTF8Encoding $false
[System.IO.File]::WriteAllText('fixtures/articles.json', $content,
$utf8NoBomEncoding)
```

IV.14.6. Kontener odpowiedzialny za kolekcjonowanie plików statycznych

Ostatni, najmniejszy kontener, którego zadaniem jest stworzenie folderu „staticfiles” wykorzystywanego przez system do przechowywania obrazów, plików stylu i innych plików potrzebnych systemowi do działania:

```
static_collector:
  build: .
  command: python manage.py collectstatic --noinput
```

IV.15. Wybrane przykłady zastosowanych stylów w szablonach

Ten rozdział poświęcony jest całkowicie temu jak wyświetlane są zawartości na stronie. Zaprezentowana będzie zarówno zawartość plików html określających strukturę treści oraz plików css opisujących wygląd i dokładne ich rozmieszczenie.

IV.15.1. Konfiguracja bazowa wszystkich szablonów

Zrealizowany w ramach szablonu „base.html”. Definiuje on elementy strony widoczne dla każdego szablonu (pasek nawigacyjny, stopka) oraz główne style dla całej zawartości (między innymi kolor, rodzaj i wielkość czcionki, kolor tła, przestrzeń dla zawartości konkretnego szablonu). Oprócz tego zawiera on instrukcje implementujące boostrapa do szablonów.

Implementacja bootstrapa odbywa się przy pomocy tych 3 instrukcji:

```
{% load bootstrap5 %}
{% bootstrap_css %}
{% bootstrap_javascript %}
```

Dzięki nim można teraz wykorzystywać w projekcie klasy bootstrapa, opisane w jego dokumentacji.

Szablon zawiera w sobie różne sekcje do wypełnienia przez szablony docelowe, wypisane poniżej:

1. Do wstawienia nazwy pliku css dla konkretnego szablonu

```
<link rel='stylesheet' type='text/css' href='{% block css %}{% endblock %}'>
```

2. Do wstawienia tytułu strony

```
<title>{% block title %}Community Helpdesk Service{% endblock %}</title>
```

3. Do wstawienia podstawowego nagłówka strony

```
<h1><span>{% block heading %}Base{% endblock %}</span></h1>
```

4. Do wstawienia pozostałej zawartości

```
{% block content %}Base Page Content{% endblock %}
```

Pozostałe szablony nadpisują te sekcje wstawiając w ich miejsce odpowiednie treści. Tak wygląda przykładowe wypełnienie:

```
{% block title %}Home{% endblock %}
```

```
{% block content %}
    {% if user.is_authenticated %}
        <h3>User is logged, can't register</h3>
    {% else %}
        <div class='main-box'>
            {% if user_already_exists %}
                <p><b>User already exists! Choose another one</b></p>
            {% endif %}
            <form action='#' method='post' class='form-group'>
                {% csrf_token %}
                {{ form.as_p }}
                <button type='submit' class='btn btn-dark
registration_button'>Register</button>
            </form>
            <h6><a href='/registration/login'>Already have an account?</a></h6>
        </div>
    {% endif %}
{% endblock %}
```

Pasek nawigacyjny złożony jest z listy hiperłączy przekierowujących do różnych widoków. To jakie elementy są wyświetlane zależy od tego czy użytkownik jest zalogowany i jeśli tak to jaką rolę posiada. Poniżej są różne przykłady sprawdzania tych warunków:

Link do panelu użytkownika jeśli użytkownik jest uwierzytelniony:

```
{% if user.is_authenticated %}
    <li class='nav-item active'>
        <a class='nav-link' href='/user_app/user_panel'>User Panel</a>
    </li>
{% endif %}
```

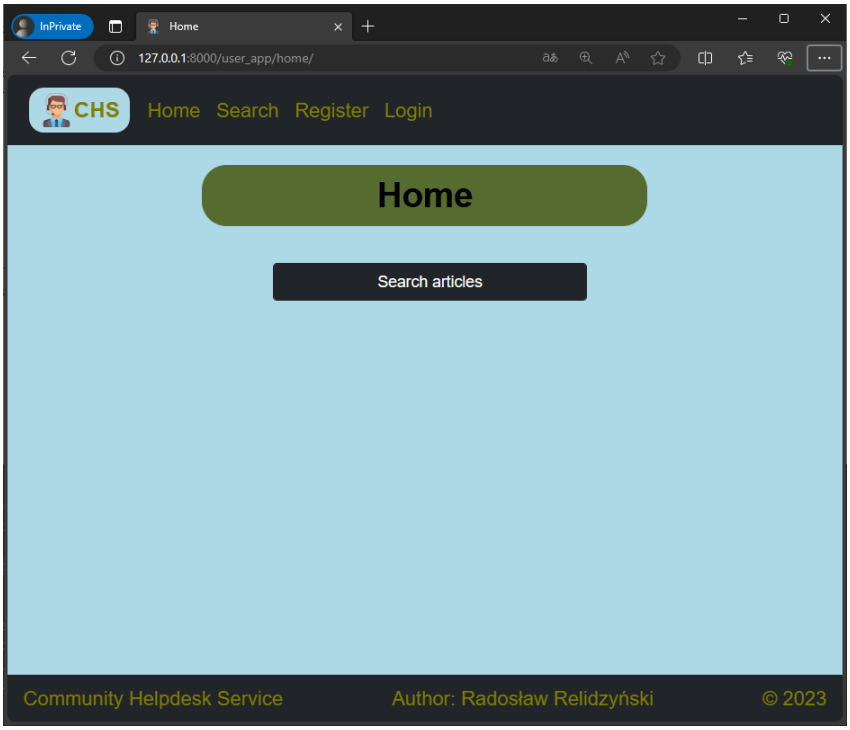
Link do panelu redaktora jeśli jest redaktorem (na podstawie zmiennej w sesji dodawanej przy zalogowaniu):

```
{% if user.is_authenticated and request.session.is_editor %}
    <li class='nav-item active'>
        <a class='nav-link' href='/editor_app/editor_panel'>Editor Panel</a>
    </li>
{% endif %}
```

Link do logowania jeśli użytkownik jest nieuwierzytelniony:

```
{% if not user.is_authenticated %}
    <li class='nav-item active'>
        <a class='nav-link' href='/registration/login'>Login</a>
    </li>
{% endif %}
```

Dodatkowo, na dole strony znajduje się stopka z informacją o nazwie systemu, autorze i roku jego powstania. Tak prezentują się te elementy na przykładzie strony domowej dla niezalogowanego użytkownika:



Rys. 16. Prezentacja elementów szablonu „base.html” na przykładzie strony domowej

IV.15.2. Szablony pomocnicze

Szablony pomocnicze do osobne pliki html do wykorzystania wewnątrz szablonu docelowego. Pozwala to na wielokrotne wykorzystanie tej samej struktury w kilku miejscach. Wykorzystywane jest to między innymi to wyświetlania wyników wyszukania w szablonie „search.html”:

```
{% if search_result %}
    <h3>Search Results:</h3>
    <div class='row'>
        {% for result in search_result %}
            {% include 'articles_list_display.html' with result=result %}
        {% endfor %}
    </div>
{% endif %}
```

Po odwołaniu się do pliku zastosowana jest instrukcja „with” pozwalająca na przekazanie potrzebnych wartości do tego szablonu.

Tak wygląda implementacja szablonu odwoływanego w powyższym przykładzie:

```
{% load static %}

<link rel='stylesheet' type='text/css' href='{% static
'css/display_list_of_articles_or_reports.css' %}'>

<div class='col-lg-6 mb-4'>
    <div class='card list_element'>
        <div class='card-body'>
            <div class='row'>
                <div class='card-text'>
                    <h5>{{ result.article.title }}</h5>
                    <p class='card-text steps_display'>Steps: {{
result.steps_amount }}</p>
                </div>
                <p class='card-text'>Author: {{ result.article.author }}</p>
                <p class='card-text'>
                    Tags: {% for tag in result.article.tags.all %}
                    #{{ tag.name }}{% if not forloop.last %} {% endif %}
                {% endfor %}
                </p>
                <form method='get' action='{% url 'view_article'
result.article.id %}'>
                    <button type='submit' name='view_article'
                            class='btn btn-dark image_button view_article_button
let_hover'>
                        <img class='rounded float-end'
                             src='{% static 'img/view_article_icon.png' %}'
alt='view'>
                        <span class='hover_text'>view article</span>
                    </button>
                </form>
            </div>
        </div>
    </div>
</div>
```

IV.15.3. Pogląd dla obrazów

Ze względu na to, że w ramach wpisów mogą zostać zamieszczone duże zrzuty ekranów, to dla łatwiejszego ich przeglądania zaimplementowany jest mechanizm ich podglądu. Pozwala on na kliknięcie na zdjęcie, co wywołuje wyświetlenie go w dużym rozmiarze, na prawie całej przestrzeni strony.

Tak wygląda zawartość odpowiadająca za wyświetlenie grafiki oraz podglądu do niego:

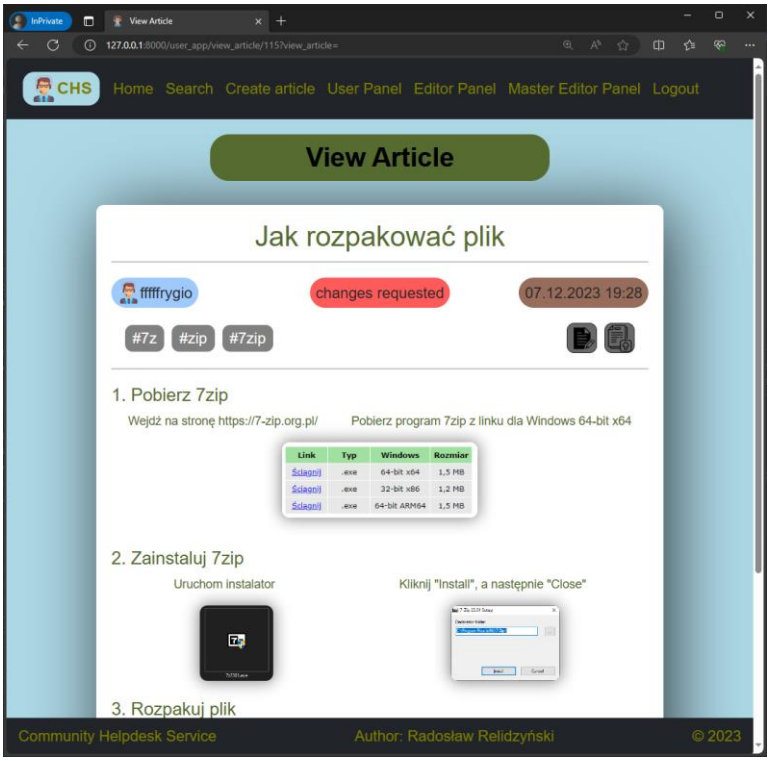
```
<div class='left-image'>
    <a href='#img1'>
        <img src='{{ report.additional_file.url }}' alt='Report Image'>
    </a>
    <a href='#' class='lightbox' id='img1'>
        <span style='background-image: url("{{ report.additional_file.url
}}");'></span>
    </a>
</div>
```

A tak wygląda kod css:

```
.lightbox {
    display: none;

    position: fixed;
    z-index: 999;
    top: 0;
    left: 0;
    right: 0;
    bottom: 0;

    padding: 5%;

    background: rgba(0, 0, 0, 0.8);
}

.lightbox:target {
    display: block;
}

.lightbox span {
    display: block;
    width: 100%;
    height: 100%;

    background-position: center;
    background-repeat: no-repeat;
    background-size: contain;
}
```

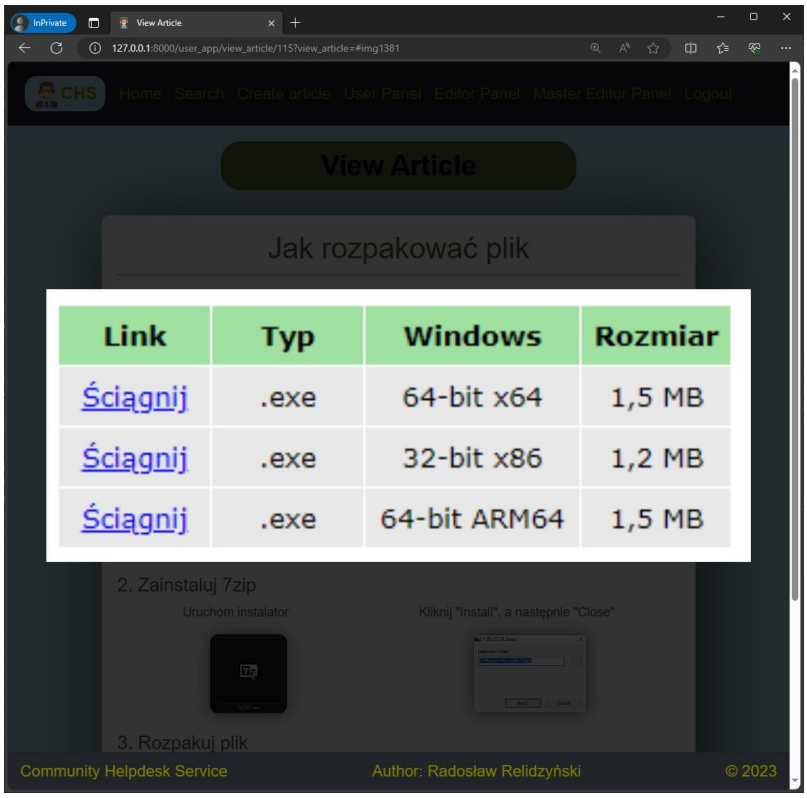
Do każdego wyświetlenia zdjęcia dołączony jest element „span”, który zawiera wyświetlaną grafikę. Jest on bazowo ukryty, ale po kliknięciu na obrazek (osadzony jako link) na stronie zostaje on pokazany.

Tutaj jest strona wyświetlająca wpis:



Rys. 17. Widok strony z wyświetlonym wpisem

A tutaj jak to wygląda po naciśnięciu na grafikę z kroku pierwszego:



Rys. 18. Widok strony z włączonym podglądem zdjęcia

IV.15.4. Wiadomość po najechaniu na przycisk

W projekcie przyciski wyrażone są grafikami reprezentującymi działanie, które wywołanie przycisku powoduje. Można dzięki temu na podstawie skojarzeń określić który przycisk służy do jakiej operacji. Jednakże nie można zakładać, że każdy użytkownik domyśli się danej operacji, więc każdy przycisk posiada dodatkową wiadomość, która wyświetla się po najechaniu na niego kursorem.

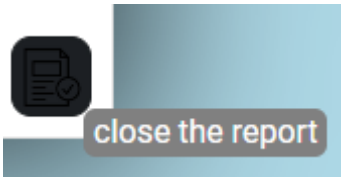
W szablonie wygląda to w sposób następujący:

```
<button type='submit' class='btn btn-dark image_button let_hover'
name='close_report'>
    <img src='{% static 'img/close_report_icon.png' %}'
alt='close_report_icon'/>
    <span class='hover_text'>close the report</span>
</button>
```

A tak wygląda konfiguracja stylu:

```
.hover_text {
    visibility: hidden;
    white-space: nowrap;
    width: auto;
    background-color: grey;
    color: #fff;
    text-align: center;
    border-radius: 6px;
    padding-left: 5px;
    padding-right: 5px;

    position: absolute;
    top: 35px;
    left: 35px;
    z-index: 1;
}

.let_hover:hover .hover_text {
    transition-delay: 1s;
    visibility: visible;
}
```

Wiadomość jest ukrywa, a w momencie najechania na przycisk po sekundzie wyświetli się w prawym dolnym rogu przycisku tak jak poniżej:

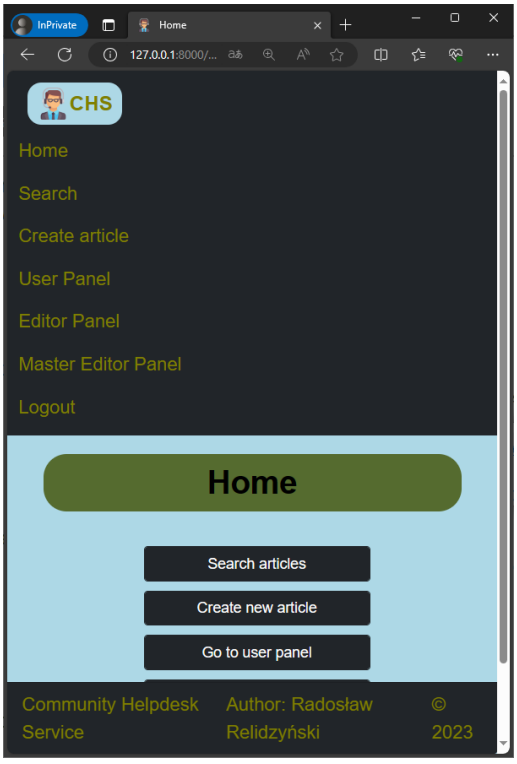


Rys. 19. Prezentacja dodatkowej wiadomości wyświetlanej po najechaniu kursorem na przycisk

IV.15.5. Dostosowywanie rozmieszczenia treści do wymiarów okna

Aby zapewnić czytelność strony dla różnych rozdzielczości zastosowane są mechanizmy pozwalające na dynamiczne dostosowywanie wyświetlanych treści do szerokości lub wysokości strony. Poniżej zaprezentowane są przykładowe fragmenty strony oraz to w jaki sposób zostały zaprogramowane.

Rys. 20. Widok paska nawigacyjnego przy odpowiednio dużej szerokości okna

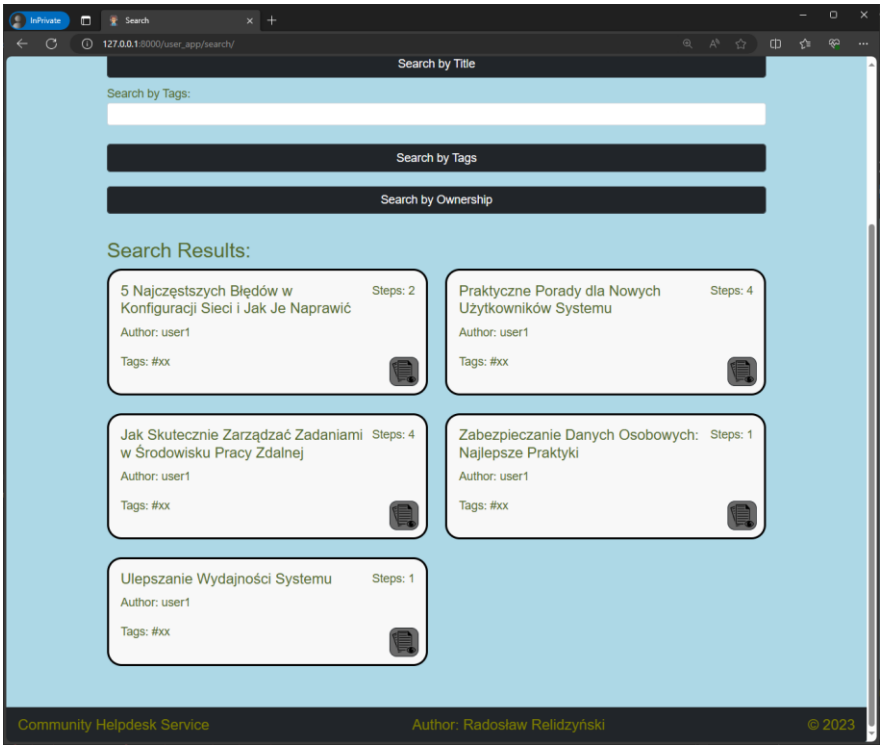


Rys. 21. Widok paska nawigacyjnego przy odpowiednio małej szerokości okna

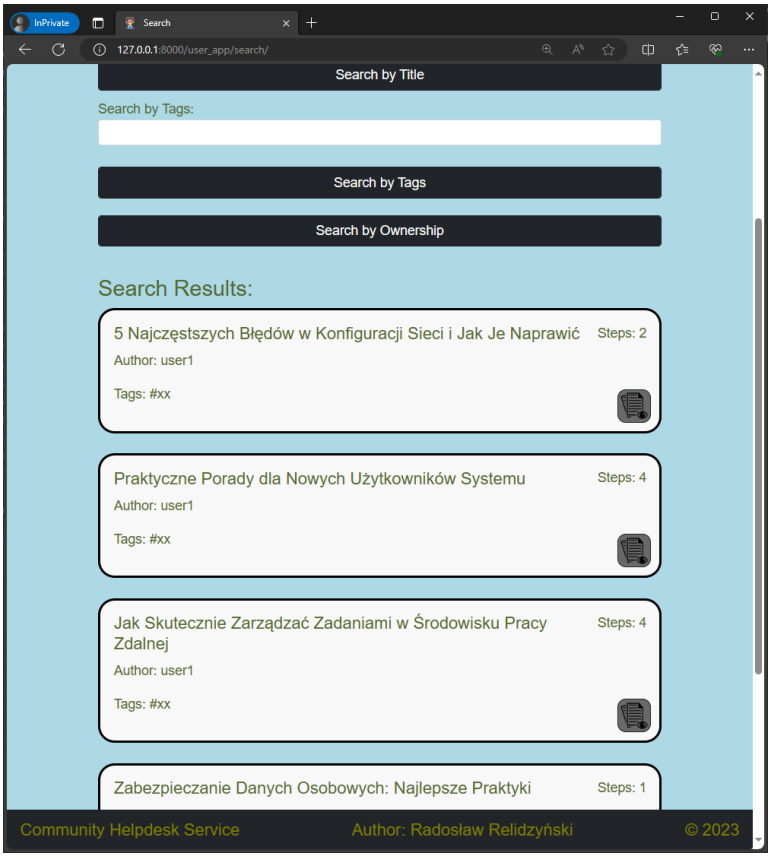
Dzieje się tak dzięki klasie bootstrapowej „navbar-expand-sm” w deklaracji paska:

```
<nav class='navbar navbar-expand-sm bg-dark container-fluid'>
```

Zmiana liczby kolumn w wynikach wyszukiwani



Rys. 22. Widok wyników wyszukiwania przy odpowiednio dużej szerokości okna



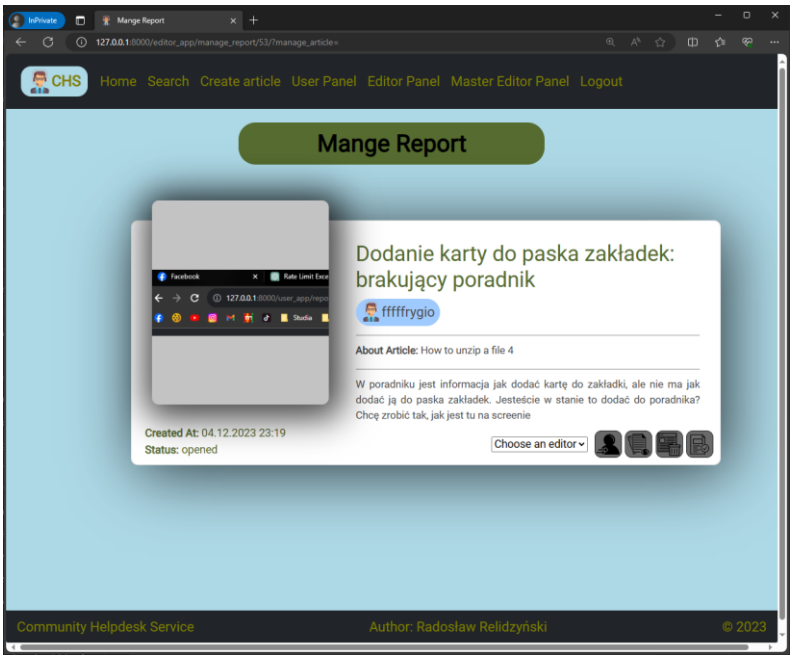
Rys. 23. Widok wyników wyszukiwania przy odpowiednio małej szerokości okna

Tak wygląda deklaracja listy:

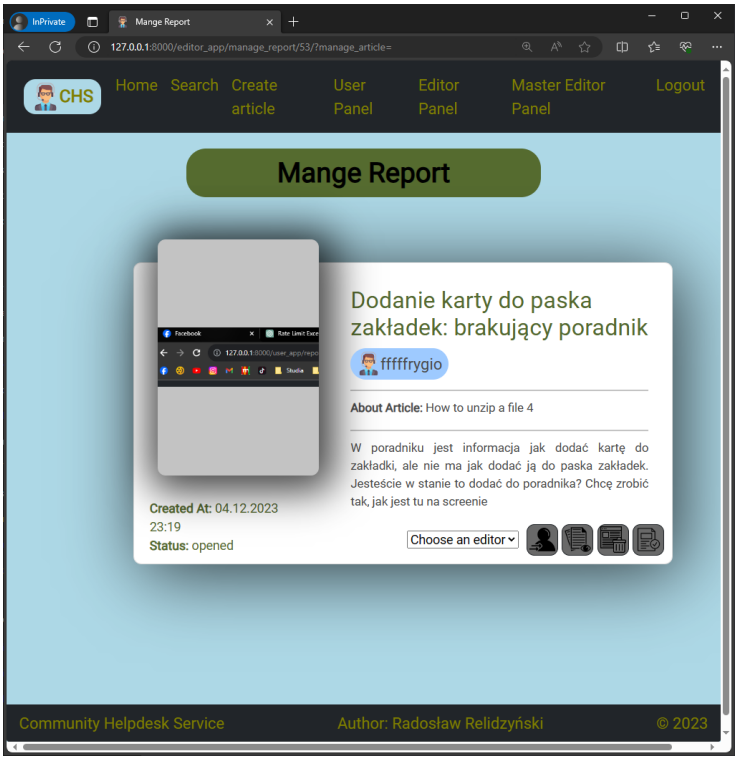
```
<div class='col-lg-6 mb-4'>
```

„col-lg-6” oznacza, że przy odpowiednio dużej szerokości okna elementy będą wyświetlane w systemie siatki, a każdy element będzie zajmował 6 z 12 możliwych kolumn (połowę dostępnej przestrzeni).

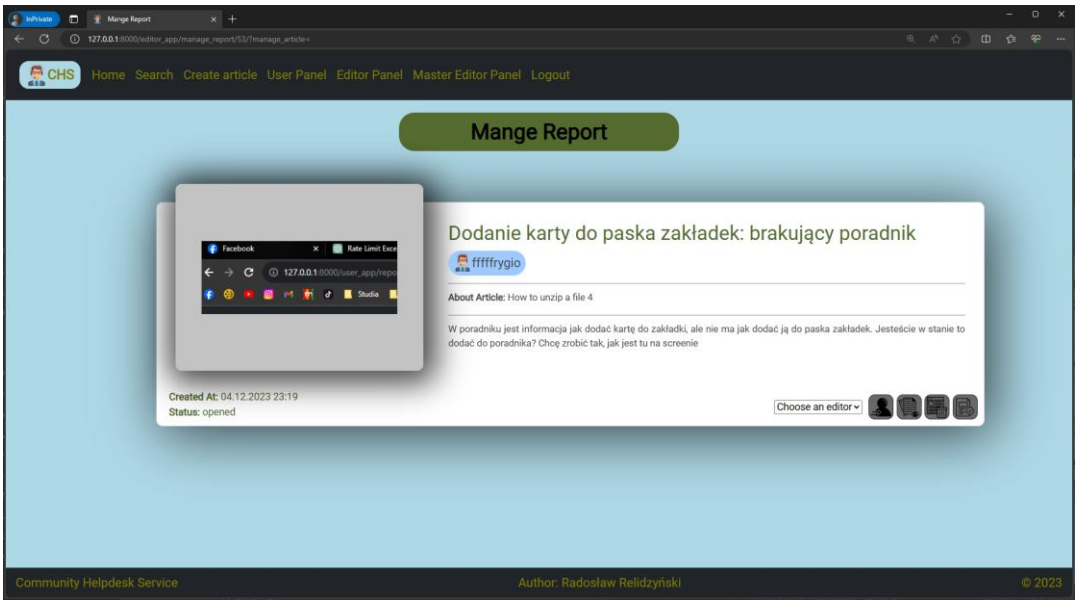
Dostosowywanie wymiarów zrzutu ekranu zgłoszenia do wielkości dostępnego pola



Rys. 24. Widok zgłoszenia dla średniej szerokości okna



Rys. 25. Widok zgłoszenia dla małej szerokości okna



Rys. 26. Widok zgłoszenia dla dużej szerokości okna

W ramach tego dostosowania spełnione są następujące założenia:

1. Obraz nigdy nie przekracza granic pola w którym się znajduje.
2. Maksymalna wielkość wyświetlenia obrazu to jego pierwotne wymiary.
3. Obraz nie zmienia kształtu, jedynie rozmiar.
4. Obraz dostosowuje się dynamicznie do rozmiarów pola jeśli wysokość lub szerokość pola jest mniejsza od wysokości lub szerokości obrazu.

Rozwiązanie w kodzie wygląda następująco:

```
.left-image {
    float: left;
    position: relative;
    left: 30px;
    top: -30px;
    height: 300px;
    width: 100%;
    box-shadow: 5px 5px 50px 15px rgba(0, 0, 0, 0.75);
    background-color: #C3C3C3;
    overflow: hidden;
    display: flex;
    align-items: center;
    justify-content: center;
    border-radius: 10px;
}

.left-image img {
    max-height: 300px;
    max-width: 100%;
}
```

Zdjęcie w polu jest wyśrodkowane, a jego maksymalne wymiary są równe docelowym wymiarom pola.

Rozdział V. Testowanie

Testowanie aplikacji ma kluczowe znaczenie w procesie tworzenia oprogramowania. Jego głównym celem jest sprawdzenie czy oprogramowanie spełnia ustalone kryteria. Testy weryfikują, czy produkt zgadza się ze specyfikacją i czy spełnia oczekiwania użytkowników. Ze względu na zastosowanie podejścia TDD testy są tworzone na bieżąco w trakcie implementacji.

V.1. Środowisko testowe

Framework Django dostarcza kompletne środowisko testowe w ramach biblioteki „django.test" będącej rozszerzeniem standardowej biblioteki „unittest". Dużą jego zaletą jest to, że podczas uruchamiania testów tworzona jest oddzielna baza danych będąca repliką tej podstawowej. Zapewnia to jednolite środowisko testowe oraz nienaruszalność podstawowej bazy danych. Dane testowe tworzone są poprzez implementację metody „setUpTestData". Dodatkowo, dla każdego testu można definiować instrukcje wywoływane przed i po nim, służą do tego metody „setUp" oraz „tearDown". Istnieją również takie metody jak „setUpClass" oraz „tearDownClass", natomiast nie odnalazły one zastosowania w projekcie. Testy uruchamiane są za pomocą polecenia „python manage.py test tests", gdzie „tests" to nazwa folderu zawierającego testy.

Wszystkie klasy testowe będą dziedziczyły po głównej klasie testowej o nazwie „MainTestBase". To w niej przechowywane będą informacje o tworzonych testowych użytkownikach, którzy będą w ramach testów wywoływać zapytania do serwera.

Testy działają poprzez wysyłanie zapytań do serwera i walidację poprawności przychodzących danych oraz stanu bazy danych. Tu są przykładowe zapytania:

```
response = self.client.post(reverse('login'),
                            data={'username': USERS[0]['username'], 'password':
USERS[0]['password']})
```

```
response = self.client.post(reverse('logout'))
```

```
response = self.client.get(reverse('search'))
```

Weryfikacja odpowiedzi od serwera odbywa się poprzez metodę „assertContains", która sprawdza, czy dany ciąg znaków zawarty jest w atrybucie odpowiedzi „response.content" będący łańcuchem binarnym zawierającym kod strony html. Oprócz tego sprawdzana jest poprawność przekierowywania stron przy pomocy metody „assertRedirects". Poniżej przedstawione są przykłady użyć tych oraz innych metod walidacji poprawności:

```
self.assertEqual(response.status_code, 200)
```

```
self.assertContains(response, article.title)
```

```
self.assertRedirects(response, reverse('home'))
```

```
self.assertEqual(Step.objects.count(), initial_step_count + 2)
```

```
self.assertNotContains(response, article.title)
```

V.2. Testy dostępu

Testy dostępu stanowią największą grupę testów. Składa się na nie klasa bazowa „AccessTestsBase” oraz 4 klasy dziedziczące po niej reprezentujące każdego użytkownika. W ich ramach sprawdzana jest widoczność tych elementów, które wspomniane zostały w projekcie aplikacji, gdzie określone jest jakie elementy widzi dany rodzaj użytkownika w zależności od widoczności.

Tak wygląda przykładowy test, ten sprawdza zawartość i dostęp do strony do zgłaszania wpisu:

```
def test_report_article_page_access_and_content(self):
    articles = Article.objects.all()
    for article in articles:
        response = self.client.get(reverse('report_article', args=[article.id]))

        if self.client.session.get('_auth_user_id'):
            self.assertEqual(response.status_code, 200)
            self.assertContains(response, '<h1><span>Report
Article</span></h1>')
            self.assertContains(response, article.title)
        else:
            self.assertRedirects(response,
                                 reverse('login') + '?next=' +
reverse('report_article', args=[article.id]))
```

Widać, że test monitoruje informacje o uwierzytelnianiu i o autoryzacji użytkownika. Instrukcja „self.client.session.get('_auth_user_id')” jest prawdziwa wtedy, gdy w sesji znajduje się podana zmienna symbolizująca udane uwierzytelnienie.

Jeśli chodzi o autoryzację, to sprawdzana jest na podstawie przynależności użytkownika do grupy, tak jak poprzez tą instrukcję:

```
self.user.groups.filter(name='MasterEditors').exists()
```

V.3. Testy wpisów

Realizowane wewnątrz klasy „ArticleTests”. Sprawdzają one jak system reaguje na różnego rodzaju konfiguracja przy edycji wpisu (brak zmian, zmiany w tekście, zmiany w liczbie kroków). Zastosowana jest parametryzacja pozwalająca na uruchomienie jednego testu kilka razy zmieniając część jego scenariusza. Widoczne jest to przy deklaracji metody testowej tak jak w tym przypadku:

```
@parameterized.expand([1, 2])
def test_edit_article_add_steps(self, modify_amount_number):
```

Ten test zostanie uruchomiony dwa razy z różną wartością modify_amount_number, dzięki czemu sprawdzane są dwa różne przypadki.

Wielokrotnie sprawdzane są tutaj całe obiekty wpisów. Żeby to usprawnić sprawdzane są ich reprezentacje słownikowe po uprzednim usunięciu atrybutu „_state”, którego różnice w obiektach nie ma znaczenia z perspektywy testowanych scenariuszy. Poniżej przedstawiony jest przykład instrukcji realizujących takie sprawdzenie:

```
del article._state, edited_article._state
self.assertNotEqual(article.__dict__, edited_article.__dict__)
```

V.4. Testy zgłoszeń

Testy zgłoszeń zaimplementowane są w klasie „ReportsTests” i sprawdzają, jak zachowuje się system wobec zgłoszeń i zorientowanych wokół nich działań użytkowników. Każdy test ma dokładnie opisane kroki, jakie podejmuje, sprawdzając czy zgłoszenie zostało pomyślnie utworzone i czy jego widoczność na stronach jest zgodna z oczekiwaniami.

Tak wygląda przykładowy test:

```
def test_manual_report(self):
    # 1. User1 creates article
    self.user_create_article()
    # 2. User2 creates report
    self.user_report_article()
    # 3. Master Editor see the report
    self.master_editor_see_the_report('open')
    # 4. Editor1 and Editor2 can't see a report
    self.editor_can_see_the_report('open')
```

Każda metoda pomocnicza realizuje zadanie opisane w jej nazwie w oparciu o konkretnego użytkownika i sprawdza, czy zmiany spełniają oczekiwania. Dobrze to widać na podstawie metody „user_create_article”:

```
def user_create_article(self):
    """ Create article to be reported"""
    response = self.client.post(reverse('login'),
                                data={'username': USERS[0]['username'],
'password': USERS[0]['password']})
    self.assertRedirects(response, reverse('home'))

    step_form_set = StepFormSetCreate(FORM_DATA)
```

```
    self.assertTrue(step_form_set.is_valid(), f'step_form_set not valid:
{step_form_set.errors}')
    response = self.client.post(reverse('create_article'), data=FORM_DATA)

    self.assertRedirects(response, reverse('home'))
    articles = Article.objects.filter(title=FORM_DATA['title'])
    self.assertEqual(len(articles), 1)
    self.assertEqual(len(Step.objects.filter(article=articles[0])), 2)

    response = self.client.post(reverse('logout'))
    self.assertRedirects(response, reverse('login'))
```

Metoda loguje się jako odpowiedni użytkownik, wykonuje swoje działanie, sprawdza, czy jego działanie zakończyło się sukcesem, a następnie wylogowuje użytkownika z sesji.

V.5. Testy statusów

Zaimplementowane w klasie „StatusesTests". Również korzystają z parametryzacji, zapewniając sprawdzanie poprawności statusów dla różnych scenariuszy. We wszystkich przypadkach dekorator wygląda tak:

```
@parameterized.expand(['reject', 'edit reject', 'close', 'edit close'])
```

Każde słowo oznacza działanie jakiego test ma się podjąć: odrzucić wpis („reject"), zamknąć zgłoszenie („close") czy dokonać edycji wpisu („edit").

Zadaniem tych testów jest sprawdzanie zgodności zmian statusów we wpisach i zgłoszeniach zgodnych z wytycznymi zaprezentowanymi w projekcie aplikacji. Tak wyglądają testy dla zgłoszeń dotyczących sprawdzenia nowo utworzonych wpisów:

```
@parameterized.expand(['reject', 'edit reject', 'close', 'edit close'])
def test_statuses_new_article(self, editor_behavior):
    self.user_create_article()
    self.master_editor_assign_the_report('new')
    if 'edit' in editor_behavior:
        self.editor_editing_article('new')

    if 'close' in editor_behavior:
        self.editor_closes_report('new')
    elif 'reject' in editor_behavior:
        self.editor_rejects_report('new')
```

Metody działają podobnie do tych znajdujących się w testach dla zgłoszeń. Użytkownik wykonuje działanie, a po jego wylogowaniu następuje sprawdzenie czy jego działania wywołały odpowiednie zmiany ze szczególnym naciskiem na zmiany statusów. Dobrze to prezentuje metoda, w ramach której redaktor dokonuje zmian we wpisie:

```
def editor_editing_article(self, report_type):
    """ Edit article by assigned editor """
    response = self.client.post(reverse('login'),
                                data={'username': USERS[2]['username'],
'password': USERS[2]['password']})
    self.assertRedirects(response, reverse('home'))

    report = None
    if report_type == 'new':
        report = Report.objects.get(description=f'Review new article
"{FORM_DATA["title"]}"')
```

```
    elif report_type == 'open':
        report = Report.objects.get(description=f'New report about article
"{FORM_DATA["title"]}"')

    article = Article.objects.get(id=report.article_id)

    edited_data = {
        'title': article.title,
        'tags': 'new_tag',
    }

    self.assertNotEquals(article.tags, edited_data['tags'])

    response = self.client.post(reverse('edit_article', args=[article.id]),
data=edited_data)
    self.assertRedirects(response, reverse('home'))

    edited_article = Article.objects.get(id=report.article_id)
    self.assertEqual(list(edited_article.tags.values_list('name', flat=True)),
edited_data['tags'].split(', '))

    edited_report = None
    if report_type == 'new':
        edited_report = Report.objects.get(description=f'Review new article
"{FORM_DATA["title"]}"')
    elif report_type == 'open':
        edited_report = Report.objects.get(description=f'New report about
article "{FORM_DATA["title"]}"')

    response = self.client.post(reverse('logout'))
    self.assertRedirects(response, reverse('login'))

    if report_type == 'new':
        self.assertEqual(edited_report.status, 'na changes applied')
    elif report_type == 'open':
        self.assertEqual(edited_report.status, 'changes applied')

    self.assertEqual(report.article.status, 'changes during report')
```

Dokonywana jest więc edycja wpisu przez wprowadzenie nowych wartości, a po tym jak już redaktor się wyloguje następuje sprawdzenie czy wpis i zgłoszenie jego dotyczące zostało zaktualizowane.

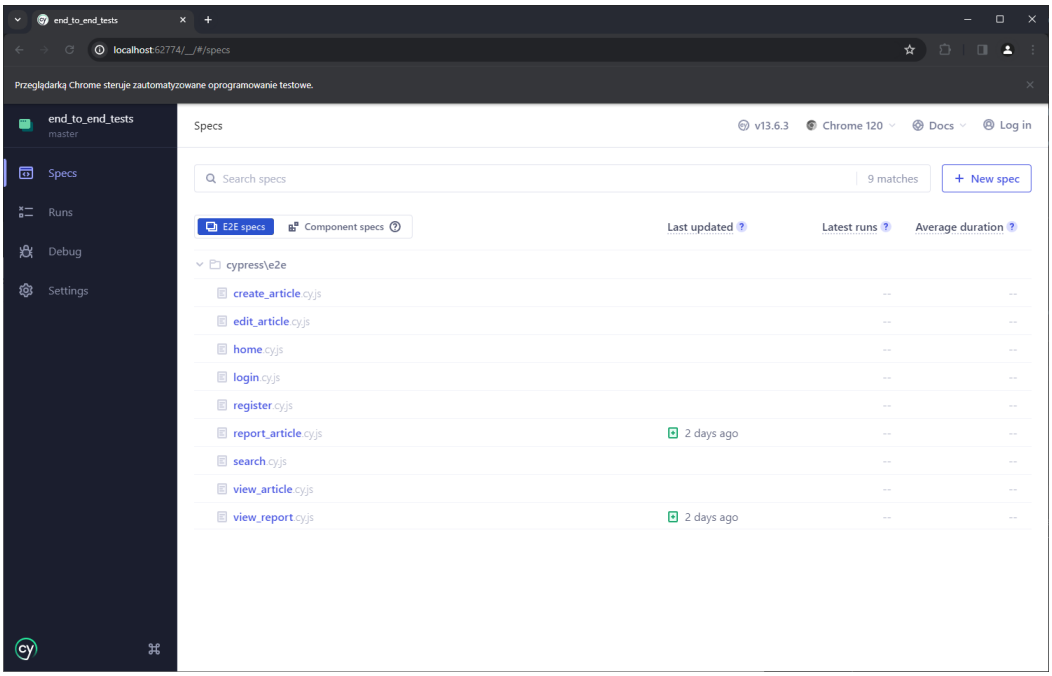
V.6. Testy end-to-end

Niezależnym środowiskiem testowym o tego opisanego wcześniej jest środowisko poświęcone testom end-to-end stworzone przy użyciu narzędzia „Cypress”.

Przygotowanie środowiska opiera się o uruchomienie takich poleceń:

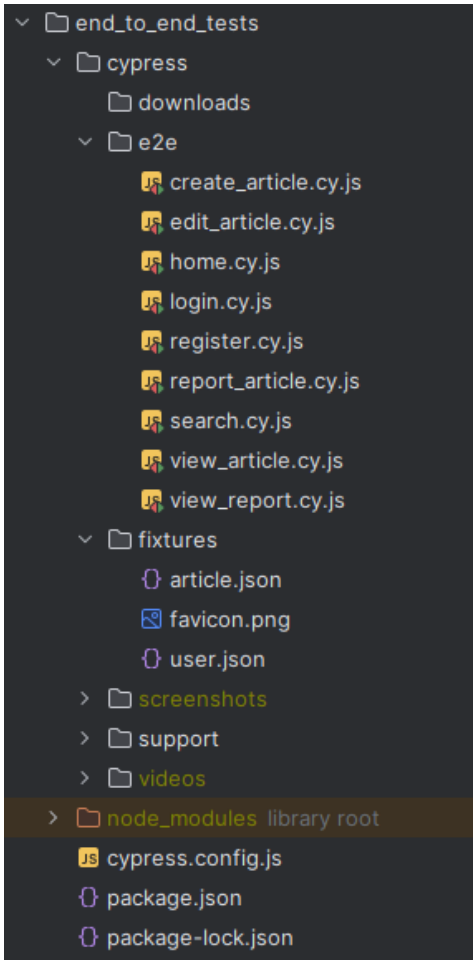
```
npm install cypress --save-dev
npx cypress open
```

Druga instrukcja otwiera interfejs testowy:



Rys. 27. Widok panelu zarządzania testami end-to-end w Cypress

Na tym zrzucie ekranu widać listę wszystkich utworzonych testów. Testy tworzone są w oparciu o automatycznie stworzoną strukturę:



Rys. 28. Struktura folderu testów end-to-end

W ramach niej zawarte są testy (folder „e2e”), dane testowe (folder „fixtures”), pomocnicze implementacje do testów (skrypt „commands.js”) oraz niezawarte w repozytorium projektu nagrania i zrzuty ekranu z przeprowadzonych testów (foldery „screenshots” oraz „videos”).

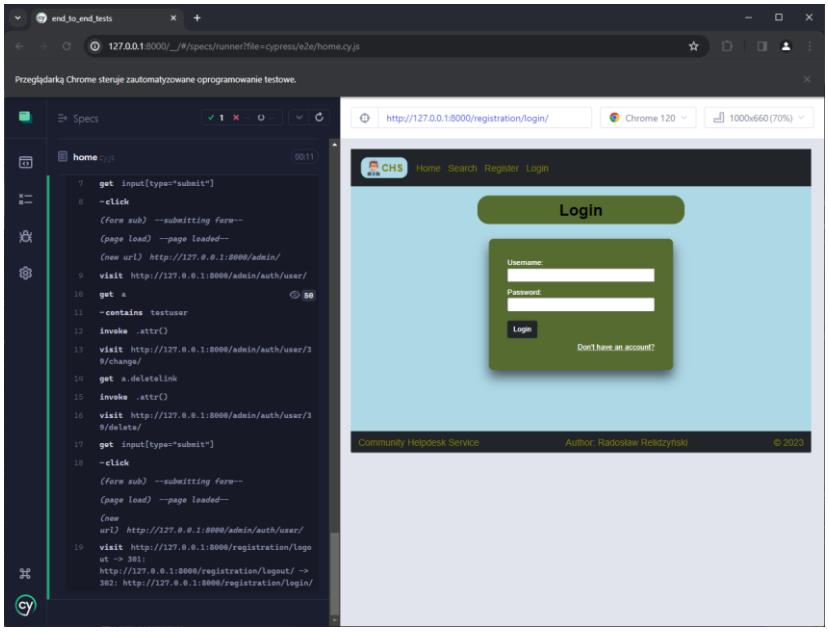
Scenariusze testowe realizowane są poprzez symulację działań użytkownika – ich implementacja opiera się o znajdowanie fragmentów strony, wywoływanie odpowiednich działań na nich i sprawdzanie jak strona reaguje. Na przykład, do znajdowania obiektów jest instrukcja „get()”, do wpisyswania treści „type()”, a do klikania na elementy „click()”. Testy napisane są w bardzo zbliżony do siebie sposób, na podstawie dokumentacji ze źródła [67] .Ich działanie zaprezentuję na podstawie testów dla strony bazowej „home”:

```
describe('template spec', () => {

    beforeEach(() => {
        cy.fixture('user').then(user => {
            cy.register_user(user.username, user.password);
            cy.login_user(user.username, user.password);
        });
    });

    it('Home Test', () => {

        cy.visit('http://127.0.0.1:8000/user_app/home/');
        cy.get('button').contains('Search articles').click();
        cy.location().should((loc) => {
            expect(loc.href).to.eq(
                'http://127.0.0.1:8000/user_app/search/?'
            );
        });

        cy.visit('http://127.0.0.1:8000/user_app/home/');
        cy.get('button').contains('Create new article').click();
        cy.location().should((loc) => {
            expect(loc.href).to.eq(
                'http://127.0.0.1:8000/user_app/create_article/?'
            );
        });

        cy.visit('http://127.0.0.1:8000/user_app/home/');
        cy.get('button').contains('Go to user panel').click();
        cy.location().should((loc) => {
            expect(loc.href).to.eq(
                'http://127.0.0.1:8000/user_app/user_panel/?'
            );
        });

    });

    afterEach(() => {
        cy.visit('http://127.0.0.1:8000/registration/logout');
        cy.fixture('user').then(user => {
            cy.cleanup_user(user.username);
        });
    });

});
```

Każdy test posiada swoje instrukcje przygotowujące i przywracające stan początkowy sprzed testu, wyrażone są odpowiednio w funkcjach „beforeEach” oraz „afterEach”, w tym przypadku obsługują utworzenie użytkownika testowego, żeby go następnie po teście usunąć.

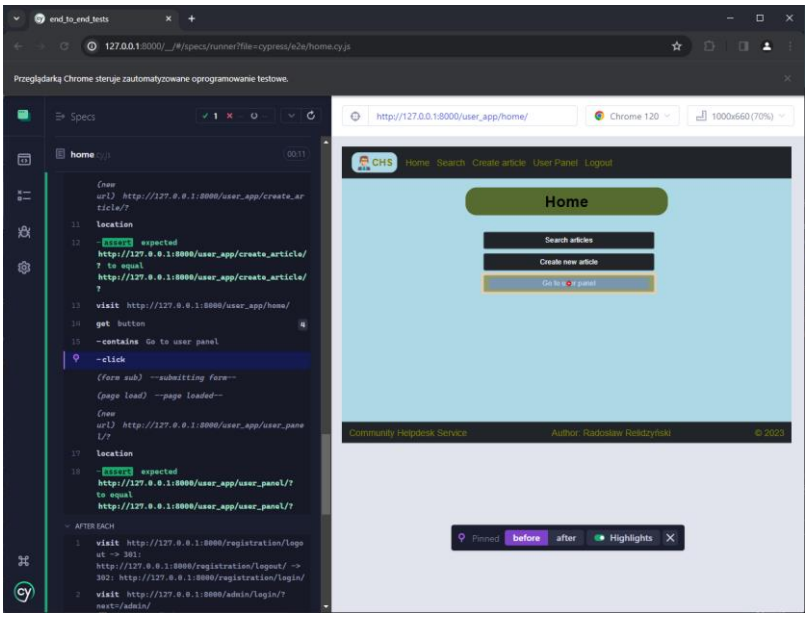
Kolejne kroki testów są w ramach funkcji „it”, które dla strony bazowej sprawdzają wszystkie linki przekierowujące, które się tam znajdują. Po każdym przekierowaniu jest sprawdzane czy zaszło skuteczne przekierowanie na odpowiednią stronę (służą do tego instrukcje „should” oraz „expect”).

Testy uruchamiane są wewnątrz panelu, ich wywołanie prezentuje się następująco:

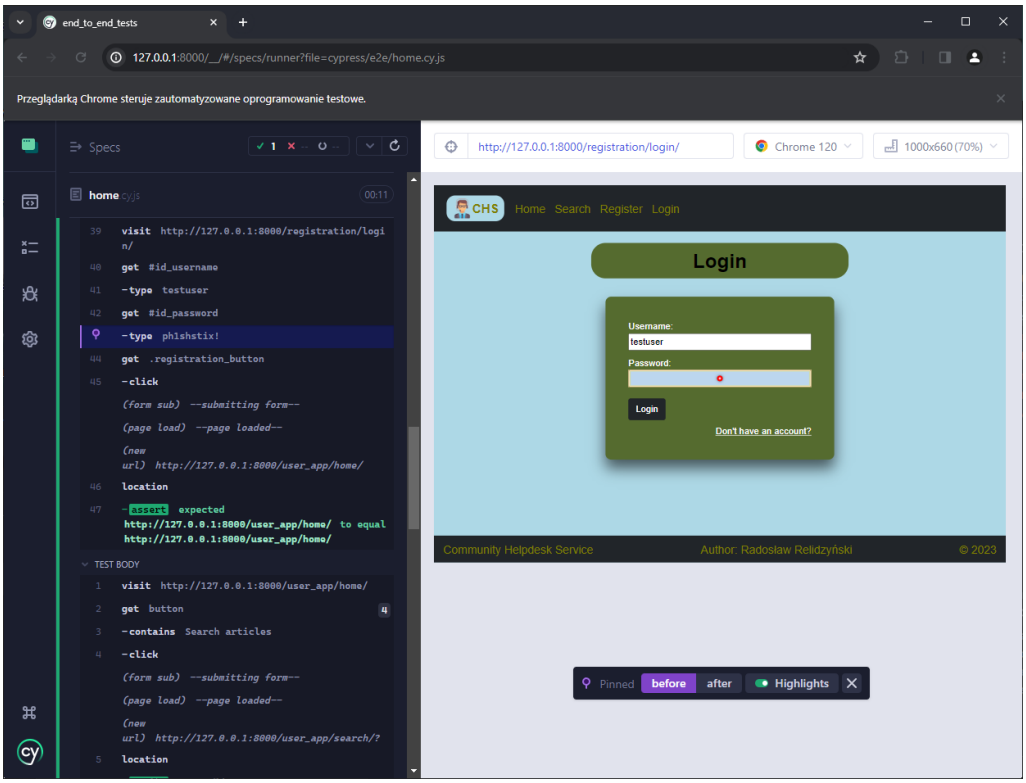


Rys. 29. Widok wywołania testu end-to-end dla testowania strony „home”

Jak widać na załączonym zrzucie ekranu, końcowym efektem testu jest widok strony logowania. Po prawej stronie widać wszystkie kolejne instrukcje oraz ich efekt. Można podejrzeć ten stan po kliknięciu w dowolną linijkę:



Rys. 30. Podgląd kroku w teście w momencie kliknięcia w przycisk



Rys. 31. Podgląd kroku w teście w momencie wypełniania pola tekstowego w teście

Dodatkowo, warto zaprezentować sposób załączania oraz sprawdzania obecności obrazów na stronie.

Tutaj załączanie zdjęcia do formularza:

```
cy.get('#id_additional_file').selectFile('cypress/fixtures/favicon.png', {force: true});
```

A tutaj sprawdzanie, czy obraz znajduje się na stronie:

```
cy.get('img[alt="Report Image"]')
    .should('exist')
    .and('have.attr', 'src')
    .and((src) => {
        expect(src).to.match(/favicon.*\.png$/);
    });
```

Zdjęcie jest walidowane wyrażeniem regularnym, gdyż system dodaje ciąg znaków na końcu nazwy pliku, jeśli nazwa ta już występuje w systemie.

Rezultaty wszystkich testów end-to-end znajdują się w folderze „end-to-end_test/videos” w postaci nagrań przedstawiających przebieg całego scenrariusza.

Podsumowanie

W ramach pracy dyplomowej udało się skutecznie utworzyć system pełniący funkcję społecznościowego serwisu helpdesk. System spełnia swoje założenia, odpowiada na potrzeby określone w wymaganiach i zawiera implementację przypadków użycia. Dzięki wykorzystaniu frameworka Django system posiada dużo istotnych zabezpieczeń przeciwko złośliwym działaniom, a dzięki narzędziu PyCharm i dobrych praktyk kod jest dobrej jakości, zoptymalizowany i przejrzysty. Dzięki podejściu TDD system posiada też duży zasób testów zapewniający poprawne działanie funkcjonalności systemu.

W trakcie realizacji projektu pojawiały się różne wyzwania wynikające w głównej mierze ze złożoności frameworka Django. Miejscami wymagały one zapoznania się z operacjami, jakie dane funkcjonalności wywoływały, co udało się przy pomocy dobrej dokumentacji na stronie Django. Innego rodzaju wyzwania pojawiły się przy projektowaniu interfejsu ze względu na konieczność zachowania czytelności wyświetlanych treści, co można było rozwiązać na kilka sposobów. Ostatecznie udawało się określić ostateczne podejście i taki interfejs powstał.

Aplikacja zdecydowanie nie posiada swojej ostatecznej formy. Można ją dalej rozwijać dodając kolejne funkcjonalności, między innymi można by dodać komentowanie wpisów, pobieranie wpisów w postaci dokumentów czy dodanie możliwości zalogowania z wykorzystaniem kont mailowych. Dzięki dużej elastyczności frameworka i skutecznie zastosowanej architekturze MVT dodawanie nowych przypadków użycia do systemu jest ułatwione.

Moim zdaniem projekt jest w pełni zdatny do wdrożenia go w życie. W ramach niego zaistnieje wspólna platformę do wymiany doświadczenia w zakresie korzystania z komputera. Mam nadzieję, że użytkownicy tego systemu nabędą nowe umiejętności, które będą mogli zastosować w wyzwaniach codziennego życia.

Bibliografia

Źródła pogrupowane są pod względem analizowanego zagadnienia, poprzedzone są datą ich wykorzystania

O organizacji ZHP:

[1] (19.04.2023) https://pl.wikipedia.org/wiki/Zwi%C4%85zek_Harcerstwa_Polskiego

[2] (19.04.2023) https://zhp.pl/

O specyfikacji wymagań:

[3] (10.05.2023) https://wolski.pro/2009/10/metoda-punktw-przypadkw-uzycia/

[4] (10.05.2023) https://pl.wikipedia.org/wiki/Wymaganie_(in%C5%BCynieria)

[5] (10.05.2023) https://visuresolutions.com/pl/blog/functional-requirements/

[6] (10.05.2023) https://pl.wikipedia.org/wiki/Wymagania_systemowe

O diagramach (use case, UML, BPMN)

[7] (17.05.2023) https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/

[8] (17.05.2023) https://www.visual-paradigm.com/guide/bpmn/how-to-use-data-objects-in-bpmn/

[9] (17.05.2023) https://www.lucidchart.com/pages/pl/notacja-i-model-procesu-biznesowego

[10] (17.05.2023) https://wolski.pro/diagramy-uml/diagram-przypadkw-uzycia/

O podejściu zwinnym

[11] https://pl.wikipedia.org/wiki/Manifest_Agile

O frameworku Django

[12] (24.05.2023) https://www.djangoproject.com/

[13] (24.05.2023) https://www.youtube.com/watch?v=sm1mokevMWk

[14] (24.05.2023) https://www.w3schools.com/django/django_intro.php

O architekturze MVT

[15] (24.05.2023) https://www.geeksforgeeks.org/django-project-mvt-structure/

[16] (24.05.2023) https://www.javatpoint.com/django-mvt

[17] (24.05.2023) https://www.educative.io/answers/what-is-mvt-structure-in-django

O testowaniu w ramach frameworka Django

[18] (09.11.2023) https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Testing

[19] (09.11.2023) https://docs.djangoproject.com/en/4.2/topics/testing/overview/

O środowisku wytwórczym

[20] (26.10.2023) https://pl.wikipedia.org/wiki/PyCharm

[21] (26.10.2023) https://pl.wikipedia.org/wiki/Google_Chrome

[22] (26.10.2023) https://www.sempire.pl/co-to-jest-windows.html

[23] (26.10.2023) https://pl.wikipedia.org/wiki/Windows_10

[24] (26.10.2023) https://pl.wikipedia.org/wiki/Git_(oprogramowanie)

[25] (26.10.2023) https://www.jetbrains.com/help/pycharm/debugging-your-first-python-application.html

[26] (26.10.2023) https://docs.python.org/3/library/pdb.html

O stosie technologicznym

[27] (09.11.2023) https://pl.wikipedia.org/wiki/Python

[28] (09.11.2023) https://pl.wikipedia.org/wiki/Django_(framework)

[29] (09.11.2023) https://pl.wikipedia.org/wiki/PostgreSQL

[30] (09.11.2023) https://semcore.pl/czym-jest-html-i-co-warto-o-tym-wiedziec-podpowiadamy/

[31] (09.11.2023) https://miroslawzelent.pl/kurs-css/

[32] (09.11.2023) https://jinja.palletsprojects.com/en/3.1.x/

[33] (09.11.2023) https://pl.wikipedia.org/wiki/Jinja2

[34] (09.11.2023) https://pl.wikipedia.org/wiki/Bootstrap_(framework)

[35] (09.11.2023) https://getbootstrap.com/docs/5.3/getting-started/introduction/
[36] (09.11.2023) https://miroslawzelent.pl/kurs-css/
O uwierzytelnianiu i autoryzacji
[37] (23.11.2023) https://www.jakubkulikowski.pl/2020/10/13/uwierzytelnianie-authentication-vs-autoryzacja-authorization/
[38] (23.11.2023) https://docs.djangoproject.com/en/3.2/_modules/django/contrib/auth/
O bezpieczeństwie frameworka Django
[39] (23.11.2023) https://docs.djangoproject.com/en/4.2/topics/security/
[40] (23.11.2023) https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/web_application_security
[41] (23.11.2023) https://docs.djangoproject.com/en/3.2/topics/security/#cross-site-scripting-xss-protection
[42] (23.11.2023) https://cheatsheetseries.owasp.org/cheatsheets/Django_Security_Cheat_Sheet.html
O Test Driven Development
[43] (09.11.2023) https://forbot.pl/blog/czym-jest-test-driven-development-wprowadzenie-przyklady-id35473
[44] (09.11.2023) https://www.youtube.com/watch?v=YiJnbGzdSi8
O innych podejściach do tworzenia oprogramowania
[45] (09.11.2023) https://swimm.io/learn/documentation-tools/tips-for-creating-self-documenting-code
[46] (09.11.2023) https://stackoverflow.com/questions/209015/what-is-self-documenting-code-and-can-it-replace-well-documented-code
[47] (09.11.2023) https://jessicabaker.co.uk/2018/09/10/comment-free-coding/
Inspiracje dotyczące stylu
[48] (15.10.2023) https://codepen.io/IMarty/pen/zrdYGe
[49] (15.10.2023) https://codepen.io/gschier/pen/kyRXVx
[50] (15.10.2023) https://expose.pl/lista-rozwijana-w-ms-excel/
O platformie heroku
[51] (15.01.2024) https://pl.wikipedia.org/wiki/Heroku
[52] (15.01.2024) https://www.heroku.com/
[53] (15.01.2024) https://www.youtube.com/watch?v=2OHc5EqfX5g
O oprogramowaniu docker
[54] (15.01.2024) https://www.docker.com/
[55] (15.01.2024) https://pl.wikipedia.org/wiki/Docker_(oprogramowanie)
[56] (15.01.2024) https://youtu.be/W5Ov0H7E_o4?si=AuEyTKD8FcJCPORt
[57] (15.01.2024) https://help.statlook.com/pl/baza-wiedzy/start-stop-postgresql
[58] (15.01.2024) https://stackoverflow.com/questions/6153113/how-to-create-a-fixture-file
[59] (15.01.2024) https://docs.docker.com/samples/django/
[60] (15.01.2024) https://testdriven.io/blog/dockerizing-django-with-postgres-gunicorn-and-nginx/
[61] (15.01.2024) https://gdevillele.github.io/compose/django/
[62] (15.01.2024) https://github.com/peter-evans/docker-compose-healthcheck
O testach end-to-end oraz Cypressie
[63] (19.01.2024) https://www.cypress.io/
[64] (19.01.2024) https://www.wyszkolewas.com.pl/testy-end-to-end-e2e/
[65] (19.01.2024) https://docs.cypress.io/guides/getting-started/installing-cypress
[66] (19.01.2024) https://sii.pl/blog/cypress-dlaczego-warto-zainteresowac-sie-tym-frameworkiem/
[67] (19.01.2024) https://docs.cypress.io/guides/end-to-end-testing/testing-your-app
Źródła zaprezentowanych fragmentów biblioteki Django:
[68] (24.01.2024) https://github.com/django/django/blob/main/django/contrib/auth/__init__.py

Spis rysunków

Spis tabel