Dokumentacja do projektu wykonywanego w ramach zajęć Techniki Internetowe

System nauki Pythona

Wydział Fizyki i Informatyki Stosowanej
Mikhail Shupliakou
02.01.2025

Spis treści

1 Wprowadzenie i informacje o projekcie

1.1 Ogólny opis projektu

Projekt stanowi internetową platformę edukacyjną typu LMS (Learning Management System), dedykowaną nauce języka programowania Python. Aplikacja zapewnia użytkownikom ustrukturyzowany dostęp do materiałów szkoleniowych podzielonych na moduły i lekcje, a także umożliwia weryfikację nabytej wiedzy za pomocą interaktywnych testów (quizów).

System obsługuje model dostępu oparty na rolach (RBAC), rozdzielając funkcjonalności dla użytkowników standardowych (studentów) oraz administratorów (twórców treści).

1.2 Koncepcja

Głównym założeń projektu jest stworzenie dostępnego i intuicyjnego środowiska do nauki programowania "od zera". Platforma łączy warstwę teoretyczną (wykłady tekstowe z przykładami kodu) z częścią praktyczną, realizowaną poprzez system testowy.

1.3 Cele i zadania projektu

Cel główny: Opracowanie narzędzia webowego automatyzującego proces nauczania podstaw języka Python, zapewniającego efektywną interakcję między autorem kursu a uczącymi się.

Główne zadania projektu:

1. **Organizacja treści edukacyjnych:**
   - Wdrożenie hierarchicznej struktury danych: Kurs (Moduł) → Lekcja → Test.
   - Umożliwienie dodawania opisów oraz grafik (okładek) dla każdego modułu w celu poprawy nawigacji i estetyki.
2. **System kontroli wiedzy:**
   - Opracowanie mechanizmu tworzenia testów wielokrotnego wyboru.
   - Implementacja algorytmu automatycznego sprawdzania odpowiedzi oraz zapisu historii wyników (w celu uniemożliwienia wielokrotnego podchodzenia do zaliczonego testu).
3. **Interfejs użytkownika (UI/UX):**
   - Stworzenie responsywnego interfejsu (RWD) z obsługą ciemnego motywu (Dark Mode), co zwiększa komfort pracy z kodem.
   - Integracja biblioteki do kolorowania składni (Prism.js) w celu czytelnego wyświetlania fragmentów kodu Python.

1.4 Realizacja techniczna

Aplikacja została zrealizowana w oparciu o następujący stos technologiczny:

- **Język programowania:** Python 3.9 (logika serwera).

- **Backend framework:** Flask (routing, obsługa żądań HTTP).
- **Baza danych:** PostgreSQL (hosting w chmurze NeonDB).
- **Frontend:** HTML5, CSS3, JavaScript, Bootstrap 5.

1.4.1 Architektura systemu (MPA & MVC)

Zgodnie z wymaganiami projektowymi, system został zaimplementowany jako **aplikacja typu MPA (Multi-Page Application)**. Każda akcja użytkownika (np. przejście do nowej lekcji) wiąże się z wygenerowaniem nowej strony HTML po stronie serwera i przesłaniem jej do klienta.

Architektura aplikacji opiera się na wzorcu **MVC (Model-View-Controller)**:

- **Model:** Reprezentowany przez klasy w pliku `models.py` (SQLAlchemy), odpowiada za strukturę danych i interakcję z bazą PostgreSQL.
- **View (Widok):** Warstwa prezentacji realizowana jest za pomocą szablonów Jinja2 (pliki HTML w katalogu `templates`).
- **Controller (Kontroler):** Logika biznesowa zawarta w pliku `routes.py`, która przetwarza żądania użytkownika, komunikuje się z modelem i zwraca odpowiedni widok.

Dodatkowo, w celu poprawy responsywności (np. przy przesyłaniu wyników quizu), zastosowano asynchroniczne zapytania JavaScript przy użyciu metody `fetch()`, co pozwala na aktualizację fragmentu strony bez jej pełnego przeładowania.

1.5 Role i poziomy dostępu

W aplikacji zaimplementowano system uwierzytelniania i autoryzacji. Dostęp do poszczególnych zasobów zależy od przypisanej roli:

1. **Gość (Użytkownik niezalogowany):** Może przeglądać stronę główną, listę modułów oraz czytać treści lekcji. Nie ma dostępu do testów.
2. **User (Student):** Posiada uprawnienia gościa, a dodatkowo może rozwiązywać quizy oraz przeglądać swoje wyniki i historię odpowiedzi.
3. **Admin (Administrator):** Posiada pełne uprawnienia w systemie. Może tworzyć, edytować i usuwać moduły, lekcje oraz quizy poprzez dedykowany panel zarządzania.

2 Baza danych

Jako warstwę trwałości danych wykorzystano relacyjną bazę danych PostgreSQL, hostowaną w usłudze chmurowej NeonDB.

2.1 Architektura i ORM

Do komunikacji między aplikacją a bazą danych wykorzystano wzorzec ORM (Object-Relational Mapping) przy użyciu biblioteki SQLAlchemy. Pozwoliło to na abstrakcję warstwy danych i operowanie na obiektach Pythonowych zamiast na surowych zapytaniach SQL, co zwiększa bezpieczeństwo (ochrona przed SQL Injection) i czytelność kodu.

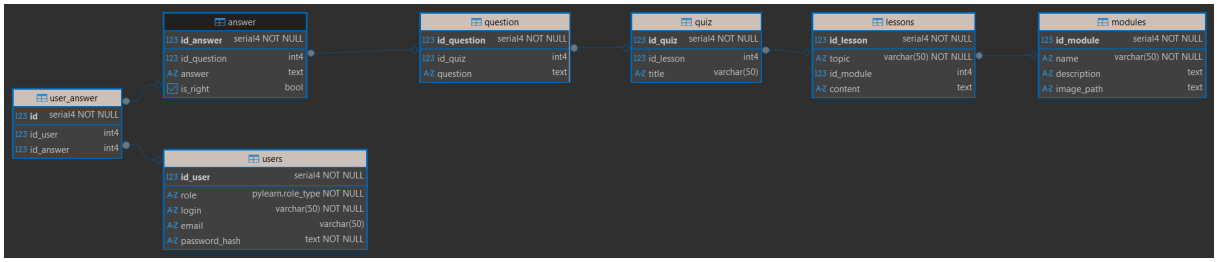
2.2 Struktura tabel i encji

Model danych składa się z siedmiu powiązanych tabel, podzielonych na trzy obszary logiczne:

1. **Zarządzanie użytkownikami**
   - Tabela *users*: Centralny punkt systemu autoryzacji. Przechowuje unikalne ID, login, email, hash hasła oraz rolę użytkownika ('admin' lub 'user').
2. **Struktura kursu** (Relacja jeden-do-wielu):
   - Tabela *modules*: Główne działy tematyczne. Zawiera nazwę, opis oraz ścieżkę do pliku graficznego (*image_path*).
   - Tabela *lessons*: Lekcje przypisane do modułów (*id_module*). Treść lekcji (*content*) przechowywana jest jako kod HTML generowany przez edytor WYSIWYG.
3. **System weryfikacji wiedzy (Quizy)**:
   - Tabela *quiz*: Łączy zestaw pytań z konkretną lekcją. Relacja 1:1 z lekcją.
   - Tabela *question*: Przechowuje treść pytania, powiązana z quizem.
   - Tabela *answer*: Warianty odpowiedzi dla pytania. Pole logiczne *is_right* wskazuje poprawną odpowiedź.
   - Tabela *user_answer*: Tabela łącząca (junction table) zapisująca postępy. Przechowuje pary: ID użytkownika i ID wybranej odpowiedzi.

2.3 Diagram związków encji (ERD)

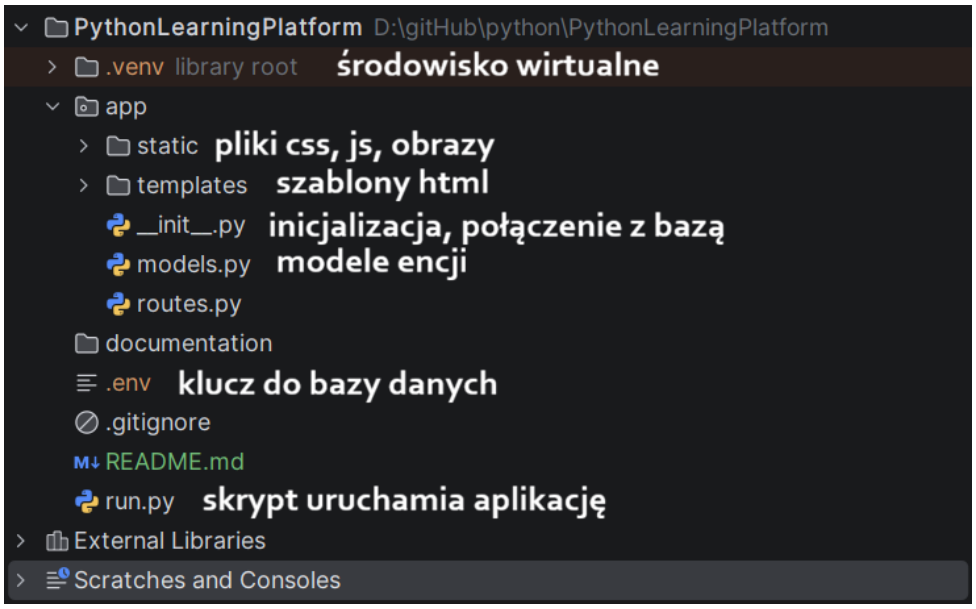
Poniższy diagram przedstawia fizyczny model bazy danych z uwzględnieniem typów danych oraz kluczy obcych.



Rysunek 1: PyLearn - diagram ERD

3 Struktura projektu

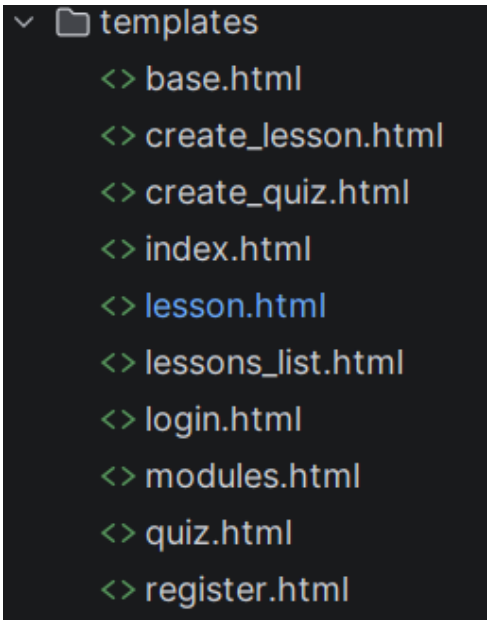
Projekt został zrealizowany przy użyciu środowiska IDE PyCharm. Poniższy zrzut ekranu prezentuje strukturę plików i katalogów aplikacji:



Rysunek 2: Drzewo katalogów projektu

Opis kluczowych elementów:

- Katalog **static**/: Zawiera zasoby statyczne – arkusze stylów (CSS), skrypty JavaScript oraz grafiki (np. logo, okładki modułów).
- Katalog **templates**/: Przechowuje szablony HTML wykorzystywane przez silnik Jinja2.
- Plik **__init__.py**: Inicjalizuje aplikację Flask i konfiguruje połączenie z bazą danych.
- Plik **models.py**: Definiuje klasy modeli mapowane na tabele bazy danych (SQLAlchemy).
- Plik **routes.py**: Zawiera definicje widoków (endpoints) oraz logikę sterującą aplikacją.
- Plik **run.py**: Punkt wejścia aplikacji (entry point).



Rysunek 3: Zawartość folderu templates

4 Opis funkcjonalności i instrukcja obsługi

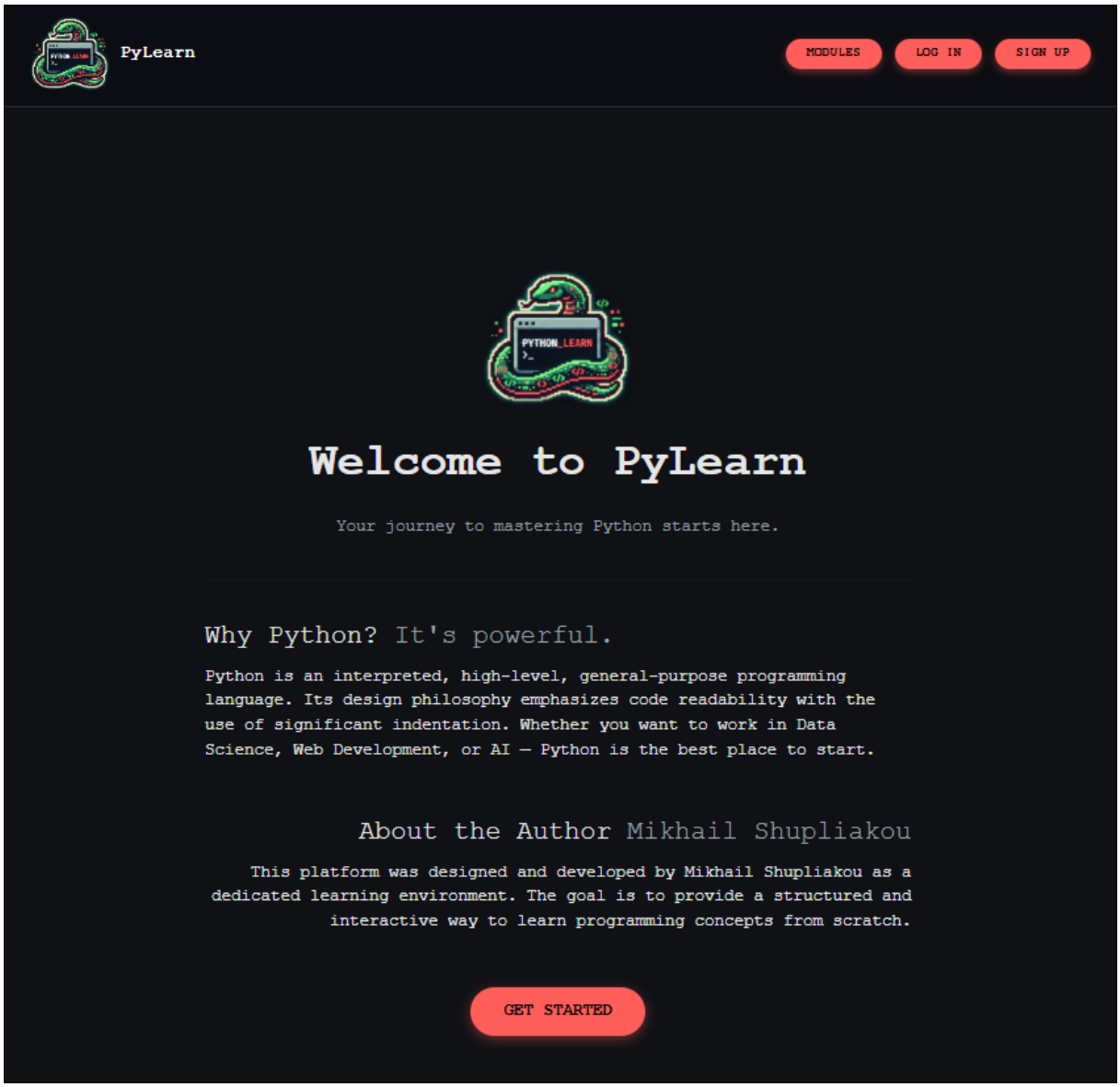
W warstwie prezentacji zastosowano mechanizm dziedziczenia szablonów (ang. *template inheritance*). Plik *base.html* definiuje szkielet strony (nagłówek, nawigację, stopkę), a treść poszczególnych podstron jest dynamicznie wstrzykiwana w blok `content`.

```
<main class="page-wrapper">
  {% block content %}{% endblock %}
</main>
```

4.1 Dostęp publiczny (Bez logowania)

4.1.1 Strona główna

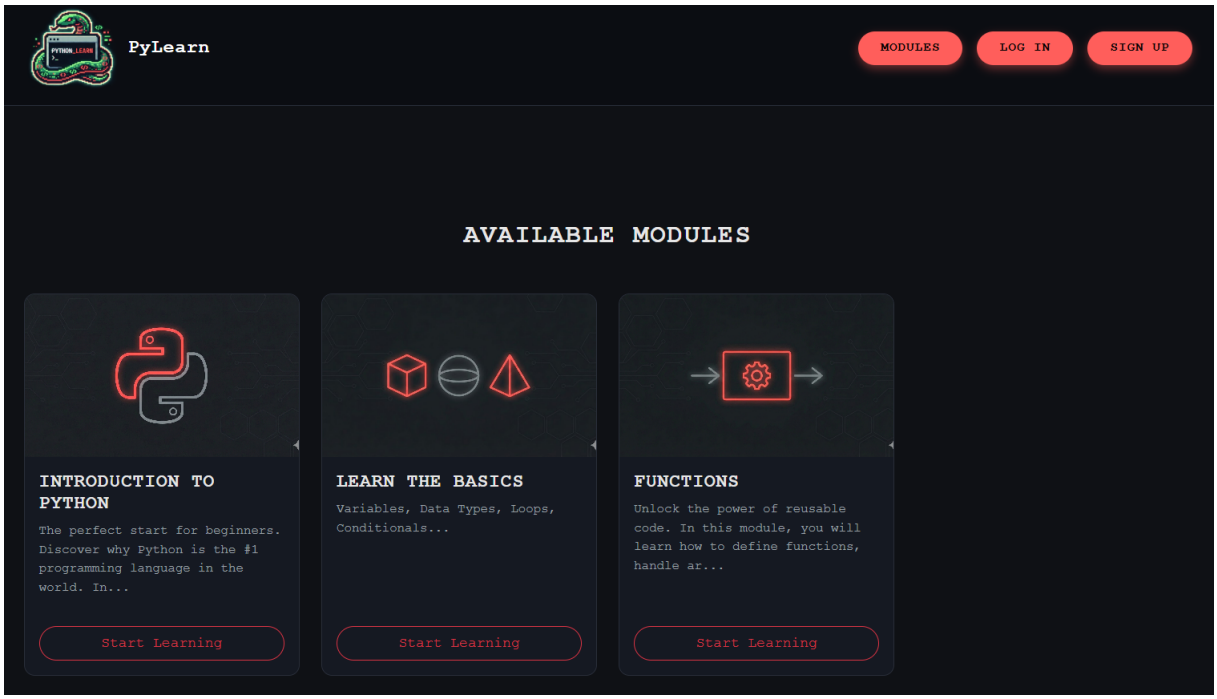
Strona startowa (*index.html*) wita użytkownika informacjami o platformie. Przycisk "GET STARTED"kieruje bezpośrednio do listy dostępnych modułów szkoleniowych.



Rysunek 4: Strona główna aplikacji

4.1.2 Przegląd modułów

Na stronie *modules.html* kursy prezentowane są w formie kafelków ("kart"). Każdy moduł posiada nazwę, opis oraz grafikę. Wybór modułu przenosi użytkownika do listy lekcji.



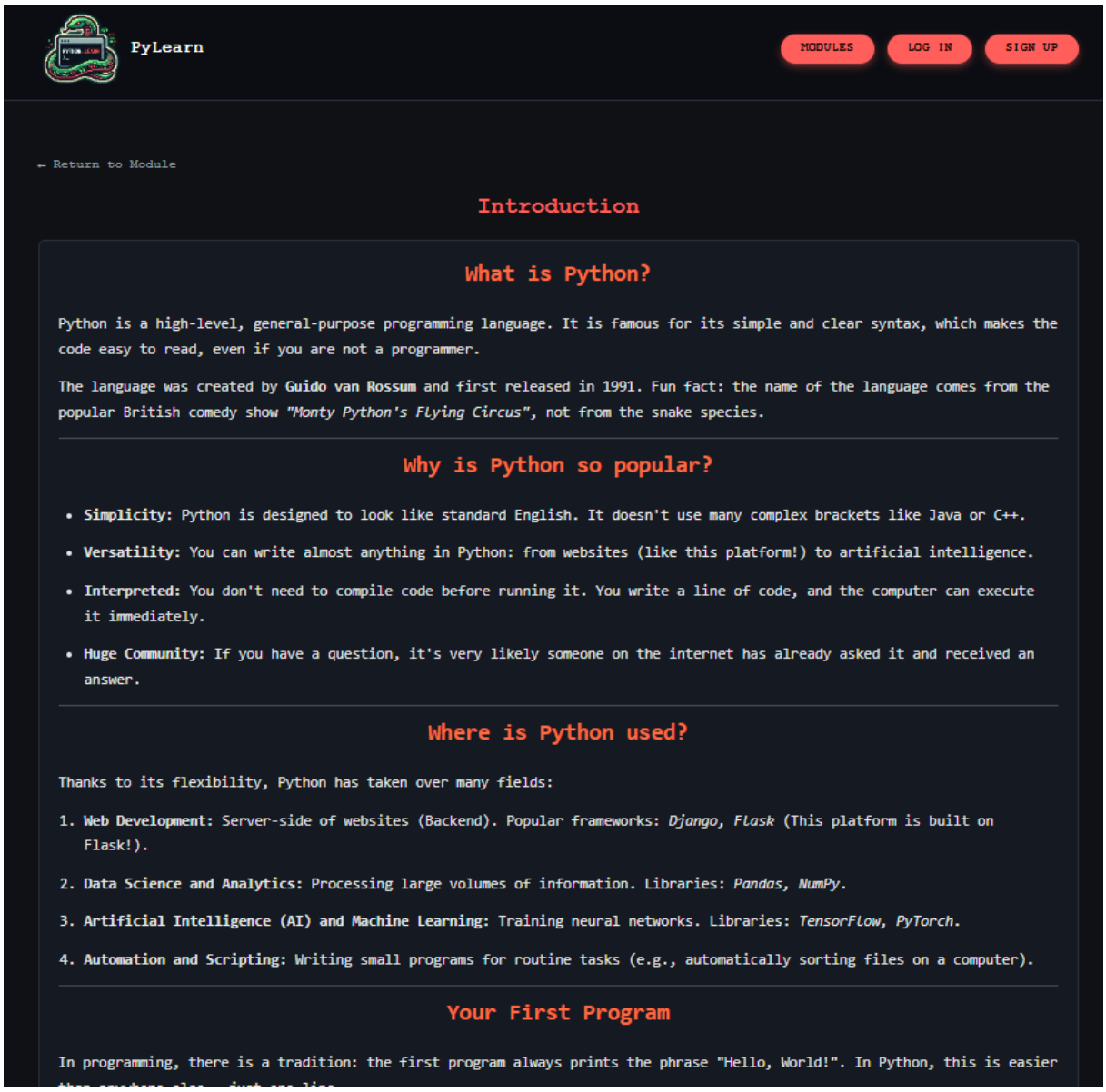
Rysunek 5: Lista dostępnych modułów

4.1.3 Widok lekcji

Po wybraniu konkretnego tematu (*lessons_list.html*), a następnie lekcji (*lesson.html*), użytkownik otrzymuje dostęp do materiałów dydaktycznych. Treść lekcji jest renderowana z kodu HTML przechowywanego w bazie danych.

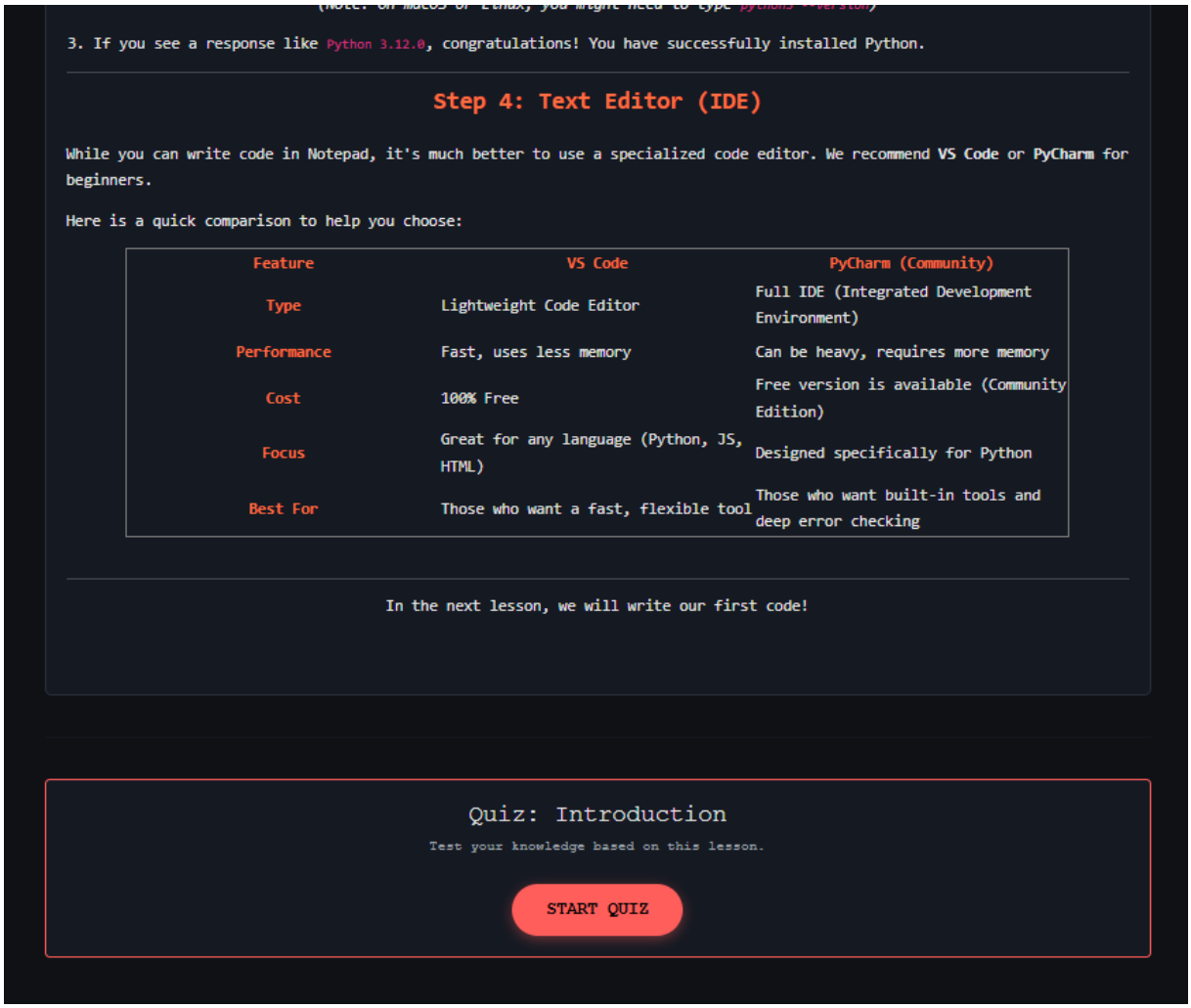
```
<h1 class="mb-4" style="color: var(--accent);">{{ lesson.topic }}</h1>

<div class="lesson-content p-4" style="...">
    <div class="lesson-text" style="...">
        {{ lesson.content | safe }}
    </div>
</div>
```

Filtr `| safe` informuje silnik szablonów, że pobrany kod HTML jest bezpieczny i należy go wyrenderować, a nie wyświetlać jako tekst.



Rysunek 6: Widok pojedynczej lekcji

Pod treścią lekcji znajduje się przycisk prowadzący do testu sprawdzającego wiedzę. Dostęp do niego jest jednak ograniczony dla niezalogowanych użytkowników.



Rysunek 7: Informacja o quizie

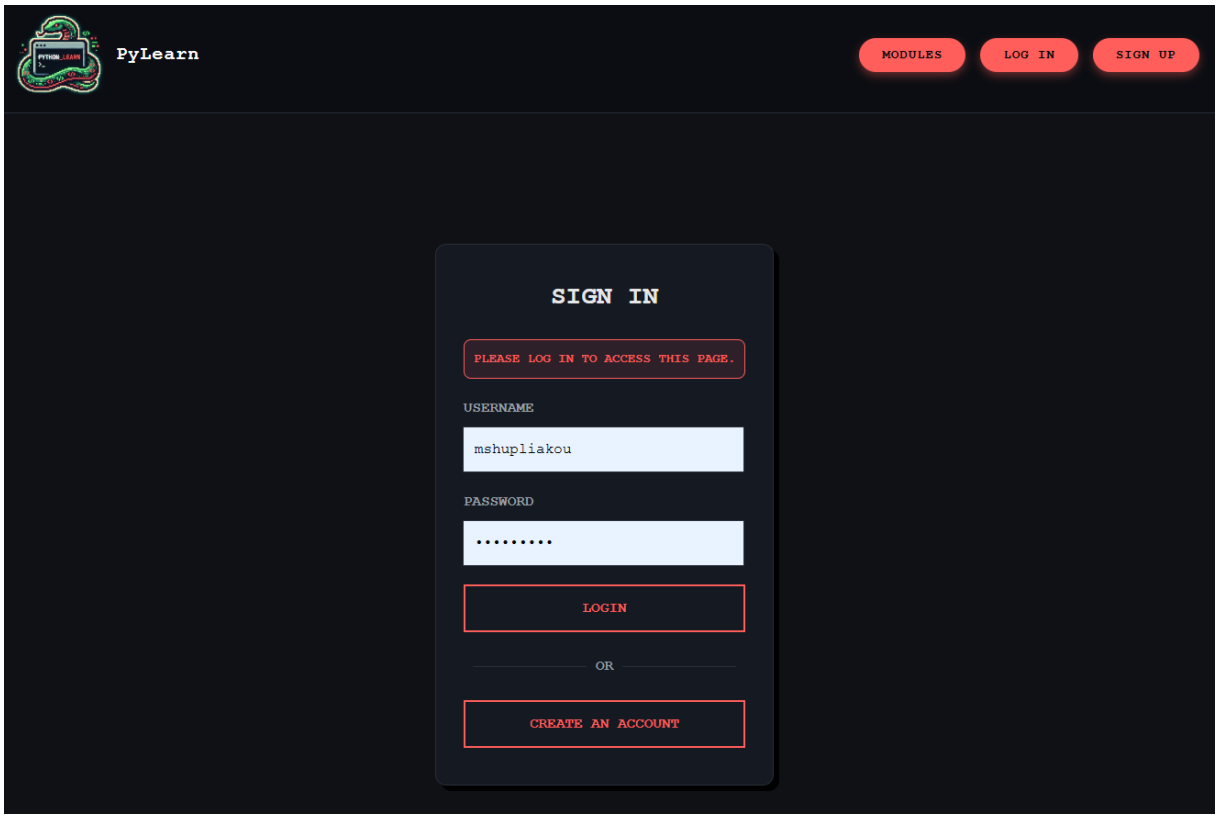
4.1.4 Rejestracja i logowanie (Zarządzanie sesją)

Aby uzyskać dostęp do funkcji interaktywnych, użytkownik musi posiadać konto. System realizuje funkcjonalność stanu za pomocą **zmiennych sesyjnych**.

- **Rejestracja:** Wykorzystuje bibliotekę `werkzeug.security` do bezpiecznego haszowania haseł przed zapisem w bazie.
- **Zarządzanie sesją:** Do obsługi uwierzytelniania wykorzystano rozszerzenie **Flask-Login**. Mechanizm ten zarządza sesją użytkownika poprzez pliki cookie, przechowując ID zalogowanego użytkownika i weryfikując jego uprawnienia przy każdym żądaniu do chronionych zasobów (dekorator `@login_required`).

```python
# Weryfikacja hasła przy logowaniu
def check_password(self, password):
    return check_password_hash(self.password_hash, password)

# Zabezpieczenie widoku (wymagana sesja)
@app.route('/quiz/<int:quiz_id>')
@login_required
def quiz_view(quiz_id):
```

```
    # ...
```



Rysunek 8: Formularz logowania

4.2 Funkcjonalności użytkownika (Rola: User)

Zalogowany użytkownik zyskuje możliwość przystępowania do testów weryfikacyjnych.

4.2.1 Rozwiązywanie quizów

Interfejs quizu składa się z pytań jednokrotnego wyboru (zazwyczaj trzy opcje). Po zaznaczeniu odpowiedzi dla wszystkich pytań, użytkownik zatwierdza test. System uniemożliwia ponowne podejście do raz rozwiązanego quizu, aby zapobiec manipulacji wynikami.

Rysunek 9: Interfejs quizu

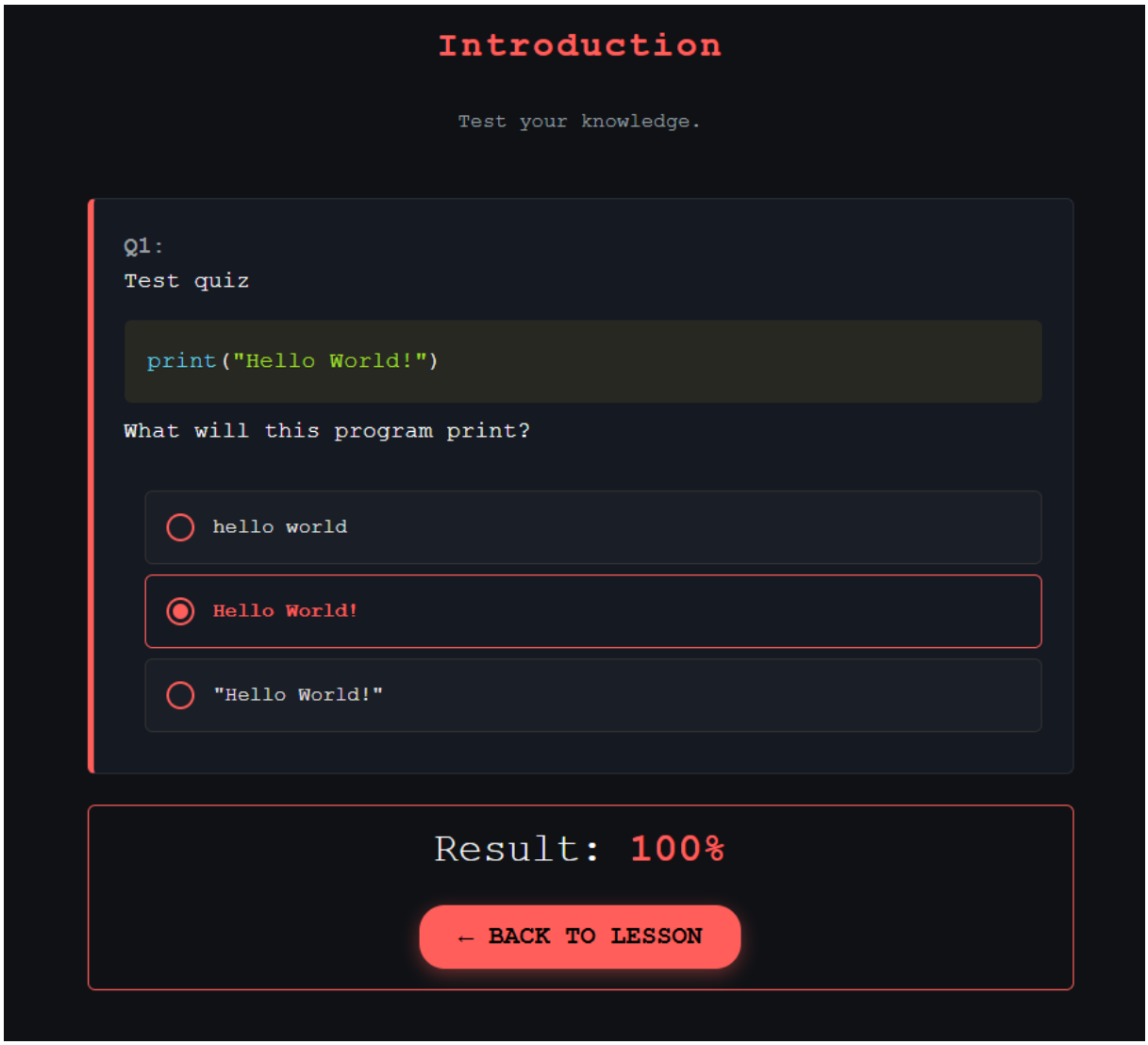
4.2.2 Wyniki i weryfikacja

Po zatwierdzeniu odpowiedzi, wynik jest obliczany po stronie serwera i prezentowany użytkownikowi w formie procentowej. System wskazuje również, które odpowiedzi były poprawne, a gdzie popełniono błąd.

Logika obliczania wyniku:

```
# Iteracja przez pytania w quizie
for q in quiz.questions:
    for a in q.answers:
        # Sprawdzenie, czy użytkownik wybrał tę odpowiedź
        ua = User_Answer.query.filter_by(
            id_user=current_user.id_user,
            id_answer=a.id_answer
        ).first()

        if ua:
            user_answers_ids.append(a.id_answer)
            already_taken = True
            if a.is_right:
                score += 1
```

```
percentage = round((score / total) * 100) if total > 0 else 0
```



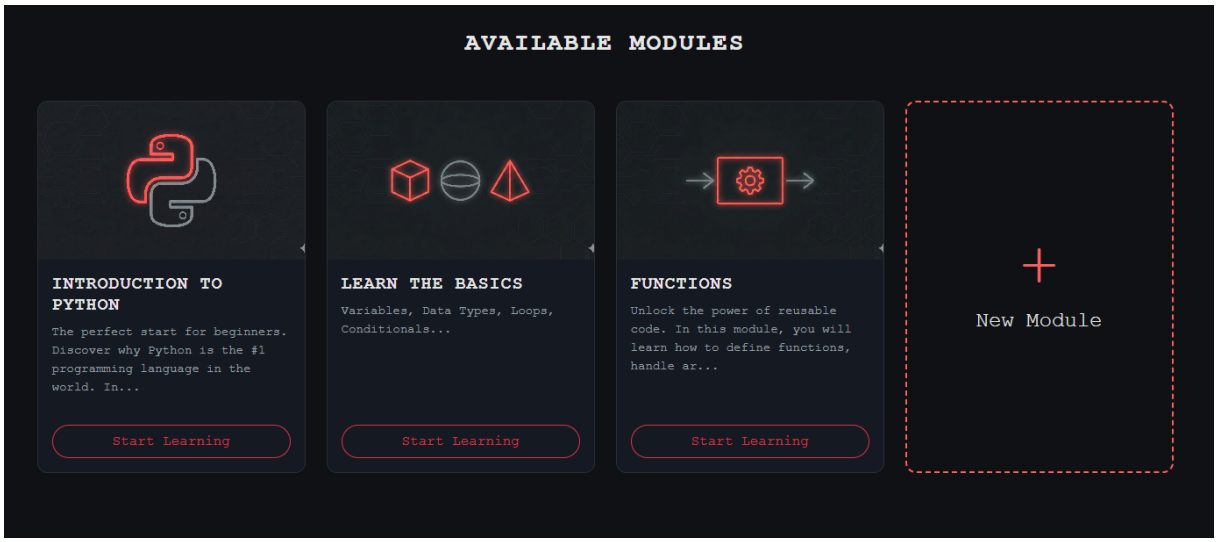
Rysunek 10: Prezentacja wyniku quizu

4.3 Funkcjonalności administratora (Rola: Admin)

Konto administratora posiada rozszerzone uprawnienia umożliwiające zarządzanie treścią (CRUD - Create, Read, Update, Delete). Dostęp do tych funkcji jest chroniony poprzez weryfikację pola `role` w obiekcie bieżącego użytkownika (UserMixin).

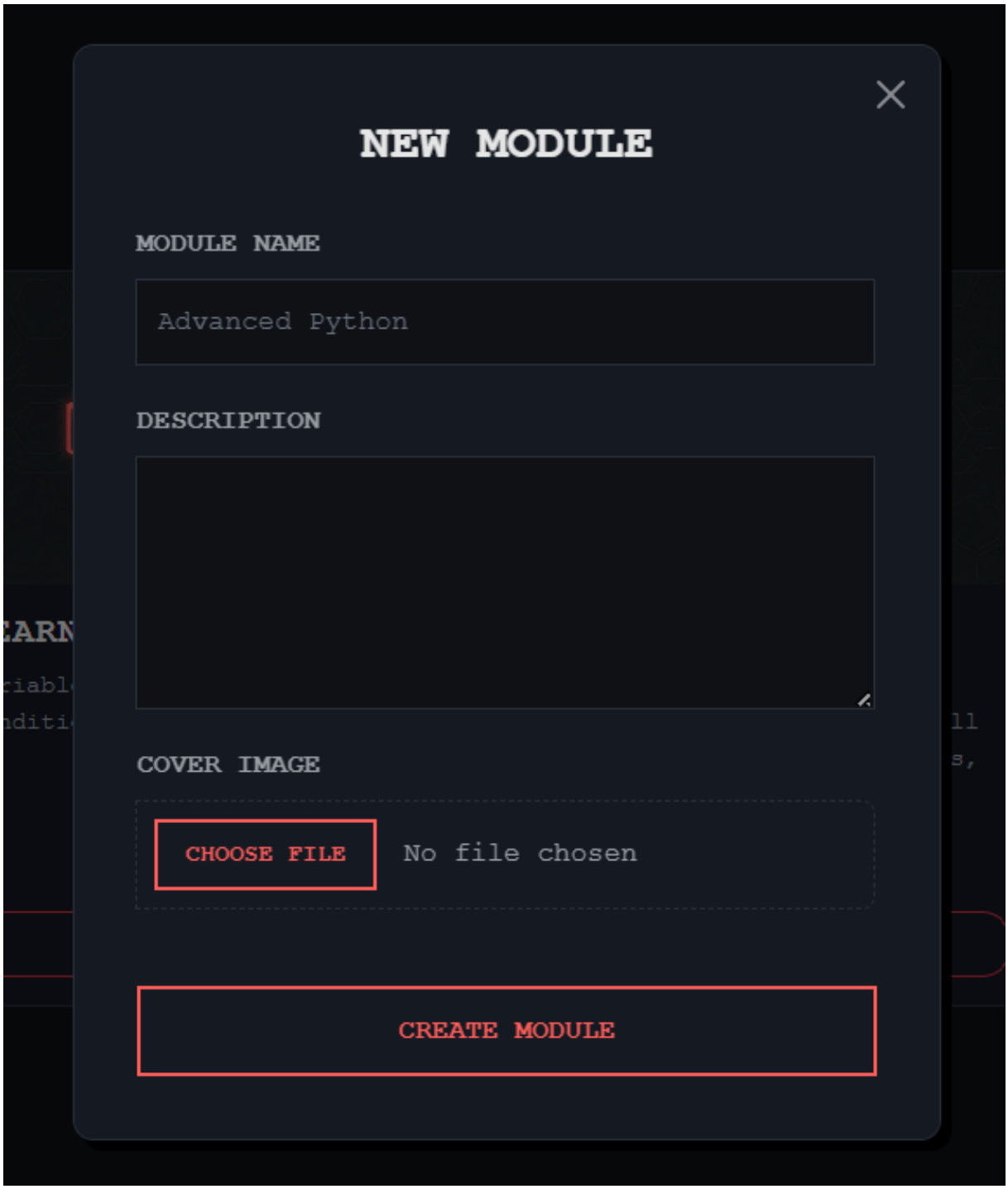
4.3.1 Zarządzanie modułami

Administrator widzi dodatkowe przyciski sterujące: "Nowy moduł"oraz opcje edycji/usuwania przy istniejących modułach.



Rysunek 11: Widok administratora - zarządzanie modułami

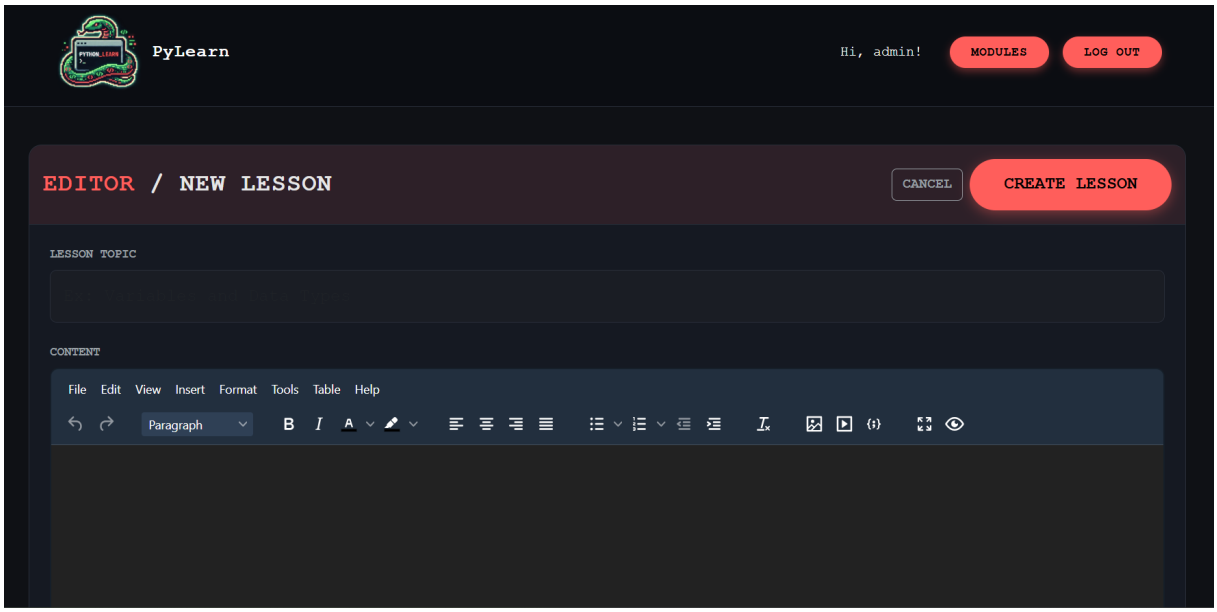
Dodawanie modułu odbywa się poprzez okno modalne, gdzie definiuje się nazwę, opis oraz przesyła plik graficzny (logo). Pliki graficzne są zapisywane na serwerze, a w bazie danych przechowywana jest tylko ścieżka do pliku.



Rysunek 12: Formularz tworzenia nowego modułu

4.3.2 Edytor lekcji (TinyMCE)

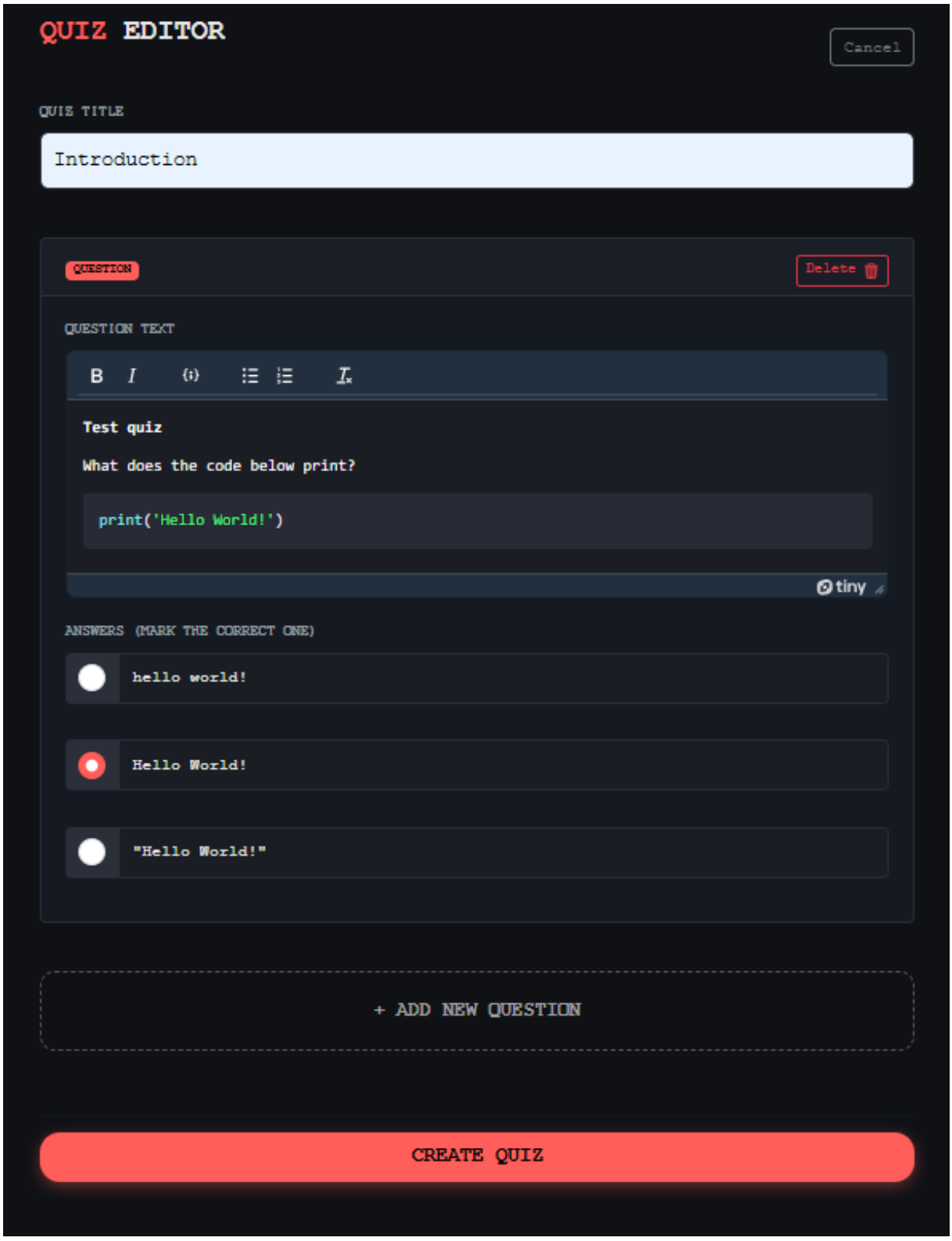
Do tworzenia i edycji treści lekcji zintegrowano zaawansowany edytor tekstu WYSIWYG – **TinyMCE**. Rozwiązanie to pozwala na formatowanie tekstu, wstawianie bloków kodu, tabel i obrazów bez konieczności znajomości języka HTML. Dzięki temu zarządzanie kursami jest dostępne dla osób nietechnicznych. Wygenerowany przez edytor kod HTML jest zapisywany bezpośrednio w bazie danych.



Rysunek 13: Edytor TinyMCE podczas tworzenia lekcji

4.3.3 Kreator quizów

Każda lekcja może posiadać jeden powiązany quiz. Kreator quizów (*create_ quiz.html*) umożliwia dynamiczne dodawanie pytań i definiowanie odpowiedzi. Administrator musi wskazać jedną poprawną odpowiedź dla każdego pytania.



Rysunek 14: Formularz tworzenia pytań i odpowiedzi do quizu

5 Walidacja projektu

Celem weryfikacji jakości kodu oraz zgodności ze standardami sieciowymi, projekt został poddany procesowi walidacji. Wykorzystano do tego celu oficjalne narzędzie konsorcjum

W3C – *Markup Validation Service*.

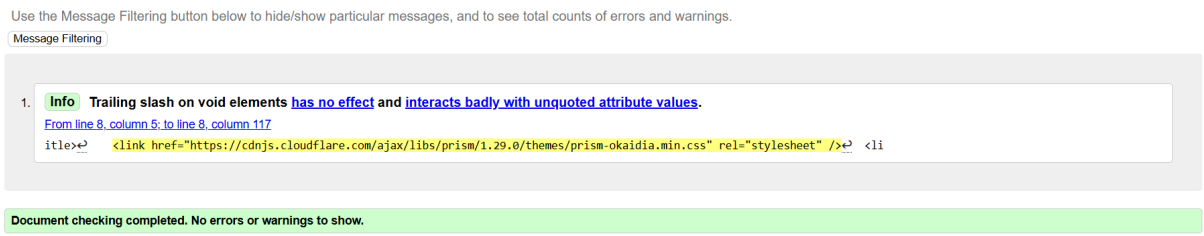
Ze względu na architekturę aplikacji (Flask + Jinja2), gdzie kod HTML jest generowany dynamicznie po stronie serwera, walidacji nie poddawano plików źródłowych szablonów (zawierających logikę programistyczną), lecz wynikowy kod HTML wyrenderowany przez przeglądarkę. Zastosowano metodę *Validate by Direct Input*.

5.1 Wyniki walidacji struktury aplikacji

Kluczowe widoki aplikacji, odpowiadające za szkielet i interfejs użytkownika, przeszły walidację pomyślnie. Przetestowano następujące podstrony:

- Strona główna (*index.html*),
- Panel logowania i rejestracji,
- Lista modułów i widok quizu.

Wszystkie elementy strukturalne szablonu bazowego (*base.html*), takie jak nawigacja, stopka oraz kontenery główne, są w pełni zgodne ze standardem HTML5.



Rysunek 15: Potwierdzenie poprawnej walidacji strony głównej

5.2 Walidacja treści dynamicznych (Lesson Content)

Specyficznym przypadkiem w procesie walidacji jest widok lekcji (*lesson.html*). Strona ta składa się z dwóch warstw:

1. **Warstwa szablonu (kod programisty):** Definiuje strukturę strony, nawigację oraz kontenery na treść. Ta część kodu jest statyczna i waliduje się poprawnie.
2. **Warstwa treści (kod autora kursu):** Treść merytoryczna lekcji jest pobierana z bazy danych i wstrzykiwana do szablonu. Treść ta jest tworzona przez administratorów za pomocą edytora tekstu bogatego (WYSIWYG) – TinyMCE.

Należy zaznaczyć, że ewentualne ostrzeżenia walidatora dotyczące samej treści lekcji (np. *"Heading follows h1, skipping h2"*) wynikają ze sposobu formatowania tekstu przez autora kursu w edytorze, a nie z błędów programistycznych aplikacji.

Na przykład, jeśli twórca kursu wewnątrz edytora użyje nagłówka stopnia 3 (H3) bezpośrednio po tytule, walidator zgłosi błąd semantyczny. Jest to jednak błąd danych (data entry error), niezależny od kodu źródłowego aplikacji. Platforma zapewnia poprawny kontener semantyczny (‘<section>‘) dla treści generowanych przez użytkowników.

5.3 Kompatybilność z przeglądarkami

Oprócz walidacji składniowej, aplikacja została przetestowana manualnie na najpopularniejszych silnikach przeglądarek internetowych:

- Google Chrome (silnik Blink),
- Mozilla Firefox (silnik Gecko),
- Microsoft Edge.

We wszystkich przypadkach układ graficzny (layout) oraz funkcjonalności interaktywne (JavaScript) działają poprawnie i w sposób identyczny.

6 Wdrożenie i uruchomienie aplikacji w środowisku chmurowym

W celu zapewnienia publicznego dostępu do platformy oraz demonstracji jej działania w rzeczywistych warunkach, aplikacja została wdrożona (zdeployowana) na platformie chmurowej **Render**. Jako system zarządzania bazą danych wykorzystano rozwiązanie chmurowe **NeonDB** (Serverless PostgreSQL).

6.1 Wybór technologii hostingu

Do realizacji części serwerowej wybrano dostawcę usług PaaS (Platform as a Service) – Render.com. Jest to nowoczesna platforma pozwalająca na szybkie wdrażanie aplikacji webowych bezpośrednio z repozytorium kodu. Wybór ten podyktowany był następującymi czynnikami:

- **Natywne wsparcie dla Pythona:** Platforma automatycznie wykrywa i konfiguruje środowisko uruchomieniowe dla aplikacji Flask.
- **Automatyzacja (CI/CD):** Integracja z systemem kontroli wersji GitHub pozwala na automatyczną aktualizację (przebudowanie i restart) aplikacji po każdym zatwierdzeniu zmian (push) w repozytorium.
- **Skalowalność i bezpieczeństwo:** Łatwe zarządzanie zasobami oraz bezpieczne przechowywanie zmiennych środowiskowych (sekretów).

6.2 Przygotowanie aplikacji do wdrożenia

Aby zapewnić poprawne działanie aplikacji w infrastrukturze chmurowej, wprowadzono następujące modyfikacje techniczne:

1. **Konfiguracja serwera WSGI:** Zamiast wbudowanego serwera Flask, przeznaczonego wyłącznie do celów deweloperskich, wdrożono profesjonalny serwer WSGI – **Gunicorn**. Zapewnia on wielowątkowe przetwarzanie żądań oraz stabilność pod obciążeniem.
2. **Zarządzanie zależnościami:** Utworzono plik `requirements.txt`, zawierający listę wszystkich niezbędnych bibliotek (Flask, SQLAlchemy, Gunicorn, Psycopg2-binary) wraz z ustalonymi wersjami, co gwarantuje kompatybilność środowisk.

3. **Konfiguracja zmiennych środowiskowych:** Ze względów bezpieczeństwa poufne dane (ciąg połączenia z bazą danych, sekretne klucze sesji) zostały usunięte z kodu źródłowego. W pliku `__init__.py` zaimplementowano logikę odczytu konfiguracji ze zmiennych środowiskowych (Environment Variables):

```
# Dostosowanie ciągu połączenia dla kompatybilności z SQLAlchemy
database_url = os.environ.get('DATABASE_URL')
if database_url and database_url.startswith("postgres://"):
    database_url = database_url.replace("postgres://", "postgresql://", 1)

app.config['SQLALCHEMY_DATABASE_URI'] = database_url
```

6.3 Proces wdrożenia

Proces publikacji aplikacji obejmował następujące etapy:

1. Przesłanie finalnej wersji kodu źródłowego do zdalnego repozytorium GitHub.
2. Utworzenie nowej usługi typu *Web Service* w panelu zarządzania Render i powiązanie jej z repozytorium.
3. Konfiguracja polecenia budowania (`pip install -r requirements.txt`) oraz polecenia uruchamiania (`gunicorn run:app`).
4. Wprowadzenie zmiennych środowiskowych `DATABASE_URL` oraz `SECRET_KEY` w bezpiecznej sekcji ustawień hostingu.

W rezultacie aplikacja została pomyślnie uruchomiona. Architektura rozwiązania zapewnia trwałość danych w chmurowej bazie NeonDB, niezależnie od restartów serwera aplikacji.

6.4 Dostęp do aplikacji

Wdrożona wersja platformy PyLearn jest publicznie dostępna w sieci Internet pod następującym adresem URL:

https://pylearn-mshupliakou.onrender.com

7 Źródła i technologie

1. NeonDB – Serverless Postgres.
2. GitHub – Repozytorium kodu źródłowego projektu.
3. TinyMCE – Edytor tekstu rich-text (WYSIWYG).
4. Flask Documentation – Dokumentacja frameworka.
5. W3C – Walidacja
6. Render – Chmura Obliczeniowa
7. PyLearn – Gotowy projekt w chmurze