UNIWERSYTET
IM. ADAMA MICKIEWICZA
W POZNANIU

Wydział Matematyki i Informatyki

Damian Kuich
Numer albumu: 434735

Mkołaj Kowalczyk
Numer albumu: 434725

Mateusz Michalski
Numer albumu: 416128

Gen-mat - platforma do generowania sprawdzianów z matematyki

Gen-mat - platform for generating math tests

Praca inżynierska na kierunku **informatyka**
napisana pod opieką
dr. Bartłomieja Przybylskiego

Poznań, 4 kwietnia 2022 r.

Poznań, 4 kwietnia 2022 r.

Oświadczenie

Ja, niżej podpisany **Damian Kuich**, student Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt. *Gen-mat - platforma do generowania sprawdzianów z matematyki* napisałem samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy lub jej części innym osobom, ani nie odpisywałem tej rozprawy lub jej części od innych osób. Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej. Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego stanowi podstawę stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego.

[TAK] wyrażam zgodę na udostępnianie mojej pracy w czytelni Archiwum UAM

[TAK] wyrażam zgodę na udostępnianie mojej pracy w zakresie koniecznym do ochrony mojego prawa do autorstwa lub praw osób trzecich

Poznań, 4 kwietnia 2022 r.

Oświadczenie

Ja, niżej podpisany **Mkołaj Kowalczyk**, student Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt. *Gen-mat - platforma do generowania sprawdzianów z matematyki* napisałem samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy lub jej części innym osobom, ani nie odpisywałem tej rozprawy lub jej części od innych osób. Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej. Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego stanowi podstawę stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego.

[TAK] wyrażam zgodę na udostępnianie mojej pracy w czytelni Archiwum UAM

[TAK] wyrażam zgodę na udostępnianie mojej pracy w zakresie koniecznym do ochrony mojego prawa do autorstwa lub praw osób trzecich

Poznań, 4 kwietnia 2022 r.

Oświadczenie

Ja, niżej podpisany **Mateusz Michalski**, student Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt. *Gen-mat - platforma do generowania sprawdzianów z matematyki* napisałem samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy lub jej części innym osobom, ani nie odpisywałem tej rozprawy lub jej części od innych osób. Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej. Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego stanowi podstawę stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego.

[TAK] wyrażam zgodę na udostępnianie mojej pracy w czytelni Archiwum UAM

[TAK] wyrażam zgodę na udostępnianie mojej pracy w zakresie koniecznym do ochrony mojego prawa do autorstwa lub praw osób trzecich

Streszczenie

Przedmiotem pracy jest przedstawienie komponentów składających się na projekt platformy webowej do generowania sprawdzianów z matematyki. W trakcie tworzenia serwisu musieliśmy podejmować wiele decyzji związanych z użyciem dostępnych rozwiązań technologicznych do rozwiązywania pojawiających się problemów. Pierwsza część ma za zadanie wprowadzenie w podstawowe struktury organizacyjne i technologiczne, który były wykorzystywane przy tworzeniu aplikacji. Poruszona została kwestia, jaki cel ma nasza aplikacja spełniać i jakie założenia zostały ustalone. Opisana została organizacja pracy i metodyka, która pomagała nam przy tworzeniu kolejnych etapów serwisu, oraz podział prac pomiędzy członków grupy deweloperskiej. W drugiej części przedstawiona została idea tworzenia serwisów na podstawie architektury REST. Zaczynając od podstawowych pojęć związanych z budową serwisów webowych, a kończąc na technologii, która została wykorzystana do implementacji owej architektury do naszej aplikacji. Trzecia część pracy opowiada o wybranych systemach bazodanowych, ich plusach oraz minusach. Następnie opisana jest integracja Django z PostgreSQL, czyli między innymi połączenie z bazą danych, stworzenie modeli dla danych oraz jak poprawnie wykonywać zapytania, aby nie narażać bazy na zbyt duże obciążenia. Ostatni rozdział opisuje javascriptową bibliotekę React, jej historię, specyfikację oraz użycie w procesie budowania interfejsu projektu Gen-Mat. Rozdział ten dodatkowo przybliża biblioteki Yup, Axios oraz Formik wykorzystywane przy tworzeniu oraz wymianie formularzy.

Spis treści

Rozdział 1

Wprowadzenie

1.1. Cel i założenia projektu

„Gen-Mat" to nowy serwis webowy do generowania i publikacji sprawdzianów z matematyki na poziomie szkół ponadpodstawowych. Jego głównym zadaniem jest uproszczenie procesu ich tworzenia. Nauczyciel zostanie odciążony z wymyślania i pisania zadań do sprawdzianu. Do jego dyspozycji przeznaczone są zadania dostępne w naszej bazie danych. Czynność, którą nauczyciel musi wykonać, to wyszukanie poszczególnych zadań oraz przeniesienie ich na sprawdzian. Atutem naszego generatora jest możliwość podzielenia sprawdzianu na grupy, dzięki czemu ograniczone zostanie oszukiwanie podczas pisania sprawdzianu. Kolejną wygodną opcją jest pominięcie wyboru zadań i postawienie na automatyczny wybór generatora. Jeżeli nauczycielowi nie spodobają się zadania wybrane przez serwis, może je w każdej chwili zamienić na inne. Aktualnie baza danych składa się wyłącznie z zadań zaczerpniętych z arkuszy Centralnej Komisji Egzaminacyjnej, ponieważ wymyślanie zadań przez założycieli serwisu wiązałoby się z bardzo dużym nakładem pracy oraz ryzykiem niskiej jakości zadań pod względem merytorycznym jak i edukacyjnym. Aby wyeliminować ryzyko niewystarczającej ilości zadań, wprowadzona została możliwość dodawania zadań bezpośrednio przez nauczycieli, aby każdy z nich mógł stworzyć swój idealny sprawdzian. W trosce o prywatność naszych użytkowników, każdy z nich będzie miał możliwość decydowania czy chce podzielić się z innymi nauczycielami stworzonym przez siebie zadaniem. Zadania dodane do bazy danych będą również weryfikowane przez osobę do tego przeznaczoną, aby nie zawierały treści, które mogą być nieodpowiednie w jakikolwiek sposób. Zespół podzielony jest na dwie grupy. Jedna z nich odpowiedzialna jest za Frontend, a druga za Backend. Zespół obrał metodykę pracy Scrum. Zadania są przypisywane przez Scrum Mastera na podstawie

umiejętności danego członka zespołu. Następnie, co jakiś czas sprawdzany jest postęp danego zadania na podstawie tzw. „Task Review". Ostatnim punktem jest testowanie, czy zadanie zostało wykonane w sposób poprawny, oraz czy nie pojawiły się jakiekolwiek błędy. Głównym założeniem projektu było stworzenie aplikacji webowej pozwalającej na generowanie sprawdzianów w łatwy i szybki sposób.

1.2. Podział pracy inżynierskiej

Kolejne rozdziały pracy dyplomowej opisują zagadnienia, z którymi zespół zapoznał się podczas tworzenia projektu inżynierskiego. Pierwszy rozdział pt. **„Implementacja REST API z wykorzystaniem frameworka Django REST"** autorstwa Damiana Kuicha opisuje metodykę REST, jak również technolgie z niej korzystające. Przedstawione są w niej podstawowe definicje potrzebne do zrozumienia całości pracy. Rozdział ten opisuje paradygmaty jakie musi spełniać aplikacja by móc być nazwana RESTful. W dalszej części przedstawiona jest technologia Django, jak i framework Django REST, oraz korelacja frameworka z ideą REST.

Mikołaj Kowalczyk w rozdziale pt. **„Integracja projektu opartego o Django z bazą danych PostgreSQL"** porusza temat baz danych oraz integracji frameworka Django z wybranym systemem bazodanowym. Rozdział odpowie między innymi na pytanie, czym jest baza danych. Zawarta w nim będzie również charakterystyka systemów bazodanowych MySQL, MS SQL Server, Oracle i PostgreSQL. W rozdziale tym opisany zostanie proces modelowania oraz tworzenia elementów bazy danych za pomocą Django, oraz sposób w jaki można się z nią komunikować. Wszelkie przykłady opierać się będą na podstawie bazy utworzonej na potrzeby projektu. Opisane zostaną wybrane modele oraz relacje między tabelami, dzięki którym projekt zachowuje spójną całość. Rozdział zawiera informacje na temat dostępu do danych oraz jak je pozyskiwać w sposób optymalny, nie narażający bazy na zbyt duże obciążenie.

Rozdział pt. **„React jako narzędzie do tworzenia interfejsu użytkownika"** autorstwa Mateusza Michalskiego opisuje proces tworzenia interfejsu graficznego aplikacji webowej. Opowiada o Reakcie - javascriptowej bibliotece przeznaczonej do budowania interfejsów, przybliża jak tworzyć oraz używać reactowe komponenty. Ponadto, przedstawia sposoby wymiany danych między użytkownikiem a serwerem z perspektywy frontendu oraz technologie ułatwiające obsługę formularzy.

1.3. Opis Technologii

Serwis „Gen-Mat" korzysta z szerokiego zakresu technologii. Produkt oparty jest w szczególności na dwóch językach: Pythonie w wersji 3 oraz JavaScriptcie. Głównym czynnikiem, który kierował autorami przy wyborze technologii było doświadczenie członków zespołu przy projektowaniu systemów wykorzystujących dane języki programowania. Równie ważnym aspektem była prostota implementacji założeń projektowych i funkcjonalności. Podstawą projektu jest serwer oparty na serwisie Heroku. Umożliwia on szybkie i łatwe wdrażanie serwisu na etap produkcyjny, zachowując przy tym wysoką jakość świadczonych usług hostingowych. Projekt, jako aplikacja webowa, podzielony został na dwie główne części. Część backendową oraz część frontendową. Backend odpowiedzialny jest za to, czego użytkownik korzystający z serwisu nie widzi. Są to czynności takie jak, na przykład, naprawa i utrzymywanie bazy danych, obsługa użytkowników, obsługa tokenów, spajanie zadań do utworzenia sprawdzianów itp. Wykorzystuje on Pythonowy framework Django, który umożliwia w łatwy i prosty sposób stworzenie wysokiej jakości produktu. W pierwszej iteracji produktu podstawową bazą danych była MariaDB, jednakże wraz z rozwojem została ona zmieniona. Aktualnie serwis korzysta z baz danych udostępnionych na Heroku, które operują na silniku PostgreSQL. Frontend odpowiedzialny jest za część wizualną projektu. Obsługuję on na przykład wszystkie zapytania wysyłane przez użytkowników, przetwarzanie danych wysyłanych z bazy danych i wyświetlanie ich, komunikację z bazą danych za pomocą API, łączenie się z serwerem za pomocą HTTP, obsługę sesji użytkownika i tym podobne. Ta część projektu wykorzystuje przede wszystkim Javascriptową bibliotekę React. Projekt korzysta również z frameworka Django REST do tworzenia RESTful API.

Rozdział 2

Implementacja REST API z wykorzystaniem frameworka Django REST

2.1. Wstęp

Głównym celem tego podrozdziału jest przedstawienie podstawowych definicji oraz postawienie fundamentów do dalszego zrozumienia pracy.

2.1.1. HTTP

HTTP (ang. *Hypertext Transfer Protocol*) jest protokołem przesyłania dokumentów hipertekstowych. Jest to podstawowy protokół sieci **WWW** (ang. *World Wide Web*), za pomocą którego przesyła się żądania. HTTP jest protokołem, który umożliwia stronom WWW publikowanie informacji.

2.1.2. HTTP Request

Żądanie (ang. *Request*) w formie HTTP powinno posiadać cztery elementy:

— **Metode HTTP** (ang. *Request method*) - określenie metody, którą chcemy wykorzystać dla danego zasobu.
— **Nagłówek** (ang. *Header*) - może składać się z kilku części i ma za zadanie przechowywać dodatkowe informacje o żądaniu.
— **Ścieżkę do zasobu** (ang. *Resource path*) - URI, czyli ścieżka dostępu do zasobu.
— **Ciało** (ang. *Body*) - Składuje się tu dodatkowe informacje, które później wykorzystywane są do edycji, lub tworzenia zasobów po stronie serwera. Może być puste.

Wyciąg 2-1. Przykład żądania HTTP

```
POST news.com/articles
Accept: application/json
Body:
{
   "article":{
      "title": "Some title",
      "author": "Damian Kuich",
      "text": "Something happened again",
      "date": "25/02/2019",
   }
}
```

2.1.3. Metody HTTP

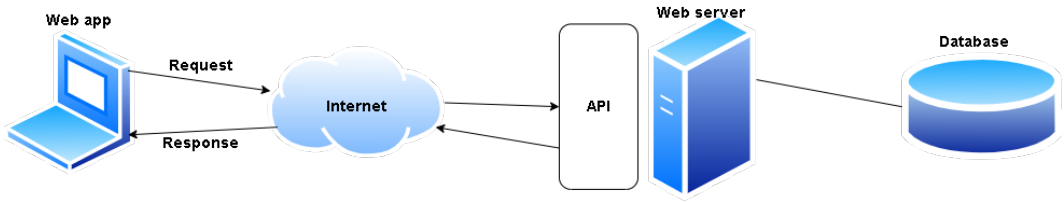
Metoda HTTP jest częścią zapytania HTTP odpowiedzialną za określenie akcji, która ma zostać podjęta w wyniku zapytania HTTP. Istnieje siedem aktualnie najczęściej wykorzystywanych metod:

— **GET** - Odczytuje reprezentację zasobu i powinno wyłącznie zwracać dane.
— **POST** - Wykorzystywane w celu przesyłania danych do wskazanego zasobu.
— **PUT** - Aktualizuje całą reprezentację zasobu z wykorzystaniem przesłanych danych.
— **PATCH** - Aktualizuje część reprezentacji zasobu z wykorzystaniem przesłanych danych.
— **DELETE** - Usuwa wskazany zasób lub kolekcje zasobów.
— **HEAD** - Pełni identyczną rolę jak GET, jednak w tym przypadku odpowiedź nie zawiera ciała, lecz nagłówki.
— **OPTIONS** - Zwraca informację o możliwych opcjach dostępnych na danym zasobie.

2.1.4. Czym w zasadzie jest API ?

API (ang. *Application Programming Interface*) możemy najprościej przyrównać do rozmowy dwóch osób. Kiedy chcemy uzyskać informację od drugiej osoby, wysyłamy do niej **prośbę** (ang. *Request*) o nią, a ona zwraca nam **odpowiedź** (ang. *Response*) zawierającą pewne dane. API systematyzuje sposób rozmowy

pomiędzy dwoma stronami. Istnieje wiele rodzajów API, jednakże w tej pracy skupimy się na Web API, czyli API dla serwisów internetowych.



Rysunek 2.1. Zasada działania API

2.1.5. Co to jest URI ?

URI (ang. *Uniform Resource Identifier*) to ciąg znaków identyfikujący zasób, który, na chwilą obecną, dla uproszczenia będziemy utożsamiać z pewną informacją. Istnieje pięć głównym charakterystyk URI:

— Ciąg znaków powinien być **zwięzły**, czyli nie może składać się ze spacji, tabulatorów i innych zbędnych znaków.

— Nasza informacja może być **materialna**, czyli odwoływać się do elementu świata rzeczywistego np. budynku, osoby, ale także **abstrakcyjna**, czyli odwoływać się do, przykładowo, usługi czy dokumentu sieci Web.

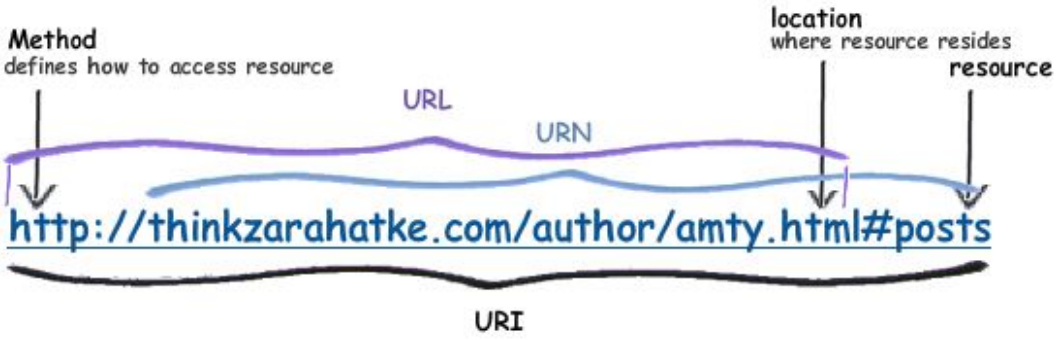
— Identyfikator URI jest na stałe przypisany do danego zasobu. Powinien być **stabilny**, czyli nie zmieniać się i nie znikać.

— Użytkownik może **odwołać się** do identyfikatora URI i otrzymać treściwą informację zwrotną.

— Identyfikatory dokumentów sieci Web muszą być **jednoznaczne**, czyli nie powinny być mylone z identyfikatorami innych zasobów.

URI składa się z dwóch komponentów. Pierwszym z nich jest **URL** (ang. *Uniform Resource Locator*), który odnosi się do podzbioru URI. Oprócz identyfikacji zasobów, zapewnia on środki do ulokowania zasobu i opisuje podstawowy mechanizm dostępu (np. jego "położenie" w sieci). **URN** (ang. *Uniform Resource Name*) to podzbiór identyfikatorów URI, które zawierają nazwę w danej przestrzeni,

ale nie zawierają lokalizacji. Dokumentacja mówi nam też, że każdy lokator może być też identyfikatorem, czyli **każdy URL może być URI, ale istnieją URI, które nie są URL**.



Rysunek 2.2. Zasada działania URI, URL i URN
Źródło: https://prateekvjoshi.com/2014/02/22/url-vs-uri-vs-urn/

2.1.6. Endpoints

Upraszczając, **punkt końcowy** (ang. *Endpoint*) to jeden koniec kanału komunikacyjnego. Gdy interfejs API współdziała z innym systemem, punkty tej komunikacji są traktowane jako punkty końcowe. W przypadku interfejsów API punkt końcowy może zawierać adres URL serwera lub usługi. Każdy punkt końcowy to lokalizacja, z której interfejsy API mogą uzyskać dostęp do zasobów potrzebnych do wykonywania swoich funkcji. Gdy interfejs API żąda informacji z aplikacji internetowej lub serwera, otrzymuje odpowiedź. Miejsce, do którego interfejsy API wysyłają żądania i gdzie znajdują się zasoby, nazywane są punktami końcowymi.

Wyciąg 2-2. Przykładowe Endpointy

```
1 https://somewebsite.org/name/{nameId} #możemy wstawić cały link URL
2 /name/{nameId} #ale nie musimy
```

2.2. REST oraz REST API

2.2.1. Historia REST i REST API

W latach dziewięćdziesiątych nie istniały jeszcze żadne ustandaryzowane metody tworzenia API oraz ich używania. Najczęściej używanym protokołem był **SOAP** (ang. *Simple Object Access Protocol*), który mimo posiadania w swej definicji słowo „*Simple*"", był niezwykle skomplikowany do tworzenia, przetwarzania i debugowania. Wymagał on ręcznego utworzenia pliku XML z wywołaniem **RPC** (ang. *Remote Procedure Call*) w body. Po napisaniu należało sprecyzować endpoint, przesłać metodą POST definicje w **kopercie** (ang. *SOAP envelope*) z informacją, co znajduje się w wiadomości, dla kogo jest przeznaczona i czy jest ona obowiązkowa lub opcjonalna do danego endpointa.

Wyciąg 2-3. Przykładowe żądanie SOAP API

```
<?xml version="1.0"?>
<SOAP:Envelope
  xmins.xsi="http://www.w3.org/1999/XMLSchema/instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema/instance"
  xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">
 <SOAP:Body>
    <calculateArea>
     <origin>
      <x xsd:type="float">10</x>
      <y xsd:type="float">20</y>
     </origin>
     <corner>
      <x xsd:type="float">100</x>
      <y xsd.type="float">200<iy>
     </corner>
    </calculateArea>
  </SOAP:Body>
</SOAP:Envelope>
```

Wszystko zmieniło się w 2000 roku, kiedy to grupka deweloperów pod kierownictwem **Roya Feldinga** postawiła sobie za cel utworzenie standardu, pozwalającego na to, by każde dwa serwery mogły porozumiewać się pomiędzy sobą w prosty sposób. W tym samym roku, ów lider wydał rozprawę doktorską pt. „*Architectural Styles and the Design of Network-based Software*", która była

zbiorem zasad nowej architektury nazwanej REST. Nowy standard opierał się na siedmiu głównych filarach:

— **Wydajność** - interakcje pomiędzy komponentami powinny być natychmiastowe.

— **Skalowalność** - umożliwiająca obsługę dużej ilości komponentów.

— **Prostota** - ujednolicenie interfejsu.

— **Modyfikowalność** - możliwość komponentów do zmian w celu sprostania zmieniającym się potrzebom.

— **Widoczność** - przejrzysta komunikacja między komponentami.

— **Przenośność** - możliwość przenoszenia kodu wraz z danymi.

— **Niezawodność** - odporność na awarie na poziomie systemu.

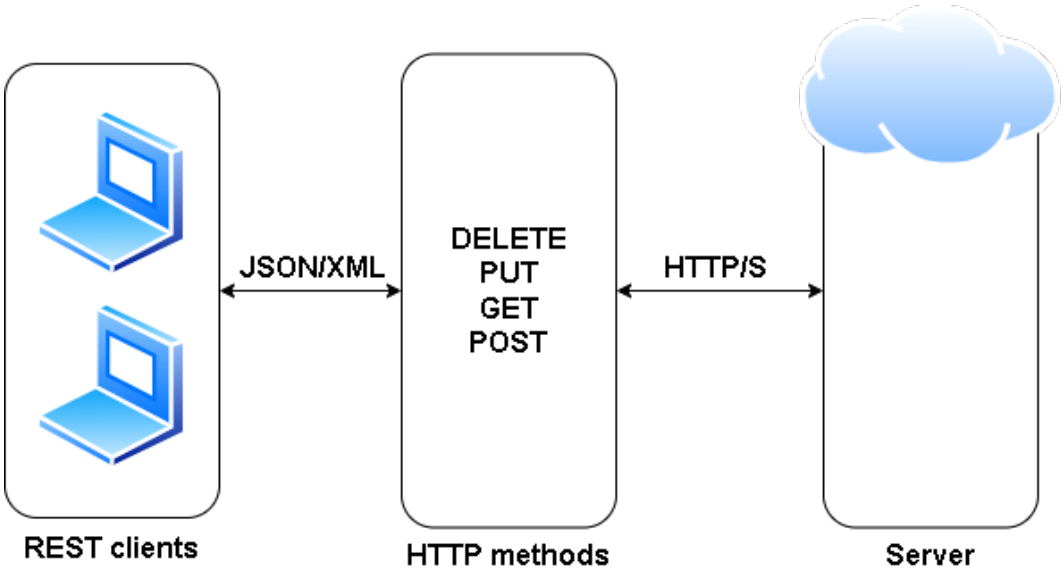
REST, czyli *The Representational State Transfer* jest niczym innym, jak architekturą dla systemów webowych, która prawie zawsze bazuje na protokole HTTP. Jest to zbiór pewnych dobrych praktyk, które mają działać jako drogowskaz przy tworzeniu skalowalnych aplikacji. **REST API** jest natomiast interfejsem definiującym w jaki sposób użytkownik może skomunikować się z systemem, uzyskać dostęp do zasobów, oraz w jakim formacie je otrzymuje, którego to kształt jest zdefiniowany przez architekturę REST.

2.3. Zasady REST

Systemy oparte na architekturze REST nazywamy RESTful, jeżeli spełniają pewne wymagania. Systemy RESTful charakteryzują się tym, że są bezstanowe (ang. stateless) i mają mocno podkreśloną granicę pomiędzy Klientem, a Serwerem. Roy Fielding podał sześć głównych cech, które powinny zawierać systemy oparte o architekturę REST:

— **Uniform Interface** - Serwer powinien udostępniać jednolite API, które będzie klarowne dla wszystkich systemów komunikujących się z nim. Dane powinny być w takim samym formacie i zawierać identyczny zakres. Interfejs powinien spełniać wymogi dla wielu urządzeń i aplikacji. Zasoby w

systemie powinny mieć jeden Uniform Resource Identifier, który pozwalałby na wymianę danych i powinny być łatwo dostępne przez np. HTTP request GET.



Rysunek 2.3. Zasada działania REST API opartej na architekturze REST

Zasada ta zawiera jeszcze cztery reguły, który pomagają nam uprościć architekturę:

— **Resource identification in requests** - Sposób reprezentacji zasobu może być różny i odłączny od samego zasobu np. serwer nie wysyła danych wprost odczytanych z bazy tylko w formacie JSON albo XML.

— **Resource manipulation through representations** - Po otrzymaniu reprezentacji zasobów wraz z jego metadanymi od serwera, klient (jeśli ma do tego uprawnienia) może modyfikować oraz usuwać zasoby.

— **Self-descriptive messages** - Odpowiedź serwera powinna zawierać wystarczającą ilość informacji, aby serwer mógł ją obsłużyć np. poprzez wartość „media type”(inaczej „MIME type”).

— **Hypermedia as the engine of application state** (HATEOAS) - HATEOAS jest ograniczeniem architektury aplikacji REST, która sprawia, że architektura stylu RESTful jest inna od większości innych architektur. Termin „hypermedia” odnosi się do wszelkich treści zawierających łącza do innych form mediów, takich jak obrazy, filmy i teksty. Styl architektoniczny

REST pozwala nam używać linków hipermedialnych w treści odpowiedzi. Pozwala klientowi dynamicznie przechodzić do odpowiednich zasobów korzystając z łącz hipermedialnych. Najlepiej zobrazować to na przykładzie Graph API od Facebooka np. korzystając z przykładowej odpowiedzi znajdującej się w dokumentacji. Niektóre wartości zostały skrócone dla przejrzystości:

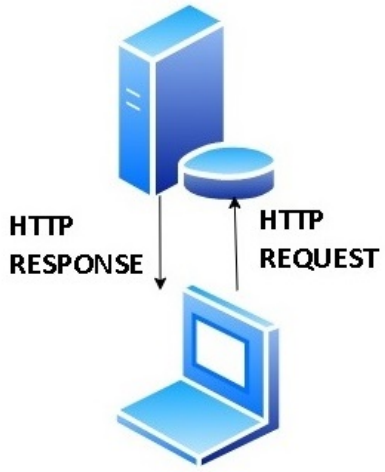
Wyciąg 2-4. Odpowiedź od Graph API

```
{
   "feed":{
      "data":[
         {
            "created_time":"2017-12-12T01:24:21+0000",
            "message":"This picture is nice",
            "id":"820882001277849_1809387339093972"
         },
         {
            "created_time":"2017-12-11T23:40:17+0000",
            "message":":)",
            "id":"820882001277849_1809316002434439"
         },
         {
            "created_time":"2017-12-11T23:31:38+0000",
            "message":"Thought you might enjoy this",
            "id":"820882001277849_1809310929101613"
         }
      ],
      "paging":{
         "cursors":{
            "before":"Q2c4U1pXNTBYM0YxWlhKNVgzTjBiM0",
            "after":"Q2c4U1pXNTBYM0YxWlhKNVgzTjBiM0"
         },
         "next":"https://graph.facebook.com/820882001277849/feed?access_token"
      }
   },
   "id":"820882001277849"
}
```

W sekcji paging, pod kluczem next widnieje link, który poprowadzi nas do kolejnych wyników.

— **Client-Serwer** - Projektując, a następnie implementując aplikacje klienta i serwera należy wyznaczyć widoczną granicę pomiędzy oba programami. Oznacza to, że klient i serwer mogą rozwijać się bez ingerencji pomiędzy sobą. Zamysł ten pomaga w realizacji ww. Uniform Interface, a także zwiększa możliwości skalowania i rozszerzalności, a obsługa błędów jest znacznie łatwiejsza. Jeżeli na przykład dokonamy zmiany w aplikacji pisanej dla systemu iOS (to będzie nasz klient), to zmiana ta nie powinna w żaden sposób

wpływać na aplikacje serwerową (strukturę bazy danych i tym podobne). Tak samo powinno to działać w drugą stronę.



Rysunek 2.4. REST w prosty sposób pomaga deweloperom w planowaniu interakcji pomiędzy niezależnymi systemami

— **Stateless** - według tego założenia wszystkie interakcje pomiędzy klientem, a serwerem powinny być bezstanowe. Oznacza to, iż serwer nie zapisuje żadnych stanów aplikacji klienta, a jedynie przetwarza informacje wysyłane przez niego. Każde żądanie powinno być przetwarzane jako nowe i zawierać komplet informacji. Jeżeli klient musi być stanowy, jak np. przy logowaniu, to każde żądanie powinno zawierać tylko potrzebne informacje, wraz z danymi do autoryzacji i uwierzytelniania.

— **Cacheable** - odpowiedź, którą użytkownik otrzyma z REST API musi jasno definiować, czy ma ona być cacheable czy non-cacheable. Ma to znaczenie przy danych, które bardzo szybko stają się nieaktualne oraz przy danych, które aktualizują się relatywnie rzadko – nie ma sensu na przykład cacheować współrzędnych geograficznych pędzącego samolotu, natomiast jego kolor czy nazwę już tak.

— **Layered System** - REST pozwala na odseparowanie warstw. Możemy wdrożyć nasze API na serwer A, przechowywać dane na serwerze B i przetwarzać żądania na serwerze C. Klient nie wie o zewnętrznych usługach i serwisach, z których korzysta serwer. Pozwala to na rozbicie serwisu na wiele mikroserwisów, które można łatwo aktualizować. Dzięki temu zachowujemy

transparentność i separację logiki od warstwy WWW. Niestety takie rozwiązanie ma pewne wady, ponieważ może wprowadzać opóźnienia przez potrzebę komunikacji pomiędzy różnymi częściami naszej aplikacji.

— **Code-On-Demand** - jest to zasada, która jest opcjonalna, ponieważ prawie zawsze będziemy wysyłać dane w statycznej formie, jak np. XML albo JSON. Nasz serwis może wysyłać gotowe fragmenty kodu np. skrypty JavaScript do ich przetworzenia.

2.4. Podstawowy element REST API, czyli zasoby

> The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service (e.g. „today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g., a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.
>
> (Roy Fielding)

2.4.1. Czym jest zasób?

Zasób (ang. *Resource*) jest elementarną częścią REST API. Wykorzystujemy go właśnie podczas interakcji z API. Czym więc jest zasób? Nie jest łatwo odpowiedzieć na to pytanie, ponieważ zasób to pojęcie abstrakcyjne. Dowolna informacja, która może posiadać nazwę, może być zasobem. Dlatego też przyjęto kilka podstawowych kryteriów, by można było nazwać daną informację zasobem:

— Musi być rzeczownikiem.
— Musi być unikatowa, czyli definiować konkretną rzecz i być jak najbardziej sprecyzowana. Przykładowo, budynek nie jest zasobem, ponieważ jest zbyt ogólny, lecz biblioteka już nim jest.
— Musi być możliwa do przedstawienia w formie danych.

— Music posiadać przynajmniej jeden adres URI, pod którym jest dostępny. URI jednocześnie jest nazwą, jak i adresem zasobu.

2.4.2. Nazewnictwo

Odpowiednie nazewnictwo i struktura informacji pozwala na zwiększenie użyteczności oraz elastyczności, ułatwiając rozbudowę i modyfikację aplikacji. Niestety sposób nazywania zasobów jest kwestią bardzo dyskusyjną i nie istnieje jedna zasada jak powinniśmy poprawnie nazywać zasoby. Jednakże istnieją wskazówki, które można zastosować:

— **Obiekty vs. Akcje** - Zasoby powinny być tak zaprojektowane by były odłączne od akcji. Dzięki temu możemy wykonywać wiele akcji na pojedynczym zasobie. Przytoczę taki przykład:

Wyciąg 2-5. Przykład pierwszy

```
1 GET /users # Gets list of all users
2 POST /users # Adds new user
3 DELETE /users # Deletes all users
```

Wyciąg 2-6. Przykład drugi

```
1 GET /getUsers # Gets list of all users
2 POST /addUsers # Adds new user
3 DELETE /users/delete # Deletes all users
```

Pierwszy z powyższych przykładów posiada jeden zasób users, na którym wykonujemy różne akcje z wykorzystaniem metod HTTP, czyli mamy jeden zasób na którym wykonujemy wiele akcji. Natomiast przykład drugi posiada trzy różne adresy, które są połączone z akcją wykonywaną w naszej aplikacji. Jednoznacznie widać, że założenia z przykładu drugiego mocno odbiegają od naszych założeń REST, ponieważ powiązanie API z naszą aplikacją uniemożliwia rozwijanie się aplikacji, jak i API niezależnie od siebie.

— **Rzeczowniki vs. Czasowniki** - jednym ze sposobów weryfikacji, czy zasób jest odłączony od akcji, jest użycie samego rzeczownika w nazwie. Jeśli w adresie URI zasobu znajduje się czasownik, np. /getUsers lub /addUsers, to

możemy podejrzewać że ten zasób jest bardzo mocno powiązany z konkretną akcją.

— **Liczba pojedyncza vs. Liczba mnoga** - starajmy się używać liczby mnogiej przy nazewnictwie zasobów, np. /users zamiast /user. Zwiększa to czytelność naszego API, jednakże ta zasada jest bardzo dyskusyjna. Obiektywnie nasze API powinno być zbudowane konsekwentnie i spójnie.

2.5. Model dojrzałości Richardsona

Jako, że REST jest tylko zbiorem wytycznych i dobrych rad dla tworzenia API, to deweloperzy mają bardzo dużą elastyczność do definiowania jak bardzo "REST-owe" będzie ich API. Zakładając sytuację, że napiszemy prostą aplikację RESTful API, to skąd możemy mieć pewność, że jest ona w stylu REST? Można powiedzieć, że skoro wykorzystujmy protokół HTTP i używamy formatu danych JSON, to serwis jest RESTful. **Model dojrzałości Richardsona**(ang. *Richardson Maturity Model*) pozwala ocenić w jakim stopniu aplikacja jest RESTful. Model definiuje 4 poziomy (0-3) i zakłada, że **każde API należy do jednego z czterech poziomów dojrzałości**.

— **Poziom 0** - Na poziomie zerowym modelu dojrzałości znajdują się aplikacje, które nie spełniają założeń architektury REST. Protokół HTTP wykorzystywany jest tylko jako warstwa transportowa. Używane są tylko metody GET i POST oraz status zwracany zawsze jest 200 OK. Nie ma unikalnych adresów dla zasobów. Do tego poziomu zaliczamy np. protokoły SOAP i RPC.

Wyciąg 2-7. Przykładowe API dla poziomu 0

```
POST /api/findSomething
POST /api/updateSomething
```

Wyciąg 2-8. Dodawanie np. rzeczy

```
POST /api/addSomething HTTP/1.1
{ "item_name":"Chair" }
```

Wyciąg 2-9. Odpowiedzi

```
200 OK
{ "item_id":50 }
# or
200 OK
{ "error":"Wrong data input." }
```

— **Poziom 1** - Na pierwszym poziomie modelu dojrzałości wykorzystywanych jest wiele identyfikatorów URI, jednakże używa się podstawowych metod HTTP, jak np. GET, czy POST. Na tym poziomie mamy już zdefiniowane zasoby, które powinny posiadać indywidualny identyfikator URI. Zasób nie powinien być dostępny pod wieloma adresami, tzn. musi mieć unikalny adres.

Wyciąg 2-10. Przykładowe API dla poziomu 1

```
POST /api/something/id/get
POST /api/something/create
```

— **Poziom 2** - Drugi poziom modelu dojrzałości rozszerza podstawowe metody HTTP o dodatkowe, jak np. PUT, PATCH i DELETE. Dodane metody używane są zgodnie z ich przeznaczeniem.

Wyciąg 2-11. Przykładowe API dla poziomu 2

```
POST /api/something
GET /api/something/item_id
PUT /api/something/item_id
PATCH /api/something/item_id
DELETE /api/something/item_id
```

Drugi poziom wprowadza też rozszerzenie dla statusów odpowiedzi. Statusy należy wykorzystywać zależnie od sposobu realizacji zapytania.

— 1XX – Kody informacyjne
— 2XX – Kody powodzenia
— 3XX – Kody przekierowania
— 4XX – Kody błędu aplikacji klienta
— 5XX – Kody błędu serwera

Wyciąg 2-12. Dodawanie np. rzeczy

```
POST /api/addSomething HTTP/1.1
{ "item_name":"Chair" }
```

Wyciąg 2-13. Odpowiedzi

```
201 Created
{ "item_id":50 }
# or
409 Conflict
{ "error":"Wrong data input." }
```

— **Poziom 3** - Trzeci poziom (najwyższy) wymaga implementacji HATEOAS. HATEOAS zarządza interakcją dla klienta. Klient nie zna wszystkich punktów końcowych API dla danych żądań.

Wyciąg 2-14. Przykład pobierania osoby o identyfikatorze 36:

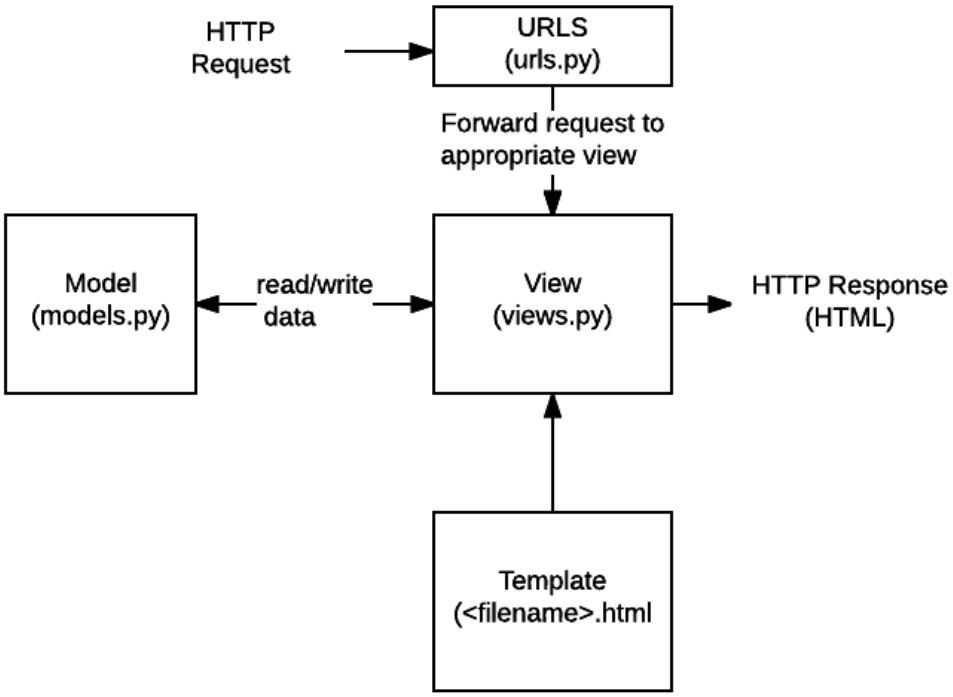
```
GET /api/persons/36 HTTP/1.1
```

Wyciąg 2-15. Przykładowa odpowiedź:

```
HTTP/1.1 200 OK
{ "id":36, "name":"Damian Kuich", "age":23,
"link":{"address":"/api/addresses/43"} }
```

2.6. Django

Django jest frameworkiem typu open-source, bazującym na języku Python, który w bardzo prosty sposób umożliwia tworzenie skalowalnych i szybkich aplikacji webowych. Operuje on na modelu architektonicznym **MTV**(ang. *Model-Template-Views*).



Rysunek 2.5. Architektura MTV
Źródło:
https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction

2.6.1. URLs

Chociaż możliwe jest przetwarzanie żądań z każdego adresu URL za pośrednictwem jednej funkcji, znacznie łatwiej jest napisać oddzielny widok (ang. *View*) do obsługi wielu zasobów. URLs służy do przekierowywania żądań HTTP do odpowiedniego widoku na podstawie adresu URL. Narzędzie odwzorowujące adresy URL może również dopasować określone wzorce ciągów lub cyfr, które pojawiają się w adresie URL, i przekazać je dalej jako dane.

2.6.2. Views

View służy do przetwarzania żądań i odpowiedzi HTTP. Pobiera dane potrzebne do wykonania żądań z pomocą modeli (ang. Models), jak i formatują odpowiedzi do szablonów (ang. Templates).

2.6.3. Models

Modele to obiekty, które kształtują strukturę danych aplikacji. Umożliwiają wykonywanie na danych operacji, takich jak usuwanie, dodawanie, czy modyfikowanie.

2.6.4. Templates

Szablony to pliki tekstowe definiujące strukturę lub układ pliku (na przykład strony HTML), z symbolami zastępczymi używanymi do reprezentowania rzeczywistej zawartości. Widok może dynamicznie tworzyć stronę HTML przy użyciu szablonu HTML, wypełniając ją danymi z modelu. Szablon może służyć do definiowania struktury dowolnego typu pliku nie tylko HTML.

2.7. Django REST Framework

Django REST Framework jest frameworkiem typu open-source napisanym w języku Python. Pozwala on na implementacje architektury REST do naszego API, co umożliwia nam utworzenie RESTful API. Opiszmy więc, jak została zaimplementowana architektura REST do tego frameworka, przechodząc przez jej główne założenia z rozdziału 2.3.

2.7.1. Unifrom Interface

Unifrom Interface zakłada ujednolicenie API i danych, by były one klarowne dla innych serwisów. Powinno to oznaczać, że system powinien posiadać jeden główny URI dzięki, któremu byśmy mogli korzystać z metod HTTP. Django REST Framework pomaga w budowaniu interfejsu API REST poprzez tworzenie punktów końcowych za pomocą widoków ogólnych. Te ogólne widoki pomagają stworzyć pełny punkt końcowy CRUD (create, read, . . .) w modelu Django przy niewielkim wysiłku. Utworzony punkt końcowy jest zgodny z interfejsem API Rest struktury adresu URL dla zasobu.

2.7.2. HATEOAS

To czego REST nie potrafi, to dostarczenie czytelnych dla maszyny formantów hypermedialnych takich jak HAL, Collection+JSON, JSON API lub mikroformantów HTML w sposób domyślny. Nie udostępnia również możliwości automatycznego stworzenia pełnego API HATEOAS zawierającego opisy formatów i semantycznie oflagowanych hiperlinków. Dla osiągnięcia takiego efektu musiałyby zostać podjęte decyzje wykraczające poza obszar frameworka.

2.7.3. Client-Server

Już sama esencja tego, że Django REST Framework pozwala utworzyć backend serwera, który nie jest częścią interfejsu użytkownika aplikacji, jest istotą architektury klient-serwer. Każda aplikacja, którą tworzymy za pomocą Django Rest Framework posiada API, które jest zupełnie oddzielne od API klienta.

2.7.4. Stateless

Django REST Framework dzięki interfejsowi WSGI/ASGI nie musi zapisywać żadnych informacji ani polegać na nich. Poniżej zobrazowano skróconą zasadę działania:

Wyciąg 2-16. Przykład kodu stateless, gdzie nie zapisujemy żadnych danych:

```
while True:
    request = io.listen_for_request()
    response = handle_request(request)
return response
```

2.7.5. Cacheable

Django REST Framework ma wbudowane narzędzia do Cachowania.

2.7.6. Layered System

Django zapewnia wiele interesujących połączeń między warstwami, a użycie go dla wszystkich warstw pozwala wykorzystać te połączenia. PRzykładowo, użycie Django ORM oznacza, że otrzymujemy świetną aplikację administracyjną. Możemy wybrać inny ORM, ale nie dostaniemy wraz z nim aplikacji administratora. Django, też nie utrudnia rozbicia naszego serwisu na wiele pomniejszych mikroserwisów.

2.7.7. Code-On-Demand

Django umożliwia na rozszerzenie funkcjonalności szablonów do wysyłania np. formularzy jako endpoint.

2.8. Podsumowanie

W pracy przedstawiono podstawowe zagadnienia związane z API. Omówiono także główne cechy i zasady REST. Zwrócono uwagę na to, co sprawia, że API jest RESTful. Wspomniano też o ewolucji API dzięki modelowi dojrzałości Richardsona. Na końcu ponownie przedstawiono główne założenia architektury REST, jednakże tym razem dla stwierdzenia czy Django REST Framework naprawdę pozwala na tworzenie prawdziwego RESTful API. Po przeanalizowaniu implementacji głównych paradygmatów REST w Django REST Framework, czy nawet w samym Django, można wywnioskować, że umożliwia on tworzenie API zgodnie z REST.

Rozdział 3

Integracja projektu opartego o Django z bazą danych PostgreSQL

Baza danych to zbiór informacji, danych, zazwyczaj przechowywanych w formie elektronicznej w systemie komputerowym. Dane umieszczone są w wierszach i kolumnach wielu tabel, dzięki czemu przetwarzanie ich oraz tworzenie zapytań ich dotyczących odbywa się w zorganizowany i sprawny sposób. W dzisiejszych czasach większość baz danych wykorzystuje język SQL (Structured Query Language), czyli strukturalny język zapytań. SQL to język programowania używany do tworzenia zapytań dotyczących danych. Jest wykorzystywany przez niemal wszystkie relacyjne bazy danych. Został on opracowany w latach 70-tych przez firmę IBM. Przy tworzeniu języka brała udział firma Oracle, czego efektem było wprowadzenie standardu SQL ANSI i powstanie wielu rozszerzeń języka SQL. Istnieje wiele typów baz danych. Oto kilka z nich:

— Relacyjne bazy danych
— Obiektowe bazy danych
— Rozproszone bazy danych
— Hurtownie danych
— Bazy danych NoSQL
— Grafowe bazy danych

Bazy danych obsługiwane są przez system zarządzania DBMS (Database Management System). Stanowi on swego rodzaju pomost między danymi a programami wykorzystującymi te dane lub użytkownikami końcowymi. Dzięki niemu można pobierać, aktualizować informacje oraz konfigurować sposób, w jaki są one zorganizowane. System pozwala również na wykonywanie zadań administracyjnych, takich jak tworzenie oraz przywracanie kopii zapasowych. W pracy krótko przedstawiono cztery systemy zarządzania bazami danych:

— MySQL
— MS SQL Server
— Oracle
— PostgreSQL

3.1. MySQL

MySQL to darmowy i otwarty system zarządzania relacyjnymi bazami danych (RDBMS). Jego właścicielem jest Oracle, który oferuje płatne wersje premium MySQL z dodatkowymi usługami. Jest to lekka baza danych, która może być instalowana i używana przez programistów na serwerach aplikacji produkcyjnych z dużymi aplikacjami wielowarstwowymi, a także na komputerze stacjonarnym. Jest bezpieczny i nie jest podatny na żadne luki w zabezpieczeniach. MySQL oferuje obsługę następujących języków: C/C++, Delphi, Java, Lisp, Node.js, Perl, PHP, R. MySQL oferuje obsługę w chmurze, instalacje lokalne oraz jest kompatybilny z następującymi systemami operacyjnymi i formatami: Windows, MacOS, Linux (Ubuntu, Debian, Generic, SUSE Linux Enterprise Server, Red Hat Enterprises, Oracle), Oracle Solaris, Fedora, FreeBSD.

Tabela 3.1. Wady i zalety MySQL

Zalety	Wady
Ochrona danych	Złożona logika biznesowa
Skalowalność na żądanie	Kilka problemów ze stabilnością
Całodobowa dostępność	Transakcje nie są obsługiwane zbyt wydajnie
Kompleksowe wsparcie transakcyjne	Funkcjonalność zależna od dodatków
Pełna kontrola przepływu pracy	Wcześniejsza wiedza jest koniecznością
Elastyczność open source	
Szerokie zastosowanie i łatwe w użyciu	
Szybki, przenośny i bezpieczny	Konieczna jest znajomość tematu

3.2. MS SQL Server

MS SQL Server to relacyjny system zarządzania bazami danych (RDBM) opracowany i obsługiwany przez firmę Microsoft. Używa wariantu *Structured Query Language* (SQL) o nazwie T-SQL. Zwykle zawiera silnik relacyjnej bazy danych, który przechowuje dane w tabelach, kolumnach i wierszach, *Integration Services* (SSIS), które są narzędziem do przenoszenia danych do importowania, eksportowania i przekształcania danych. W jego skład wchodzą również *Reporting Services* (SSRS), które służą do tworzenia raportów i udostępniają je użytkownikom końcowym, a także *Analysis Services* (SSAS), czyli wielowymiarowa baza danych służąca do wykonywania zapytań o dane z głównego silnika. MS SQL Server oferuje obsługę języków programowania takich jak: C#, C++,

Delphi, Go, Java, JavaScript, PHP, Python, R, Ruby , Visual Basic. MS SQL Server jest dostępny na systemach `Microsoft Windows` oraz Linux.

Tabela 3.2. Wady i zalety MS SQL Server

Zalety	Wady
Dostępna wersja darmowa systemu	Kosztowna wersja dla przedsiębiorstw
Wysokie bezpieczeństwo danych	Ograniczona kompatybilność
Łatwość konfiguracji	Kilka problemów ze stabilnością
Łatwość konfiguracji	
Wsparcie odzyskiwania danych	

3.3. Oracle

Oracle to system zarządzania relacyjnymi bazami danych wydany w 1980 roku z podstawowymi funkcjami SQL, który sam się steruje, zabezpiecza, samo naprawia i eliminuje podatne na błędy ręczne zarządzanie bazą danych. Umożliwia bezpieczne przechowywanie i szybkie pobieranie danych. Oracle to pierwsze narzędzie bazodanowe opracowane w celach biznesowych, do zarządzania danymi za pomocą języka zapytań. Oracle oferuje obsługę języków programowania takich jak: C, C#, C++, Delphi, Haskell, Java, JavaScript, C, Perl, PHP, Python, R, Ruby, Visual Basic. Oracle jest dostępny dla następujących systemów operacyjnych: Windows, Mac OS X, Linux, UNIX, z/OS.

Tabela 3.3. Wady i zalety MySQL

Zalety	Wady
Wszechstronność	Trudny do nauczenia i obsługi
Kompatybilność ze składnią SQL i PL	Wysoki koszt obsługi
Poprawiona wydajność	Składniki zapytań poza standardem ANSI
Grupowanie transakcji	Intensywne obciążenie sprzętu

3.4. PostgreSQL

PostgreSQL to darmowy i otwarty system zarządzania obiektowo-relacyjnymi

bazami danych (ORDBMS). Obecnie zarządzany jest przez społeczność programistów, która pomaga ułatwić i przyspieszyć pracę wszystkich użytkowników. PostgreSQL zapewnia zarówno obiektową, jak i relacyjną funkcjonalność bazy danych. Był pierwszym systemem DBMS, który zaimplementował funkcje kontroli współbieżności wielu wersji (MVCC). Jest obecnie uważany za najbardziej zaawansowany silnik bazy danych. Obsługuje tekst, obrazy, dźwięki i wideo oraz zawiera interfejsy programistyczne dla wielu języków. PostgreSQL może być rozszerzany przez użytkownika na wiele sposobów. PostgreSQL oferuje obsługę nieco szerszej gamy języków: C/C++, Delphi, Erlang, Go, Java, JavaScript, Lisp, .Net, Python, R, Tcl. PostgreSQL jest dostępny dla następujących systemów operacyjnych: MacOS, Solaris, Windows, BSD (FreeBSD, OpenBSD), Linux, CentOS, Fedora, Scientific, Oracle, Debian, Ubuntu Linux i pochodne.

Tabela 3.4. Wady i zalety PostgreSQL

Zalety	Wady
Wysoce skalowalny	Stosunkowo niska prędkość czytania
Posiada wiele interfejsów	Optymalna konfiguracja może być kłopotliwa
Obsługuje JSON	
Szerokie możliwości rozbudowy	
Przetwarzanie złożonych typów danych	
Elastyczne wyszukiwanie pełno tekstowe	
Wieloplatformowy	
Zdolny do obsługi danych w TB	

W projekcie składającym się na praktyczną część pracy, zastosowany został PostgreSQL. Jednym w głównych czynników wpływających na wybór był fakt, że PostgreSQL oferowany jest poprzez używaną podczs wdrożenia platformę chmurową Heroku.

3.5. Łączenie z bazą danych PostgreSQL

Integracja Django z PostgreSQL rozpoczyna się od stworzenia nowej bazy lub połączenia z obecnie istniejącą. Domyślnie Django jest skonfigurowane do używania `SQLite` jako swojego zaplecza, dlatego wymagana jest lekka zmiana zawartości pliku settings.py tak, aby używanie Postgresa było możliwe.

Poniżej znajduję się kod **??** potrzebny do połączenia się bazą danych oferowaną przez Heroku.

Wyciąg 3-1. Fragment pliku `settings.py`

```
DATABASES = {,
        'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': '',
        'USER': '',
        'PASSWORD': '',
        'HOST': 'ec2-54-160-18-230.compute-1.amazonaws.com',
        'PORT': '5432',
        }
}
```

3.6. Modele

Na początku baza posiada podstawowe modele danych takie jak `Users` oraz `Groups`. Aby zawierała inne dane, potrzebne jest zdefiniowanie ich we frameworku, w pliku models.py. Najważniejszą częścią modelu są zdefiniowane pola bazy danych, będące instancja klasy `Field`. Są to na przykład: `CharField ,TextField, ForeignKey, IntegerField, BooleanField, DateTimeField.`

Pola modeli przeznaczone dla PostgreSQL (django.contrib.postgres.fields):

— `ArrayField` - pole do przechowywania list danych.
— `HStoreField` - pole do przechowywania par klucz:wartość. W Pythonie typ danych reprezentowany jako dict.
— `JSONField` - pole do przechowywania danych zakodowanych w formacie `JSON`.
— `IntegerRangeField` - pole przechowuje zakres liczb całkowitych.
— `BigIntegerRangeField` - pole przechowuje zakres dużych liczb całkowitych.
— `DecimalRangeField` - pole przechowuje zakres wartości zmiennoprzecinkowych.
— `DateTimeRangeField` - pole przechowuje szereg sygnatur czasowych.
— `DateRangeField` - pole przechowuje zakres dat.

Każde pole określone jest poprzez argumenty. Przykładowo pole `CharField` wymaga podania argumentu `max_length`, który określa maksymalną ilość znaków. Każde pole posiada również zestaw ogólnodostępnych argumentów np.: `null (True/False)`, `blank (True/False)`, `default`, `unique (True/False)`.

Wyciąg 3-2. Fragment pliku models

```
class Section(models.Model):
    section_name = models.CharField(max_length=500,unique=True)

    def __str__(self):
        return self.self_name()

    def self_name(self):
        return self.name

class Skill(models.Model):
    skill_name = models.CharField(max_length=500,unique=True)
    section = models.ForeignKey(Section, on_delete=models.CASCADE)
          # ...
        return self.skill_name

class Task(models.Model):
    TYPES = {
        (1, "Otwarte"),
        (2, "Zamknięte"),
    }
    LEVELS = {
        (1, "Podstawowy"),
        (2, "Rozszerzony"),
    }
    text = models.CharField(max_length=600,unique=True)
    wrong_answers =ArrayField(models.CharField(
                                max_length=200), blank=True)
    correct_answers =ArrayField(models.CharField(
                                max_length=200), blank=True)
    task_type = models.IntegerField(choices=TYPES)
    level = models.IntegerField(choices=LEVELS)

          # ...
        return self.text


class TestJSON(models.Model):
    name = models.TextField(null=True)
    tasks = JSONField(null=True)

    # ...
        return self.name
```

Powyższy przykład **??** obrazuje definicje modeli `Skill`, `Section`, `Task` oraz `TestJSON`, które odpowiadają zadaniu, działom wraz z umiejętnościami oraz obiektu testu w Generatorze Sprawdzianów. Model `Task` zawiera m.in. pola wrong_answers oraz correct_answers będące polami typu `ArrayField`. Pola te przeznaczone są do przechowywania poprawnych oraz niepoprawnych odpowiedzi na pytania zamknięte. Odpowiedzi może być wiele, dlatego z pomocą przychodzi `ArrayField`, czyli lista obiektów. W tym przypadku są to obiekty typu tekstowego. Model `TestJSON` zawiera m.in. pole tasks, zawierające wszystkie zadania w formacie JSON przypisane do danego testu. Dzięki temu każdy użytkownik może dodawać do testu zadanie z naszej bazy, a następnie edytować je według własnego uznania bez ingerencji w oryginał. Na końcu każdego modelu dodana została jego reprezentacja, przez co w zakładce admin widoczna będzie nazwa obiektu, co w znacznym stopniu ułatwi jego identyfikację.

3.7. Migracje

Po zdefiniowaniu modelu, wymagane jest wykonanie migracji. Migracje to sposób Django na propagowanie zmian schematu bazy danych takich jak dodawanie pola, edytowanie, usuwanie modelu itp. Są zaprojektowane w taki sposób, aby były w większości automatyczne, jednak wymagane jest ich tworzenie w taki sposób, aby uniknąć powstania błędów. *Makemigrations* jest odpowiedzialne za pakowanie zmian w modelu do indywidualnych plików migracyjnych, z kolei *migrate* jest odpowiedzialny za stosowanie ich w bazie danych. Niektóre bazy danych są bardziej wydajne niż inne, biorąc pod uwagę migrację schematów. PostgreSQL cechuje się wysoką wydajnością pod względem obsługi schematów, natomiast MySQL nie obsługuje transakcji zmiany schematu. Jeżeli dana migracja się nie powiedzie, zmuszeni jesteśmy ręcznie cofnąć zmiany, a następnie spróbować ponownie.

Aby zbudować tabele w bazie danych, wykonujemy migrację i uruchamiany ją poniższymi poleceniami:

```
1 >python manage.py makemigrations
2 >python manage.py migrate
```

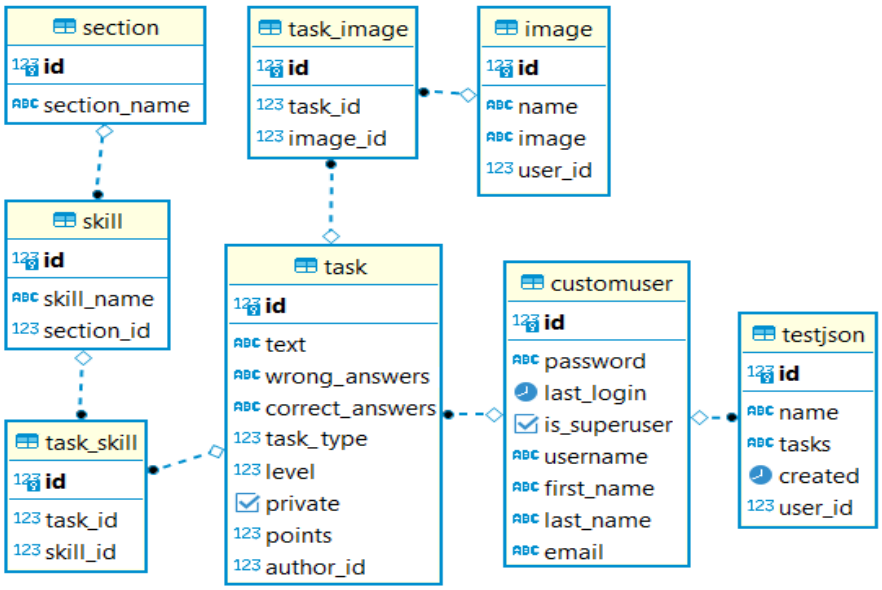
Po wykonaniu migracji, Django wprowadzi do bazy danych następujące polecenie.

Wyciąg 3-3. SQL dla modelu Task, Skill oraz Section

```
CREATE TABLE "section" (
        "id" serial NOT NULL PRIMARY KEY,
        "section_name" varchar(500) NOT NULL UNIQUE);

CREATE TABLE "skill" (
        "id" serial NOT NULL PRIMARY KEY,
        "skill_name" varchar(500) NOT NULL UNIQUE,
        "section_id" integer NOT NULL);

CREATE TABLE "task" (
        "id" serial NOT NULL PRIMARY KEY,
        "text" varchar(600) NOT NULL UNIQUE,
        "wrong_answers" varchar(200)[] NOT NULL,
        "correct_answers" varchar(200)[] NOT NULL,
        "task_type" integer NOT NULL,
        "level" integer NOT NULL,
        "private" boolean NOT NULL,
        "points" integer NOT NULL,
        "author_id" integer NOT NULL);

CREATE TABLE "task_skill" (
        "id" serial NOT NULL PRIMARY KEY,
        "task_id" integer NOT NULL,
        "skill_id" integer NOT NULL);
        # ...
```

Porównując wycinek **??** z poleceniem **??** zauważyć można, że w pliku models.py nie został zdefiniowany atrybut `id`, będący kluczem głównym tabeli. W takim przypadku framework Django własnoręcznie wykonał jego określenie.

3.8. Relacje



Rysunek 3.1. Główne tabele

Powyższy rysunek **??** obrazuje głowne tabele znajdujące się w bazie danych naszego projektu. Każda tabela połączona jest z jedną lub wieloma tabelami za pomocą relacji.

Relacjami nazywamy skojarzenia między dwoma typami jednostek. W relacyjnej bazie danych działają one między dwiema tabelami, przy czym jedna tabela posiada klucz obcy klucza podstawowego drugiej tabeli. Dzięki relacjom możemy dzielić i przechowywać dane w różnych tabelach.

W Django możliwe jest użycie trzech najpopularniejszych typów relacji w bazie danych:

— Jeden do jednego ang. *One-to-one*
— Jeden do wielu ang. *One-to-many*
— Wiele do wielu ang. *Many-to-many*

3.8.1. Relacje jeden do jednego

W relacji jeden do jednego tylko jeden rekord tabeli A (najczęściej klucz podstawowy) może być powiązany z innym rekordem tabeli B mającym taką

samą wartość. Przykładem może być uczeń, który posiada tylko jedną szafkę w szkole. W Django należy użyć typu `OneToOneField`, aby zdefiniować relację jeden do jednego.

Wyciąg 3-4. Jeden do jednego Student-School locker

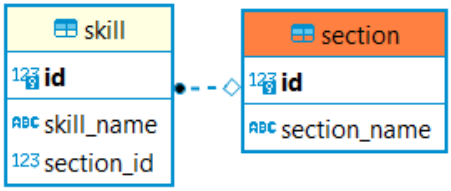
```
class School_locker(models.Model):
    address = models.CharField(max_length=50)

    def __str__(self):
        return self.address


class Student(models.Model):
    name = models.CharField(max_length=25)
    locker = models.OneToOneField(
        School_locker,
        on_delete=models.CASCADE,
        primary_key=True,
    )

    def __str__(self):
        return self.name
```

3.8.2. Relacje jeden do wielu

Relacja jeden do wielu jest najczęściej używanym typem połączenia i odpowiada sytuacji, gdy pojedynczemu rekordowi z tabeli A jest przyporządkowany jeden lub wiele rekordów z tabeli B, natomiast pojedynczemu rekordowi z tabeli B jest przyporządkowany dokładnie jeden rekord z tabeli A. Aby zdefiniować relację jeden do wielu w Django, należy użyć **ForeignKey**. Przykładem mogą być modele `Section` (dział) oraz `Skill` (umiejętność) zawarte w projekcie.

Wyciąg 3-5. Jeden do wielu Section-Skill

```
class Section(models.Model):
    section_name = models.CharField(max_length=500, unique=True)
        # ...


class Skill(models.Model):
    skill_name = models.CharField(max_length=500, unique=True)
    section = models.ForeignKey(Section, on_delete=models.CASCADE)
        # ...
```



Rysunek 3.2. Wiele do jednego Section-Skill

W modelu Skill zawarty jest klucz obcy modelu Section, co pozwala na przypisanie kilku umiejętności do jednego działu.

3.8.3. Relacje wiele do wielu

Relacja wiele do wielu występuje w sytuacji, gdy rekordowi tabeli A przyporządkowany jest jeden lub wiele rekordów tabeli B. Sytuacja ta ma miejsce również z drugiej strony. Aby zdefiniować relację wiele do wielu w Django, należy użyć `ManyToManyField`. Za przykład można obrać model `Task` (zadanie) oraz `Skill` (umiejętność).

Wyciąg 3-6. Wiele do wielu Task-Skill

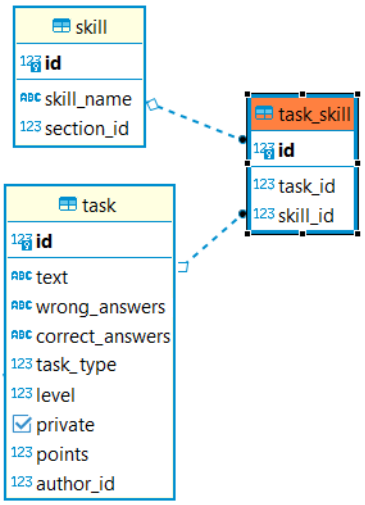
```
class Skill(models.Model):
    skill_name = models.CharField(max_length=500, unique=True)
    section = models.ForeignKey(Section, on_delete=models.CASCADE)
        # ...


class Task(models.Model):
        # ...
```

```
9       skill = models.ManyToManyField(Skill)
10          # ...
```

Sposób w jaki obiekty są ze sobą wiązane za pomocą relacji wiele do wielu znacząco różni się od radzenia sobie z kluczem obcym, ponieważ wymagane jest zapisanie umiejętności w bazie danych, zanim przypisana zostanie do zadania i odwrotnie. Dzieje się tak, ponieważ `ManyToManyField` tworzy **pośrednią tabelę złączeń**, która wiąże model źródłowy z modelem docelowym. Aby stworzyć połączenie między modelami, oba muszą zostać dodane do tego niewidocznego modelu. Dokumentacja Django informuje, że:

> ...nadal istnieje niejawna klasa modelu, której możesz użyć do bezpośredniego dostępu do tabeli utworzonej do przechowywania skojarzenia. Posiada trzy pola do łączenia modeli.
> Jeśli model źródłowy i docelowy różnią się, generowane są następujące pola:
> **id**: klucz główny relacji
> **<zawierający_model>_id**: model **id**, który deklaruje **ManyToManyField**
> **<inny_model>_id**: model **id**, który wskazywany jest przez **ManyToManyField**
>
> (")

Niewidoczny model pośredni stworzony przez Django wymaga pary kluczy. Jest to klucz podstawowy modelu źródłowego oraz klucz modelu docelowego, dlatego oba obiekty muszą istnieć, zanim będą mogły zostać powiązane.



Rysunek 3.3. Wiele do wielu Task-Skill

W przypadku projektu stworzona została tabela o nazwie `task_skill` zawierająca trzy kolumny: `id`, `task_id` oraz `skill_id`.

3.8.4. Integralność danych

Wszystkie relacje tworzą zależności między połączonymi modelami, więc ważne jest, aby określić co dzieje się z drugą stroną, gdy pierwsza zostanie usunięta. W tym celu została stworzona opcja `on_delete`, której zadaniem jest określać, co należy zrobić z rekordami, których druga strona została usunięta. Opcja ta jest dostępna dla wszystkich trzech relacji i może przyjmować następujące wartości:

— `on_delete=models.CASCADE` - Automatycznie usuwa powiązane rekordy, które były powiązane z daną instancją.
— `on_delete=models.PROTECT` - Zapobiega usunięciu powiązanej instancji.
— `on_delete=models.SET_NULL` - Przypisuje `NULL` do powiązanych rekordów, gdy powiązana instancja jest usuwana. Należy pamiętać aby użyć opcji null=True w polu.
— `on_delete=models.SET_DEFAULT` - Przypisuje wartość domyślną do powiązanych rekordów, gdy powiązana instancja jest usuwana.
— `on_delete=models.SET` - Przypisuje wartość ustawioną przez wywołanie do powiązanych rekordów, gdy powiązana instancja zostanie usunięta.
— `on_delete=models.DO_NOTHING` - Żadne działanie nie jest podejmowane po usunięciu powiązanych rekordów. Zwykle jest to zła praktyka dotycząca relacyjnych baz danych, ponieważ powstają osierocone rekordy bez wartości.

3.9. ORM w Django

Po utworzeniu modeli danych, Django automatycznie udostępnia interfejs `QuerySet`, który umożliwia tworzenie, pobieranie, aktualizowanie i usuwanie obiektów. Są to polecenia, dzięki którym możliwe jest pobieranie informacji z bazy danych oraz otrzymywanie ich w postaci obiektów. Polecenia te możemy wykonywać z pomocą konsoli po wpisaniu następującej komendy:

```
1 >python manage.py shell
```

Kolejnym wymogiem jest zaimportowanie modeli, na których chcemy pracować.

```
>>>from projekt.models import Task, CustomUser, Skill
```

W tym momencie mamy możliwość tworzenia, otrzymywania oraz filtrowania obiektów.

3.9.1. Tworzenie obiektu

Wyciąg 3-7. .

```
>>>user = CustomUser.objects.get(id = 1)
>>>skill = Skill.objects.get(id = 1)
>>>task = Task.objects.create(text = 'zadanie przyklad',
        wrong_answers = [20,12,32], correct_answers = [21],type = 2,
        author = user,level = 1, private = False, points = 2)
>>>task.skill.add(skill)
```

W powyższym przykładzie **??** należy początkowo pobrać z bazy obiekt użytkownika oraz wybraną umiejętność. Następnym krokiem jest utworzenie obiektu `Task` przy pomocy metody `create` i po wprowadzeniu odpowiednich danych. Na końcu należy przypisać obiekt `skill` do pola `task.skill`. Poniżej znajduję się skrypt SQL **??** wykonany przez Django.

Wyciąg 3-8. .

```
SELECT "customuser"."id",
       "customuser"."password",
          ...
  FROM "customuser"
 WHERE "customuser"."id" = 1

SELECT "skill"."id",
       "skill"."skill_name",
       "skill"."section_id"
  FROM "skill"
 WHERE "skill"."id" = 1

INSERT INTO "task" ("text", "wrong_answers",
                "correct_answers", "type", "author_id",
                "level", "private", "points")
VALUES ('zadanie przyklad', ARRAY['20','12','32']::varchar(200)[],
        ARRAY['21']::varchar(200)[], 2, 1, 1, false, 2)
        RETURNING "task"."id"
```

3.9.2. Pobieranie obiektu

Pobieranie danych z bazy może być wykonane na wiele sposobów. Oto kilka metod:

— `all()` - zwraca wszystkie rekordy danego modelu
— `get()` - zwraca tylko jeden rekord spełniający warunek w `get()`
— `filter()` - zwraca rekordy spełniające warunek w `filter()`
— `reverse()` - obiekty zwracane będą w odwrotnej kolejności
— `distinct()` - zwraca tylko unikalne obiekty bez duplikatów

Pobieranie obiektów z bazy danych czasem może trwać długo, dlatego istotne jest to, aby wykonywane skrypty działały w sposób optymalny.

Ważnym jest, że zapytania wykonują się dopiero w momencie odniesienia do danych. Jeśli ponownie wykonamy to samo zapytanie, czas jego wykonania będzie krótszy niż poprzednio, ponieważ zostanie ono zapisane w cache.

Wyciąg 3-9. .

```
>>>tasks = Task.objects.all()
>>>list(tasks)
>>>tasks[0]
```

W powyższym przykładzie **??** zapytanie zostało wykonane dopiero w drugiej linii. Element listy `tasks[0]` został zapamiętany, więc zapytanie nie zostało ponownie wywołane.

Zdarza się, że podczas pozyskiwania danych z bazy zostają pobrane całe obiekty, mimo że wykorzystywane jest jedynie kilka parametrów. Takie działanie może znacząco wpłynąć na czas wykonania, dlatego warto skorzystać z `values` (wynik w formie słownika), lub `values_list` (wynik w formie listy)

```
>>> Task.objects.all()
SELECT "task"."id",
       "task"."text",
       "task"."wrong_answers",
       "task"."correct_answers",
       "task"."type",
       "task"."author_id",
       "task"."level",
       "task"."private",
       "task"."points"
  FROM "task"

```

```
>>> Task.objects.values("text")
SELECT "task"."text"
  FROM "task"
```

Przechodząc po liście obiektów tylko raz, można wykorzystać do tego iterator, który podczas iteracji odwoływać się będzie do kolejnych elementów. W ten sposób unikniemy niepotrzebnego ładowania całego zapytania. Poniższy przykład **??** ukazuje o wiele szybsze działanie polecenia przy użyciu iteratora.

Wyciąg 3-10. .

```
>>> for task in Task.objects.all():
...     text = task.text
SELECT "task"."id",
       ...
  FROM "task"
Execution time: 0.011384s [Database: default]
>>> for task in Task.objects.all().iterator():
...     text = task.text
DECLARE "_django_curs_140417171636352_sync_8" NO SCROLL
CURSOR WITH HOLD
   FOR SELECT "task"."id",
              ...
  FROM "task"
Execution time: 0.002761s [Database: default]
```

Funkcja `select_related` podczas wykonywania zapytania podąża za relacjami klucza obcego, pobierając odpowiednie dane obiektu powiązanego relacją. Funkcja ta, w przeciwieństwie do wykonywania wielu zapytań, pozwala na stworzenie jednego, złożonego, w którym użycie klucza obcego nie będzie skutkowało wykonaniem kolejnego zapytania. Przykładem będzie sytuacja, w której dla dziesięciu zadań posiadających umiejętność o `id` równym 6, chcemy wypisać ich autorów.

Wyciąg 3-11. .

```
>>> id_of_authors = []
>>> e = Task.objects.filter(skill=6)[:10]
>>> exec("for task in e: text = print(task.author.id)")
SELECT "task"."id",
       ...
  FROM "task"
 INNER JOIN "task_skill"
    ON ("task"."id" = "task_skill"."task_id")
```

```
 WHERE "task_skill"."skill_id" = 6
 LIMIT 10
Execution time: 0.001817s [Database: default]
SELECT "customuser"."id",
       ...
 WHERE "customuser"."id" = 2
 LIMIT 21
Execution time: 0.001256s [Database: default]
Execution time: 0.001192s [Database: default]
Execution time: 0.001339s [Database: default]
Execution time: 0.001212s [Database: default]
Execution time: 0.001298s [Database: default]
Execution time: 0.001349s [Database: default]
Execution time: 0.001313s [Database: default]
Execution time: 0.001406s [Database: default]
Execution time: 0.001275s [Database: default]
Execution time: 0.001305s [Database: default]
>>> print(id_of_authors)
[2, 4, 4, 3, 4, 2, 1, 3, 4, 3]
```

Powyższy kod **??** bez użycia `select_related()` wykona aż 11 zapytań do bazy. Jedno zwracające zadania, a każde kolejne o autora w pętli. Łączny czas wykonywanie wynosi 0.012945s. Poniżej znajduje się kod **??** wykorzystujący `select_related()`, który dokona tylko jednego zapytania, a czas wynosić będzie 0.002796s.

Wyciąg 3-12. .

```
>>> id_of_authors = []
>>> e = Task.objects.select_related('author').filter(skill=6)[:10]
>>> exec("for task in e: text = print(task.author.id)")
SELECT "task"."id",
       ...
      "customuser"."id",
       ...
  FROM "task"
 INNER JOIN "task_skill"
    ON ("task"."id" = "task_skill"."task_id")
 INNER JOIN "customuser"
    ON ("task"."author_id" =
Execution time: 0.002796s [Database: default]
>>> print(id_of_authors)
[2, 4, 4, 3, 4, 2, 1, 3, 4, 3]
```

3.9.3. Aktualizacja obiektu

Metoda `update()` wykonuje zapytanie aktualizujące SQL dla określonych pól i zwraca liczbę dopasowanych wierszy. Na przykład, aby zmienić typ wszystkich zadań o punktacji równej 4 potrzebne jest polecenie:

```
>>> Task.objects.filter(points = 4).update(type = 1)
UPDATE "authentication_task"
   SET "type" = 1
 WHERE "authentication_task"."points" = 4
Execution time: 0.170321s [Database: default]
```

3.9.4. Usuwanie obiektu

Metoda `delete()` wykonuje zapytanie SQL usuwające dla wszystkich wierszy w QuerySet.

```
>>> Task.objects.filter(points = 1).delete()
```

3.10. Wyzwalacze Django-PostgreSQL

Wyzwalacze mogą rozwiązywać wiele różnych problemów na poziomie bazy danych lepiej niż na poziomie aplikacji Django. Dzięki wyzwalaczom możliwa jest np.:
— ochrona przez usuwaniem oraz zmienianiem wierszy lub kolumn
— śledzenie zmian w modelach lub kolumnach
— synchronizacja pól z innymi polami.
Aby używać wyzwalaczy bazy PostgreSQL w Django z pomocą przychodzi dodatek `django-pgtrigger`.

```
>pip install django-pgtrigger
```

Wykonując powyższe polecenie oraz dodając `pgtrigger` do **INSTALLED_APPS** można rozpocząć współpracę z `django-pgtrigger`.

3.10.1. Ochrona przed usunięciem modelu

Definicja pgtrigger.Protect wywołuje ochronę przed wykonaniem wybranej operacji.

Wyciąg 3-13. Fragment pliku `models.py`

```
import pgtrigger
@pgtrigger.register(
    pgtrigger.Protect(
        name="delete_protect", operation=pgtrigger.Delete)
)
class Section(models.Model):
   ...
```

```
>>> section = Section.objects.get(id=3)
>>> section.delete()
django.db.utils.InternalError: pgtrigger:
 Cannot delete rows from section table
CONTEXT:
 PL/pgSQL function pgtrigger_protect_deletes_f1129() line 14 at RAISE
```

Chcąc usunąć dział o id równym 3 otrzymujemy błąd informujący o tym, że operacja ta nie jest możliwa. Takie działanie daje nam pewność, że działy lub umiejętności nie zostaną usunięte.

3.10.2. Możliwość wyłącznie tworzenia lub odczytu

Wyzwalacz `pgtrigger.Protect` oferuję nie tylko ochronę przed usunięciem modelu. Kolejną przydatną opcją jest ochrona przez aktualizowaniem danych modelu.

Wyciąg 3-14. Fragment pliku `models.py`

```
@pgtrigger.register(
    pgtrigger.Protect(
        name="delete_update_protect",
        operation=(pgtrigger.Update | pgtrigger.Delete)
    )
)
class Task(models.Model):
   ...
```

Powyższy kod **??** ukazuję ochronę modelu, dającą możliwość jedynie tworzenia nowego zadania w bazie. Próba zmiany jego wartości lub usunięcia skutkować będzie błędem. Możliwa jest również ochrona poszczególnego pola modelu. Jeżeli chcemy, aby użytkownik nie mógł zmieniać typu lub poziomu zadania, należy użyć:

Wyciąg 3-15. Fragment pliku `models.py`

```
@pgtrigger.register(
    pgtrigger.Protect(
        operation=pgtrigger.Update,
        name="type_level_protect",
        condition=(
            pgtrigger.Q(old__type__df=pgtrigger.F("new__type")),
            pgtrigger.Q(old__level__df=pgtrigger.F("new__level"))
        )
    )
)
class Task(models.Model):
   ...
```

3.10.3. Miękkie usuwanie

Zdarza się, że użytkownik usunie utworzony przez siebie sprawdzian, lecz po pewnym czasie chce go odzyskac. Można użyć wtedy `pgtrigger.SoftDelete`, który zapobiegnie całkowitemu usunięciu obiektu, poprzez zmianę wartości pola obiektu.

Wyciąg 3-16. Fragment pliku `models.py`

```
@pgtrigger.register(
    pgtrigger.SoftDelete(
        name="soft_del",
        field="is_active", value=False)
)
class Test(models.Model):
    is_active = models.BooleanField(default=True)
    ...
```

Gdy użytkownik usunie sprawdzian, pole `is_active` zmieni swoją wartość na `False` i nie będzie juz wyświetlane na jego liście. Jeśli jednak będzie on chciał

owy sprawdzian odzyskać, musi zrobić to z listy usuniętych sprawdzianów lub przez inne przeznaczone do tego rozwiązanie.

3.10.4. Śledzenie zmian

Jeżeli potrzebna jest możliwość śledzenia zmian w modelach użyteczne może okazac się `django-pghistory`, które zapewnia automatyczne i konfigurowalne śledzenie historii dla modeli Django przy użyciu wyzwalaczy. Aby używać `django-pghistory` potrzebne również będzie zainstalowanie aplikacji `django-pgconnection` oraz odpowiednie skonfigurowanie pliku settings.py. Po instalacji należy wykonać migrację.

Wyciąg 3-17. Fragment pliku `settings.py`

```
DATABASES = pgconnection.configure(
    {
        "default": {...}
    }
)
```

Przykładem użycia `django-pghistory` może być potrzeba zbierania informacji na temat utworzenia nowych zadań matematycznych.

Wyciąg 3-18. Fragment pliku `settings.py`

```
@pghistory.track(
    pghistory.AfterInsert("create")
)

class Task(models.Model):
        ...
```

Powyższy kod **??** pozwala na zapisywanie informacji dotyczących tworzonych zadań przez użytkowników. W tym celu Django stworzył nową tabele `taskevent`, gdzie zapisywane są wszelkie dane. Istnieje wiele innych zastosowań `django-pghistory`. Aby poznać wszystkie z nich, warto zapoznać się z dokumentacją.

3.11. Podsumowanie

W tym rozdziale zawarte zostały informacje na temat współpracy frameworka Django z bazą danych PostgreSQL. Treści te pokazują w jaki sposób poprawnie łączyć się z bazą, tworzyć modele oraz wykorzystywać relację na przykładzie projektu Generatora Sprawdzianów. Opisane zostało również, jak w optymalny sposób odpytywać bazę danych oraz wykorzystywać przydatne pakiety mające na celu pomoc we współpracy z PostgreSQL.

Rozdział 4

React jako narzędzie do tworzenia interfejsu użytkownika

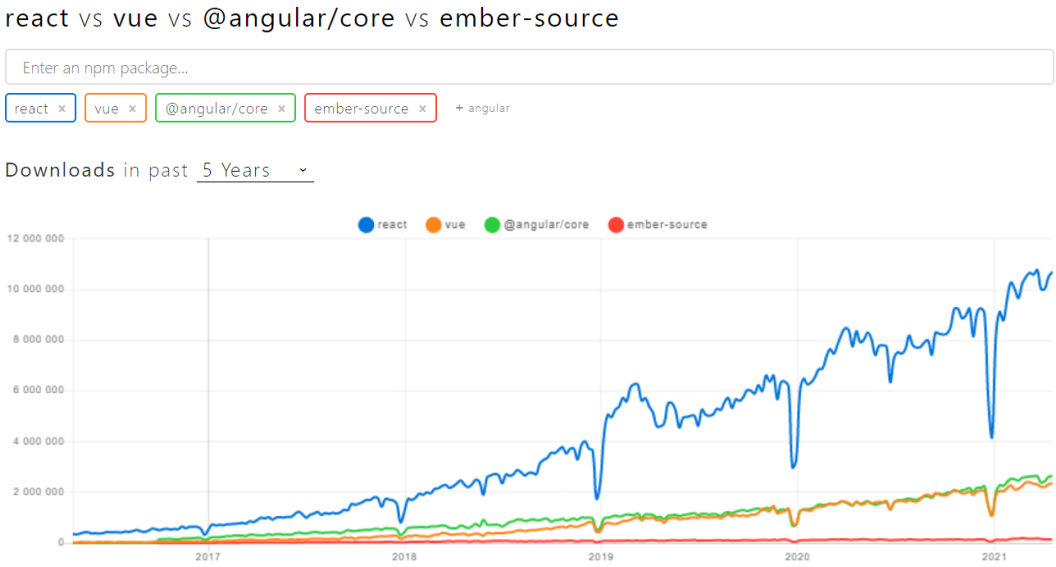
4.1. Czym jest React?

React jest javascriptową biblioteką typu open-source[1] używaną do tworzenia interfejsów użytkownika aplikacji internetowych. Jego oryginalnym autorem jest programista firmy Facebook - Jordan Walke. Pierwszy, bardzo wczesny prototyp Reacta - FaxJS powstał już w 2011 roku i został wykorzystany do zaimplementowania funkcji "Szukaj" na platformie Facebook. Niewiele później, aby móc łatwiej obsługiwać Facebook Ads, Jordan rozwinął prototyp i stworzył React. Po nabyciu przez Facebooka Instagrama w 2012 roku, Instagram wyrażał chęć wdrożenia nowej technologii na swoją platformę, co powodowało nacisk na odłączenie Reacta od Facebooka i przejście na licencję open-source. W 2013 roku biblioteka przeszła na licencję open source [**reactstory**].
Od tamtego momentu React cały czas prężnie się rozwija, pozostaje jednym z najpopularniejszych narzędzi do tworzenia interfejsów graficznych, używany jest nie tylko przez ogromne firmy takie jak Uber, Netflix, Airbnb, Amazon, Twitter oraz wiele innych, ale również przez zwykłych użytkowników. Popularność Reacta nie przestaje rosnąć, w styczniu 2021 roku biblioteka została pobrana ponad 10 milionów razy - zdecydowanie wyróżniając się na tle podobnych narzędzi przeznaczonych do tworzenia interfejsów użytkownika.

4.2. Dlaczego warto korzystać z Reacta?

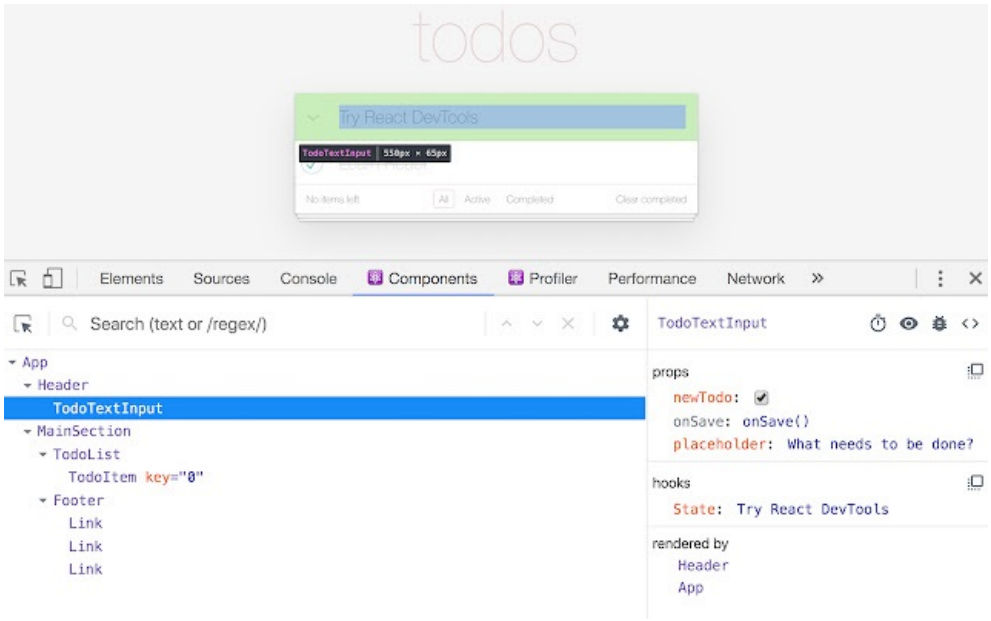
React posiada wiele zalet, które sprawiły, że został użyty w projekcie. Relatywnie niska bariera wejścia, dzięki której nauka powinna być w miarę prosta

[1] open source - rodzaj oprogramowania oparty na licencji przyznającej użytkownikom prawo do badania, zmiany i rozpowszechniania oprogramowania [**opensourcewiki**]



Rysunek 4.1. npm trends, statystyki dotyczące ilości pobrań popularnych narzędzi do tworzenia interfejsów

dla osób mających wcześniejszą styczność z programowaniem. Łatwość testowania oraz debugowania - w tym celu bardzo przydatne jest narzędzie React Developer Tools, którego możemy używać z poziomu przeglądarki internetowej. React Developer Tools jest dostępne w postaci rozszerzenia, po którego zainstalowaniu ujrzymy dwie zakładki, zakładkę Components oraz zakładkę Profiler. W zakładce Components możemy obserwować hierarchię wszystkich komponentów naszej aplikacji, dowolnego komponentu oraz to jak zmienia się on w czasie. Druga zakładka Profiler pozwala nam sprawdzić wydajność naszej aplikacji. Domyślnie React pokazuje programiście wiele użytecznych, dobrze opisanych komentarzy oraz ostrzeżeń usprawniających pracę, są one dostępne, kiedy aplikacja znajduje się w fazie developerskiej. Wszystko to sprawia, że aplikacja w fazie developerskiej działa wolniej. Przejście do fazy produkcyjnej usuwa większość komentarzy, ostrzeżeń oraz możliwość sprawdzania każdego komponentu i dowolnej informacji o nim przy pomocy React Developer Tools, ale za to znacząco przyspiesza wydajność aplikacji. Właśnie z tych powodów tryb developerski stosujemy w czasie budowania aplikacji, a trybu produkcyjnego w ostatecznej wersji produktu, która ma trafić do użytkownika.



Rysunek 4.2. React Developer Tools - obraz poglądowy
Źródło: `https://lh3.googleusercontent.com/XWuZGqIrIsaoKHUqqQ2rs_GhS5JaH1p5pPBIUpj22mjNRNdR3Ana8FKz4B7JwsA6HIFVXGuU7pa4ELiW6iUNhs0Iyg=w640-h400-e365-rj-sc0x00ffffff`

React posiada obszerną, bardzo dobrze napisaną dokumentację wraz z samouczkiem [**reactjs**] dostępną w wielu językach, między innymi w języku Polskim. Kolejnym z plusów jest budowa komponentów, interfejs zbudowany w Reaccie jest stworzony z mniejszych komponentów, które łączą w sobie wygląd wraz z logiką odpowiedzialną za ich zachowanie, co pomaga w zorganizowaniu pracy, dodatkową pomocą jest JSX - rozszerzenie składni pozwalające na używanie znaczników HTML razem z językiem Javascript. Ogromna popularność Reacta przekłada się również na mnogość dostępnych samouczków i innych materiałów pozwalających lepiej zrozumieć tę bibliotekę. Dodatkowo w poszukiwaniu odpowiedzi na nurtujące nas pytania możemy zajrzeć do społeczności programistów React takich jak dev.to, hashnode, lub najpopularniejszy z nich pod względem obserwujących użytkowników (aktualnie jest ich ponad ćwierć miliona) - Reddit.

4.3. Npm

Npm (pierwotnie skrót od Node Package Manager) jest to manager pakietów języka Javascript. Pierwsze wydanie npm miało miejsce już w 2010

roku [**npmwiki**]. Npm jest domyślnym managerem pakietów środowiska Node.js, npm jest instalowane razem z Node.js. Najważniejszym aspektem npm jest ogromna baza pakietów, bibliotek o niezwykle szerokim wachlarzu zastosowań. Interesujące nas rozszerzenie możemy zainstalować przy pomocy komendy
npm install nazwapakietu
Kiedy nad projektem pracuje wiele osób, ilość używanych bibliotek może rozrastać się dużo szybciej wraz z szukaniem nowych rozwiązań do pojawiających się problemów. Potrzeba wpisywania nazwy każdego pakietu oraz instalowania go osobno mogłaby być męcząca, dlatego npm śledzi wszystkie używane w danym projekcie pakiety w jsonowym pliku packages. Aby zainstalować wszystkie pakiety zarejestrowane przez plik packages należy użyć komendy
npm update

4.4. Pojęcia Reacta

4.4.1. JSX

JSX, czyli Javascript XML jest rozszerzeniem składni języka Javascript, które pozwala na używanie znaczników HTML razem z językiem Javascript. Jest to niezwykle przydatne narzędzie, które ułatwia nam na operowanie danymi przetwarzanymi w kodzie Javascriptowym oraz ułatwia ich wyświetlanie przy pomocy znaczników HTML. JSX możemy wykorzystać w wielu celach.

Wyciąg 4-1. Fragment pliku MaterialUiUserExams.js, przykładowe użycie JSX

```
<ListItemText>
        primary={<h3>{section.Section}</h3>}
        secondary={"Dostępnych zadań: " + section.sectionTaskCount}
</ListItemText>
```

Jedną z zalet JSX jest łatwość otrzymania wartości zmiennej używanej w kodzie javascriptowym. W powyższym przykładzie wykorzystujemy to, aby otrzymać nazwę każdego działu, który następnie wyświetlamy użytkownikowi. JSX jest w stanie obsłużyć dowolne, prawidłowe wyrażenie w języku Javascript, które zostało zamknięte wewnątrz nawiasów klamrowych. Dzięki czemu możemy nie tylko uzyskać wartości zmiennych, ale również je modyfikować oraz wywoływać funkcje javascriptowe, jeśli tego potrzebujemy.

Wyciąg 4-2. Wywołanie funkcji map wewnątrz elementu <List>

```
<List>
 {sections.map((section) => {

          return (
         <>
         <ListItem
         button onClick={this.hideSection(section)}>

                 <ListItemText>
                 primary={<h3>{section.Section}</h3>}
                 secondary={"Dostępnych zadań: " + section.sectionTaskCount}
                 </ListItemText>
                 {(this.state.hiddenSections.indexOf(section.id) !== -1)
                 ? <ExpandLess /> : <ExpandMore />}
         </ListItem>

                 <Collapse in={(this.state.hiddenSections
                        .includes(section.id))}
                timeout="auto" unmountOnExit>
                          <List component="div" disablePadding>
                          <ListItem button
                          key={section.id}
                          onClick={this.handleToggleAll(section)}>
                          <ListItemIcon>
                                 <Checkbox
                                 edge="start"
                                 checked={this.state.
                                 fullyChecked.indexOf(section.id) !== -1}
                                  tabIndex={-1}/>
                          </ListItemIcon>
                          <ListItemText
                          primary={<b>Zaznacz wszystkie</b>}  />
         </ListItem>
...
</List>
```

Po kompilacji wyrażenia JSX stają się wywołaniami funkcji w języku Javascript, oznacza to, że możliwe jest używanie wyrażeń w JSX wewnątrz instrukcji warunkowych, pętli, przypisywanie je do zmiennych, używanie jako argumentów oraz zwracanie z funkcji. Oczywiście, jeśli ktoś nie chce korzystać z JSX, może tego nie robić. Możliwe jest stworzenie wszystkich komponentów w czystym Javascripcie, natomiast JSX jest narzędziem bardzo wygodnym, moc-

no usprawniającym pracę z Reactem, dlatego jest zdecydowanie zalecanym rozwiązaniem.

4.4.2. Komponenty

Interfejs w React składa się z komponentów. Komponenty są jak funkcje javascriptowe, które przyjmują pewne wartości na wejściu, a następnie zwracają elementy reactowe, które opisują, co ma pojawić się na ekranie. Wyróżniamy dwa typy komponentów - komponenty klasowe oraz komponenty funkcyjne.

Wyciąg 4-3. Przykład komponentu klasowego

```
class ClassComponent extends Component{
  render(){
    return <h2>Przykład komponentu klasowego</h2>
  }
}
```

Wyciąg 4-4. Przykład komponentu funkcyjnego

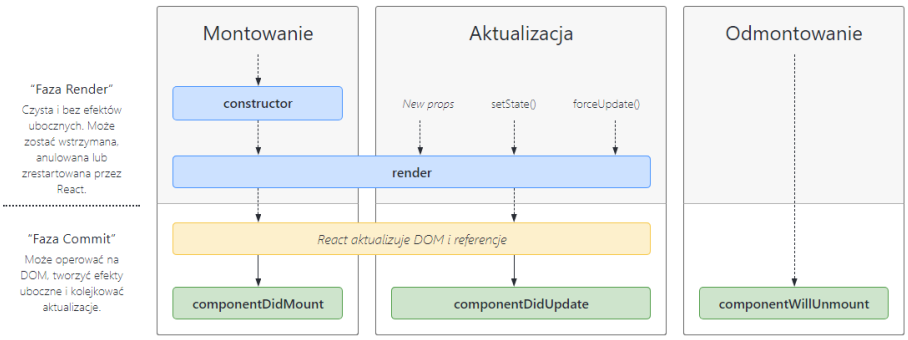
```
function FunctionComponent(props){
  return <h2>Przykład komponentu funkcyjnego</h2>
}
```

Jedną z zasad obowiązujących podczas budowy komponentów, zarówno funkcyjnych jak i klasowych pozostaje nienaruszalność właściwości (ang. props) przekazywanych podczas deklaracji funkcji, dane, które mają zmieniać się dynamiczne powinny być przechowywane w stanach. Obecnie podstawową różnicą pozostaje składnia, w przeszłości komponenty funkcyjne nie posiadały stanów ani metod pozwalających na uruchamianie kodu w odpowiednich momentach, z tych powodów komponentów funkcyjnych używano tylko do budowy komponentów, które nie musiały być dynamiczne. Zmieniło się to wraz z wprowadzeniem Hooków. Warto również wiedzieć, że komponenty mogą odwoływać się do innych komponentów, co promuje pisanie ich z myślą o możliwości wielokrotnego wykorzystania. Możemy to wykorzystać na przykład tworząc komponent ekranu ładowania, następnie możemy go użyć w innych komponentach, renderując go wtedy, gdy użytkownik będzie oczekiwał na odpowiedź od serwera.

4.4.3. Stan i cykl życia

Stan zawiera dane związane z danym komponentem, które mogą ulegać zmianie. Stan powinien być obiektem javascriptowym, aby go zmodyfikować należy wywołać metodę setState. Dodatkowo, komponenty posiadają metody cyklu życia, które są odpowiedzialne za uruchamianie kodu w odpowiednich momentach.
Cykl życia komponentu możemy podzielić na 3 etapy:

— Montowanie (ang. Mounting), czyli pierwsze stworzenie instancji komponentu i włożenie go do drzewa DOM
— Aktualizacja, czyli zmiany we właściwościach lub stanie komponentu
— Odmontowanie (ang. Unmounting), czyli usunięcie komponentu z drzewa DOM



Rysunek 4.3. Diagram metod cyklu życia
Źródło:
`https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/`

Każdy cykl wywołuje inny zestaw metod, najważniejszymi metodami wywoływanymi w czasie montowania są constructor - odpowiedzialny za inicjalizację stanu lokalnego oraz związanie(ang. bind) metody obsługi zdarzeń, render oraz componentDidMount - ta metoda jest odpowiednim miejscem do wysłania zapytania, jeśli potrzebujemy pobrać dane z zewnątrz. W czasie aktualizacji wywołana zostaje analogiczna metoda - componentDidUpdate, a w czasie Odmontowania wywołana zostaje metoda componentWillUnmount - powinniśmy z niej korzystać głównie w celach posprzątania np. wyzerowanie liczników.

Stan może przyjąć dowolną wartość, true, false, null, może być liczbą, ciągiem znaków (string) lub tablicą.

Wyciąg 4-5. Przykład komponentu klasowego korzystającego ze stanów

```
class UserExams extends Component {
  constructor(props) {
    super(props);
    this.state = {
      exams: null,
      open: false,
      listExpand:{},
      generatedId: null,
      generatedName: null,
      sections: null,
      groups: null,
      ileotw: '',
      ilezamk: '',
      level: '',
      skills: '',
      groups: '',
    }
  }
}
```

4.4.4. Renderowanie warunkowe

Renderowanie warunkowe umożliwia wyświetlanie elementów na podstawie stanu danej aplikacji. Oznacza to, że jesteśmy w stanie uzależnić interfejs (to, co widzi użytkownik) od wartości zwracanej przez zastosowaną przez nas instrukcje warunkową. Do renderowania warunkowego możemy użyć klasycznej instrukcji if,

Wyciąg 4-6. Renderowanie warunkowe przy pomocy instrukcji if

```
    if (!exams ) {
      return (
        <Loading  message={"Ładowanie kolekcji sprawdzianów"} >
      </Loading>
      );
    }
```

operatora warunkowego, często stosowanego jako krótsza wersja instrukcji if,

Wyciąg 4-7. Renderowanie przy pomocy operatora warunkowego

```
{editView == "email" ?
        (
                <h1>Widok, który zobaczy użytkownik,
                jeśli warunek zostanie spełniony </h1>
        ):
        (
                <h1>Widok, który zobaczy użytkownik
                 jeśli warunek nie zostanie spełniony</h1>
        )
```

jak i operatora logicznego &&.

Wyciąg 4-8. Renderowanie warunkowe przy pomocy operatora logicznego &&

```
{
        isSubmitting &&  <LoadingScreenB></LoadingScreenB>
}
```

4.4.5. Hooki

Hooki zostały wprowadzone w wersji 16.8. Pozwalają używać stanów oraz innych funkcjonalności Reacta w komponentach funkcyjnych.
Jak podają developerzy, głównymi powodami, dla których wprowadzili hooki było to, że:

— Współdzielenie logiki związanej ze stanem pomiędzy komponentami jest trudne
— Wraz ze wzrostem złożoności komponentów stają się one zbyt trudne do zrozumienia
— Klasy są mylące zarówno dla ludzi jak i maszyn

Jeśli chcemy skorzystać ze stanu w komponencie funkcyjnym, należy użyć hooka stanu.

Wyciąg 4-9. Wywołanie hooka stanu

```
const [editView, setEditView] = React.useState("email");
```

Powyższy przykład pokazuje, jak użyć hooka stanu, gdzie editView jest nazwą stanu, a email jest jego wartością początkową. W przypadku potrzeby zaktualizowania wartości należy wywołać funkcję setEditView.

Wyciąg 4-10. Zmiana stanu przy pomocy funkcji setEditView

```
<Button variant="contained"
color="primary"
onClick={(e)=>setEditView("name")}  >
        Zmień nazwę użytkownika
</Button>
```

Jeśli chcemy dokonać operacji w określonym cyklu życia komponentu, należy użyć hooka useEffect. Możemy o nim myśleć jak o połączeniu metod componentDidMount, componentDidUpdate oraz componentWillUnmount w jedną.

Wyciąg 4-11. Przykładowe użycie hooka useEffect

```
  useEffect(() => {
    const fetchSections = () => {
      console.log("useeffect");
      axiosInstanceNoAuth
        .get("/user/sections2/")
        .then((response) => {
          const parsed = response.data.map((section) => {
            section.skill = section.skilll;
            return section;
          });
          setSections(parsed);
        })
        .catch((error) => {
          setSections(false);
        });
    };
    fetchSections();
  }
```

4.4.6. Obsługa zdarzeń

Kolejnym pojęciem często używanym w budowaniu interfejsów są zdarzenia. Zdarzenia określają zajście pewnej czynności, którą nasz interfejs jest w stanie wyłapać i następnie obsłużyć. Przykładowymi zdarzeniami mogą być onClick - w przypadku kliknięcia myszką, onChange - w przypadku modyfikacji elementu lub onSubmit - w przypadku przesłania formularza. W odróżnieniu od np.

HTMLa w React zdarzenia zapisujemy tzw. camelCasem. [2] Instrukcje obsługi zdarzenia są przekazywane jako funkcje.

Wyciąg 4-12. Przykładowa obsługa zdarzenia onClick

```
<BootstrapTooltip title="Skopiuj sprawdzian">
        <IconButton  onClick={() =>
        {this.createExamCopy(exam);}}>
                <FileCopyIcon />
        </IconButton>
</BootstrapTooltip>
```

Powyższy przykład pokazuje jeden ze sposobów obsługi zdarzeń wykorzystywanych w aplikacji Gen-mat. Kliknięcie przycisku IconButton powoduje wywołanie funkcji createExamCopy, która jest odpowiedzialna za utworzenie kopii sprawdzianu. Innym sposobem na przekazanie instrukcji obsługi zdarzenia jest napisanie funkcji obsługi, następnie jej związanie (ang. bind) i wywołanie jej przy pomocy *this*.

Wyciąg 4-13. Związanie

```
this.hideSection = this.hideSection.bind(this);
```

Wyciąg 4-14. Wywołanie obsługi zdarzenia po związaniu

```
<ListItem
button
onClick={this.hideSection(section)}>
```

4.5. Tworzenie formularzy oraz wymiana danych z serwerem

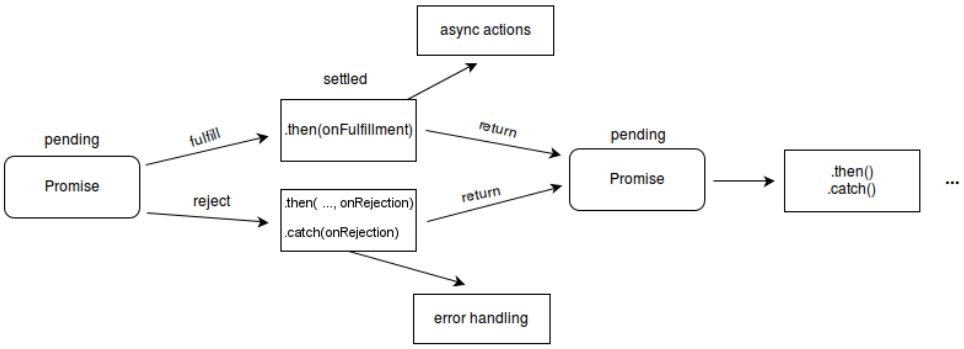
Jedną z podstawowych rzeczy, które musi zrobić prawie każdy interfejs w dzisiejszych aplikacjach internetowych jest obsługa formularzy oraz wymiana danych między serwerem, aplikacją oraz użytkownikiem. Interfejs projektu Gen-mat w tym celu używa głównie trzech bibliotek:

[2] camelCase jest konwencją, w której w nazwach kolejne wyrazy zapisujemy łącznie, a każdy kolejny(nie licząc pierwszego) rozpoczynamy wielką literą

— Axios
— Formik
— Yup

4.5.1. Axios

Axios [**axios**] jest Klientem HTTP dla node.js oraz przeglądarki, opartym na javascriptowym obiekcie Promise, który służy do obsługi operacji asynchronicznych.



Rysunek 4.4. Jak działa Promise?
Źródło: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/promises.png`

Dzięki Axios możemy tworzyć requesty - żądania HTTP, które są podstawową formą komunikacji z serwerem. HTTP opisuje formę żądania klienta oraz formę odpowiedzi na żądanie ze strony serwera. Do najważniejszych metod HTTP, na których się skupimy, należą:

— GET - metoda wysyłająca żądanie otrzymania danych, powinna być ograniczona jedynie do pobierania danych, nie powinna ich przesyłać
— PUT - metoda używana w celu przesłania danych między klientem oraz serwerem w celu aktualizacji wybranego zasobu
— POST - metoda używana w celu przesłania danych między klientem oraz serwerem
— DELETE - żądanie usunięcia danych

Axios pozwala na tworzenie instancji z własną konfiguracją, która będzie stosowana domyślnie przy każdym wysłaniu żądania przy pomocy tej instancji.

Gen-mat używa dwóch instancji Axios.

Wyciąg 4-15. Instancje axios projektu gen-mat

```
import axios from "axios";

export const axiosInstanceNoAuth = axios.create({
  baseURL: window.location.origin + "/api/",
  timeout: 30000,
  headers: {
    "Content-Type": "application/json",
    accept: "application/json",
  },
});

export const axiosInstance = axios.create({
  baseURL: window.location.origin + "/api/",
  timeout: 30000,
  headers: {
    Authorization: "JWT " + localStorage.getItem("access_token"),
    "Content-Type": "application/json",
    accept: "application/json",
  },
});
```

Gdzie baseURL oznacza adres URL, timeout służy do określania maksymalnego czasu (w milisekundach) jaki żądanie ma na wykonanie, zanim zostanie anulowane. Dodatkowo, na podstawie zawartości nagłówka serwer jest w stanie odróżnić czy zapytanie zostało wysłane przez użytkownika zalogowanego, czy przez użytkownika niezalogowanego. Aby wysłać żądanie należy odwołać się do instancji, określić rodzaj żądania, podać dokładną ścieżkę oraz, w zależności od tego, co mamy do przekazania serwerowi, przesłać dane.

Wyciąg 4-16. Przykładowe wywołanie zapytania GET, aby otrzymać sprawdziany użytkownika

```
  updateExams = () => {
    this.setState({ exams: null });
    axiosInstance
      .get("/user/tests/")
      .then((response) => {
        console.log("UE update response", response, "exams", response.data);
        this.setState({ exams: response.data });
      })
      .catch((error) => {
        console.log("UE update error response", error);
```

```
11        });
12    }
```

Wyciąg 4-17. Wywołanie zapytania POST w celu utworzenia rejestracji nowego użytkownika

```
axiosInstanceNoAuth
        .post("/user/create/", {
                username: values.name,
                password: values.password,
                email: values.email,
        })
```

4.5.2. Yup

Yup jest narzędziem przydatnym przy parsowaniu oraz walidacji [**yup**]. Yup pozwala na tworzenie schematów, do których następnie można przyrównać interesujący nas element. Yup pozwala nam porównywać zarówno wartości zmiennych, jak i ich typy.

Wyciąg 4-18. Yup - przykład użycia

```
validationSchema={Yup.object().shape({
        name: Yup.string()
                .min(2, "Nazwa użytkownika musi
                się składać z minimum 2 znaków!")
                .max(50, "Hasło może zawierać maksymalnie 50 znaków!")
                .required(FRS),
        password: Yup.string()
                .min(8, "Hasło musi zawierać co najmniej 8 znaków!")
                .max(50, "Hasło może zawierać maksymalnie 50 znaków!")
                .required(FRS),
        passwordConfirm: Yup.string()
                .oneOf([Yup.ref("password")], "Hasła są różne")
                .required(FRS),
        email: Yup.string()
                .email("Nieprawidłowy adres e-mail")
                .required(FRS),
})}
```

4.5.3. Formik

Formik jest biblioteką usprawniającą tworzenie formularzy w React. Pomaga w przekazywaniu wartości ze stanów oraz do stanów, walidacji i wyłapywaniu błędów oraz przesyłaniu formularzy.

Jak podaje sam autor, powodem, dla którego stworzył on Formik, była chęć zestandaryzowania komponentów wejścia oraz sposobu przepływu danych wewnątrz formularzy [**formik**].

Formik daje nam dostęp do przemyślanych pól oraz metod pomocniczych przydatnych przy tworzeniu formularzy. Na początku możemy przekazać początkowe wartości w initialValues, podstawowymi metodami pomocniczymi są:

— handleSubmit - procedura obsługi wysyłania
— handleChange - procedura obsługi zmian
— values - w tej metodzie przechowywane są aktualne wartości obecne w formularzu.

Kolejnym sposobem użycia Formika razem z Yupem, wykorzystywanym w projekcie Gen-mat, jest walidacja pól na poziomie formularza przy pomocy validationSchema. Najpierw stworzony zostaje obiekt Yupowy o określonych wartościach a następnie zostaje on przypisany jako validationSchema.

Wyciąg 4-19. Formik - deklaracja

```
<Formik
  initialValues={{
    name: "",
    password: "",
  }}
  validationSchema={Yup.object().shape({
    password: Yup.string()
      .min(8, "Hasło musi zawierać co najmniej 8 znaków!")
      .max(50, "Hasło może zawierać maksymalnie 50 znaków!")
      .required(FRS),
    name: Yup.string()
      .min(2, "Nazwa musi zawierać co najmniej 8 znaków!")
      .max(50, "Nazwa może zawierać maksymalnie 50 znaków!")
      .required(FRS),
  })}
  onSubmit={(values, helpers) => {
    setTimeout(() => {
      helpers.setSubmitting(true);
      axiosInstance
```

```
20          .post("/token/obtain/", {
21            username: values.name,
22            password: values.password,
23          })
24          .then((response) => {
25            axiosInstance.defaults.headers["Authorization"] =
26              "JWT " + response.data.access;
27            localStorage.setItem(
28              "access_token",
29              response.data.access
30            );
31            localStorage.setItem(
32              "refresh_token",
33              response.data.refresh
34            );
35            helpers.setSubmitting(false);
36            props.checkUser();
37            props.history.push("/");
38          })
39          .catch((error) => {
40            const errResponse = error.response;
41            helpers.setSubmitting(false);
42            if (
43              errResponse.status === 401 &&
44              errResponse.statusText === "Unauthorized"
45            ) {
46              enqueueSnackbar("Nieprawidłowy adres e-mail lub hasło", {
47                variant: 'error',
48              })
49              helpers.setValues(
50                {
51  name: "",
52  password: "",
53                },
54                false
55              );
56              helpers.setTouched(
57                {
58  name: false,
59  password: false,
60                },
61                false
62              );
63              helpers.setFieldError(
```

```
64                "general",
65                "Nieprawidłowa nazwa użytkownika lub hasło"
66              );
67
68            }
69          });
70      }, 400);
71    }}
72  >
73    {({
74      values,
75      errors,
76      touched,
77      handleChange,
78      handleBlur,
79      handleSubmit,
80      isSubmitting,
81    }) => {
```

Oprócz wcześniej wspomnianych podstawowych właściwości, Formik zapewnia:

— dirty - zwracające true, gdy między wartościami values a początkowymi wartościami przekazanymi w initialValues nie zachodzi głęboka równość.
— handleBlur - procedura obsługi zdarzenia onBlur - używana, gdy chcemy monitorować, czy konkretna dana wejściowa została dotknięta.
— isSubmitting - który sprawdza stan przesyłania formularza, zwraca wartość *true*, gdy formularz jest aktualnie przesyłany, w innym wypadku zwraca *false*
— setFieldError - pozwalający na ustawienie dodatkowej wiadomośc, gdy w wybranym polu wystąpi błąd

4.6. React Router

Modułem odpowiedzialnym za routing - nawigację pomiędzy ścieżkami jest React Router [**reactrouter**]. Moduł ten odpowiada za możliwość przechodzenia pomiędzy rożnymi widokami obecnymi w projekcie, jak i wyświetlanie prawidłowego widoku pod danym adresem URL. Kolejnym zastosowaniem jest zabezpieczenie ścieżki przed niepożądanym użytkownikiem oraz możliwość przekierowania użytkownika pod wybrany adres. Dodatkowo React Router pozwala na wyłuskanie opcjonalnego parametru w adresie URL.

Jednym z najważniejszych komponentów w React Router jest Route, którego podstawową funkcją jest wyświetlanie właściwej części interfejsu pod podanym adresem URL. Kolejnym przydatnym komponentem jest Switch, który sprawia, że wyrenderuje się wyłącznie podana ścieżka pod podanym adresem. Zapobiega to niechcianym renderom widoków, które przypadkiem mogą również pasować do podanej ścieżki. Komponent Redirect nadpisuje obecny adres i nawiguje pod nowy adres.

Wyciąg 4-20. Ścieżki dostępne w projekcie Gen-mat

```
<Switch>
  <Route
    exact
    path={"/login/"}
    render={(props) => <Login {...global} {...props} />}
  />

  <Route
    exact
    path={"/mytasks/"}
    render={(props) => <MaterialUiTaskCollection {...global} {...props} />}
  />
  <Route
    exact
    path={"/signup/"}
    render={(props) => <Signup {...global} {...props} />}
  />
  <Route
    exact
    path={"/signupsuccess/:token"}
    render={(props) => (
      <RegisterSuccess {...global} {...props} />
    )}
  />
  <Route
    exact
    path="/activateaccount/:token/"
    render={(props) => (
      <AccountActivation {...props} {...global} />
    )}
  />
  <Route
    exact
```

```
34      path="/passreset/:token"
35      render={(props) => <PasswordReset {...props} {...global} />}
36    />
37    <Route
38      exact
39      path="/requestresetpassword/"
40      render={(props) => (
41        <PasswordResetRequest {...props} {...global} />
42      )}
43    />
44    <Route
45      exact
46      path="/addtask/"
47      render={(props) => <AddTask {...props} {...global} />}
48    />
49    <Route
50      path={"/myaccount/"}
51      render={(props) => (
52        <UserAccountManager {...props} {...global} />
53      )}
54    />
55    <Route
56      path={"/editor/:id/"}
57      render={(props) => <ExamEditor {...props} {...global} />}
58    />
59    <Route
60      path={"/userexams/"}
61      render={(props) => <UserExams {...props} {...global} />}
62    />
63    <Route
64      path={"/loggedout"}
65      render={(props) => <LoggedOut {...props} {...global} />}
66    />
67    <Route
68      path={"/"}
69      render={(props) => <HomePage {...props} {...global} />}
70    />
71  </Switch>
```

React Router posiada kilka przydatnych hooków: useHistory, useLocation, useParams, useRouteMatch. Hook useHistory daje nam dostęp do pakietu history, który oferuje kolejne usprawnienia nawigacyjne oferując właściwości i metody takie jak:

— location - obiekt reprezentujący obecny adres

— push - który przepycha nowy adres na ostatnie miejsce na stosie historii
— goBack - funkcja wracająca na poprzedni adres

Rozważmy następującą sytuację - użytkownik zapomniał hasła, korzysta z funkcji „reset hasła”, na adres e-mail otrzymuje link, którego ostatnią częścią jest losowo wygenerowany ciąg znaków - token. Aplikacja musi być w stanie wydobyć ten token oraz sprawdzić czy jest poprawny, jest to możliwe dzięki użyciu useParams.

Wyciąg 4-21. Przykładowe użycie useParams

```
1  const passwordToken = useParams().token;
```

4.7. Podsumowanie

React jest potężnym narzędziem do tworzenia interfejsów. Jego ogromna popularność przekłada się na gigantyczną ilość materiałów wspomagających jego przyswojenie, co w połączeniu z relatywnie niską barierą wejścia składa się na atrakcyjny wybór dla osób bez większego doświadczenia w tworzeniu interfejsów. Dodatkowym atutem jest kompatybilność z szeroką ilością narzędzi ułatwiających pracę oraz dających więcej możliwości. Fakt, że React jest stale rozwijany oraz używany przez ogromne firmy sugeruje, że nie zostanie porzucony w najbliższej przyszłości, co również może skłaniać do wybrania właśnie tej biblioteki.