Bazy Danych 2

Household App - Zarządzanie gospodarstwem domowym

Maja Bojarska

Damian Koper

19 stycznia 2020

Spis treści

Spis rysunków

Spis listingów

1 Zespół

Członkowie zespołu tworzącego aplikację ze wstępnym podziałem na role:

- Maja Bojarska:
  - Projekt, implementacja i testowanie encji bazy danych wraz z mapowaniem obiektowo-relacyjnym.
  - Projekt, implementacja i testowanie RESTful API.
- Damian Koper:
  - Projekt odwzorowania obiektów świata rzeczywistego za pomocą encji w bazie danych.
  - Projekt i implementacja interfejsu użytkownika.
  - Testy jednostkowe i integracyjne poszczególnych komponentów systemu.
  - Testy end-to-end interfejsu użytkownika.

2 Opis systemu

Gospodarstwo domowe jest miejscem zrzeszającym domowników. Każda z osób ma wkład w jego rozwój. W celu śledzenia i równoważenia kosztów wynikających z użytkowania dostępnych zasobów potrzebny jest system śledzący i obliczający wydatki. Domownicy mogą dzielić się kosztami utrzymania po równo, lub odgórnie zdefiniować zasady określające procentowy wkład każdej osoby.

Działania te, razem ze śledzeniem zużycia mediów (woda, prąd, gaz), pozwolą na dokładniejsze przeanalizowanie wydatków. W przyszłości może się to przełożyć na bardziej świadome zarządzanie zasobami, a dzięki temu na optymalizację ponoszonych kosztów.

3 Wymagania funkcjonalne

3.1 Zarządzanie użytkownikami i gospodarstwem

1. Użytkownik może zalogować się na swoje konto.
2. Użytkownik może wylogować się ze swojego konta.
3. Użytkownik może edytować dane swojego konta.
4. Administrator może utworzyć konto użytkownika.
5. Administrator może edytować dane i rolę użytkownika.
6. Administrator może usunąć konto użytkownika.
7. Administrator może zmienić ustawienia procentowego udziału danego użytkownika w wydatkach gospodarstwa.

3.2 Zakupy

1. Użytkownik może wprowadzić dane zakupu przedmiotu.
2. Użytkownik może edytować dane zakupu przedmiotu.
3. Użytkownik może usuwać dane zakupu przedmiotu.
4. Użytkownik może tworzyć sklepy.
5. Użytkownik może edytować sklepy.
6. Użytkownik może usuwać sklepy.
7. Użytkownik może tworzyć kategorie.
8. Użytkownik może edytować kategorie.
9. Użytkownik może usuwać kategorie.

3.3 Listy zakupów

1. Użytkownik może dodać listy zakupów
2. Użytkownik może edytować listy zakupów
3. Użytkownik może usuwać listy zakupów
4. Użytkownik może dodać pozycję listy zakupów.
5. Użytkownik może edytować pozycję listy zakupów.
6. Użytkownik może usuwać pozycję listy zakupów.
7. Użytkownik może przekonwertować listę zakupów na wstępnie utworzone dane o zakupach.

3.4 Wydatki gospodarstwa

1. Uzytkownik może wprowadzić dane o jednorazowych wydatkach.
2. Uzytkownik może edytować dane o jednorazowych wydatkach.
3. Uzytkownik może usuwać dane o jednorazowych wydatkach.
4. Użytkownik może wprowadzić reguły obliczeń kosztów zużycia mediów.
5. Użytkownik może edytować reguły obliczeń kosztów zużycia mediów.
6. Użytkownik może usuwać reguły obliczeń kosztów zużycia mediów.
7. Użytkownik może wprowadzić dane zużycia mediów.
8. Użytkownik może edytować dane zużycia mediów.
9. Użytkownik może usuwać dane zużycia mediów.

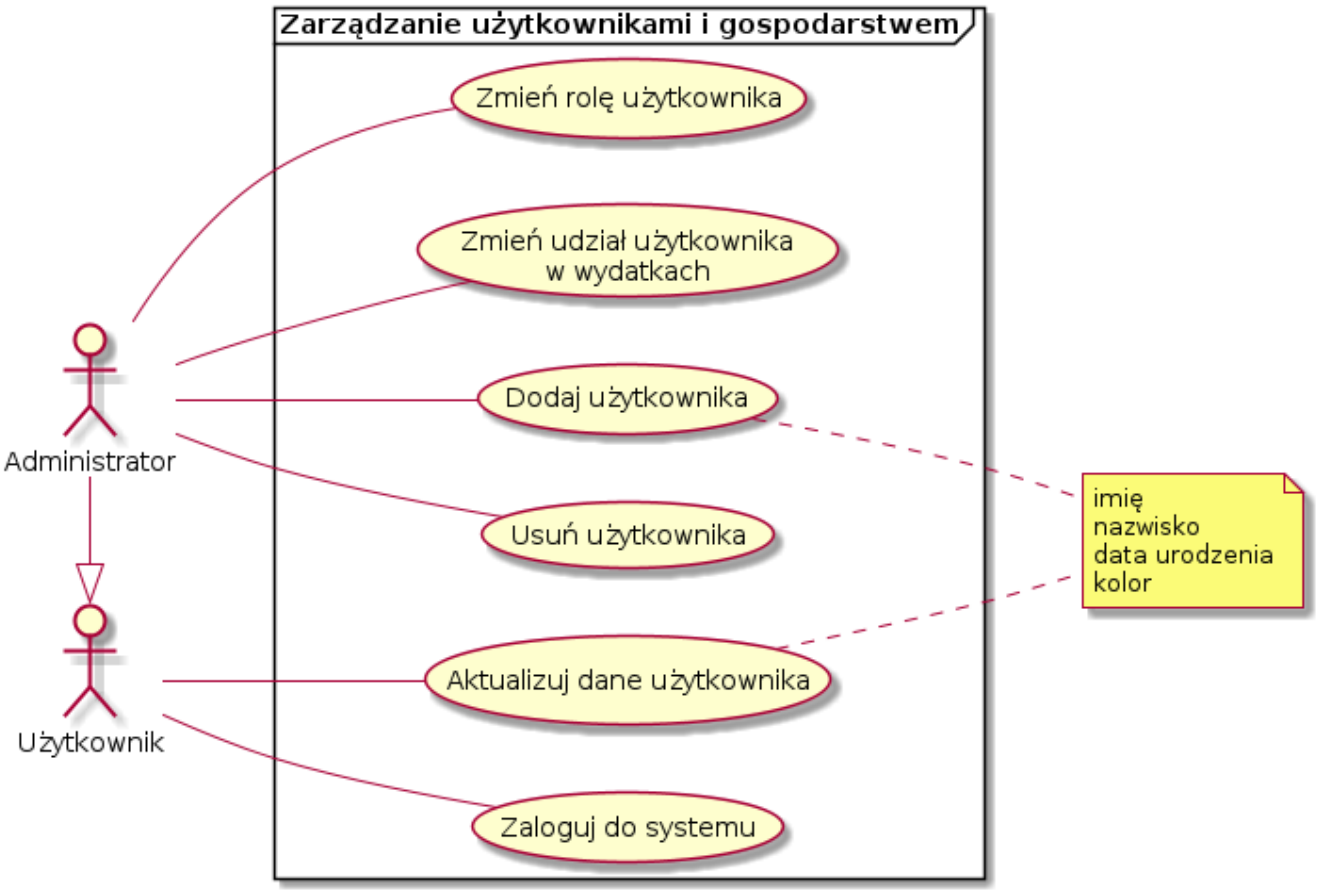
3.5 Statystyki i raporty

1. Użytkownik może wyświetlić statystyki kosztów zakupów dla każdego użytkownika, dla ustawionego okresu:
   (a) Zakupy w czasie jako wykres liniowy.
   (b) Zakupy zgrupowane w kategorie jako wykres kolumnowy.
   (c) Procentowy udział kategorii we wszystkich zakupach jako wykres kołowy.
2. Użytkownik może wyświetlić statystyki dla wszystkich zakupów lub tylko dla zakupów współdzielonych.
3. Użytkownik może wyświetlić liczbowe podsumowanie danego miesiąca zawierające:
   (a) Kwoty zakupów współdzielonych podzielone na wszystkich użytkowników z uwzględnionym ustawionym podziałem.
   (b) Kwoty do zapłaty dla poszczególnych użytkowników pozostałym użytkownikom wynikające z wprowadzonych zakupów i opłaconych rachunków.

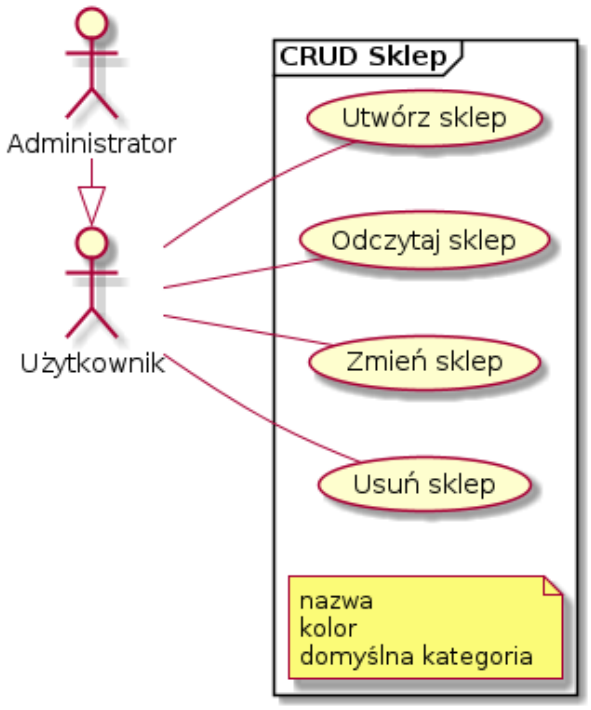
4 Wymagania niefunkcjonalne

1. Aplikacja zapewnia bezpieczeństwo sesji użytkownika.
2. Aplikacja jest odporna na popularne ataki - SQL Injection, XSS, Man in the middle.
3. Interfejs graficzny aplikacji działa po stronie przeglądarki użytkownika i komunikuje się z jej serwerem za pomocą RESTful API.
4. Aplikacja zapewnia spójność interfejsu z aplikacjami mobilnymi używając stylu *Material Design*.
5. Aplikacja zapewnia funkcjonalności Progressive Web App.
6. Architektura aplikacji musi umożliwiać szybką instalację wszystkich jej komponentów, serwisów i jej uruchomienie za pomocą jednej komendy.

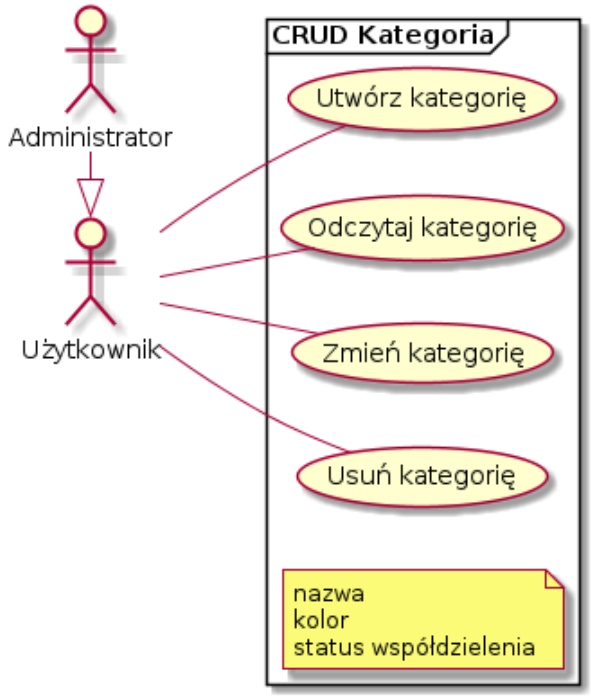
5 Przypadki użycia



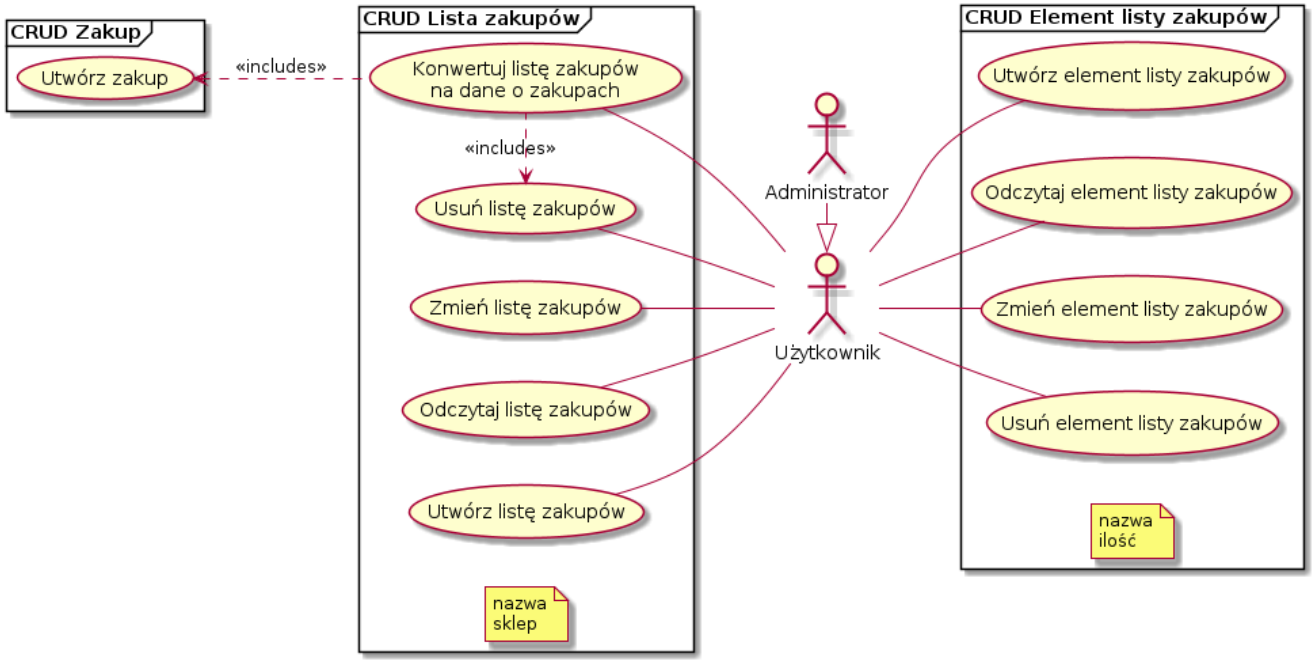
Rysunek 1: Diagram przypadków użycia - użytkownicy.



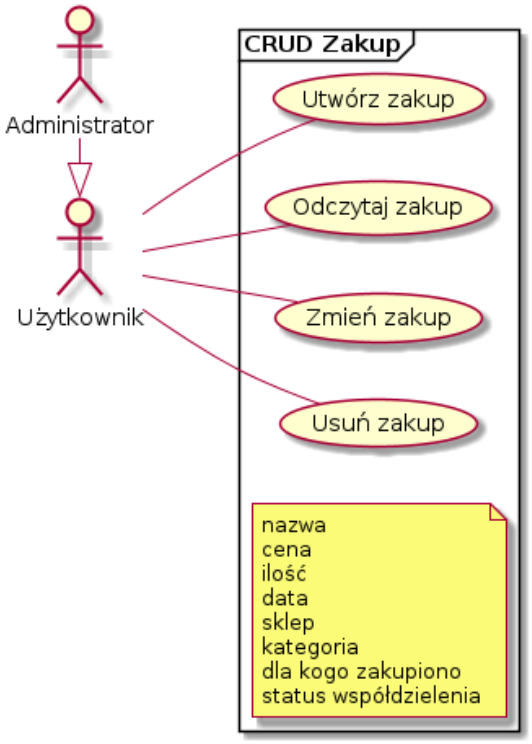
Rysunek 2: Diagram przypadków użycia - sklepy.



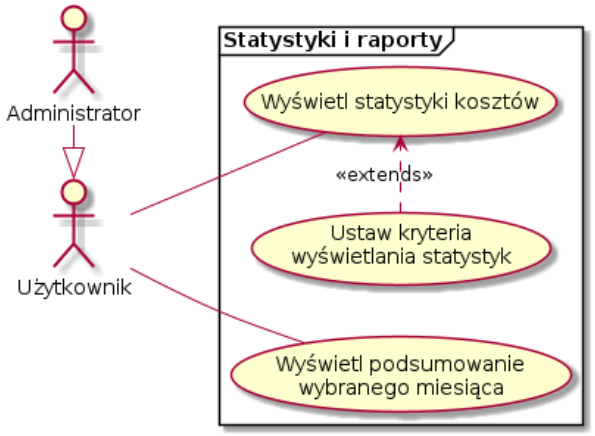
Rysunek 3: Diagram przypadków użycia - kategorie.



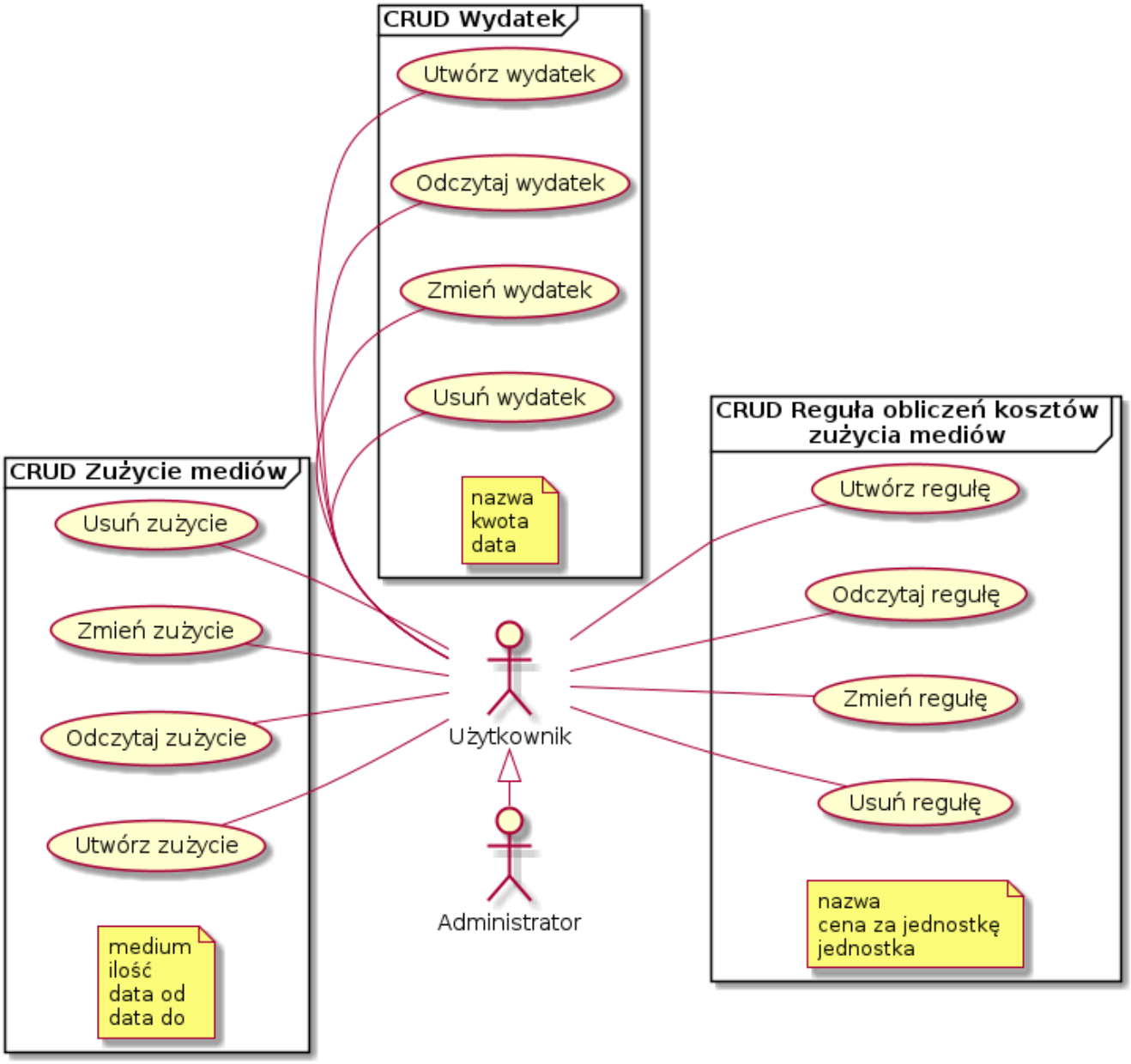
Rysunek 4: Diagram przypadków użycia - lista zakupów.



Rysunek 5: Diagram przypadków użycia - zakupy.



Rysunek 6: Diagram przypadków użycia - statystyki.



Rysunek 7: Diagram przypadków użycia - wydatki.

6 Architektura

Aplikacja wykorzystuje architekturę trójwarstwową, gdzie interface użytkownika, część zawierająca logikę biznesową i baza danych stanowią oddzielne, osobno rozwijane moduły. Użytkownik ma dostęp do aplikacji poprzez przeglądarkę internetową, co przy zachowaniu responsywności interface'u, daje mu dowolność wyboru wykorzystywanego typu urządzenia. Użytkownik komunikując się z aplikacją wysyła zapytania do dwóch serwerów:

1. Do serwera plików statycznych, który wysyła kod części działającej po stronie przeglądarki użytkownika - aplikacji *SPA*.

2. Do serwera zarządzającym zasobami aplikacji zapewniającym bezstanową komunikację w stylu REST, który odpowiedzialny jest za pobieranie i modyfikowanie stanu aplikacji.

Podział ruchu na dwa serwery pozwala odseparować pobieranie zasobów aplikacji od zapytań ingerujących w jej stan - przetwarzających dane, co ułatwi późniesze skalowanie aplikacji. w późniejszych fazach rozwoju projektu serwer plików statycznych może znajdować się na innej maszynie.

6.1 Multitenancy

Struktura danych aplikacji nie zakłada podziału przypisania encji na wiele gospodarstw, co oznacza, że jedna baza danych może obsłużyć tylko jedno gospodarstwo. Jest to celowa konstrukcja, która zakłada w przyszłości zastosowanie architektury multitenancy. Zakłada ona dynamiczne przekierowywanie ruchu do odpowiedniej bazy danych w zależności od ustalonej zmiennej - np. subdomeny. Aplikacja w zakładanej obecnie formie jest przeznaczona do instalacji *on premises*.

7 Technologie

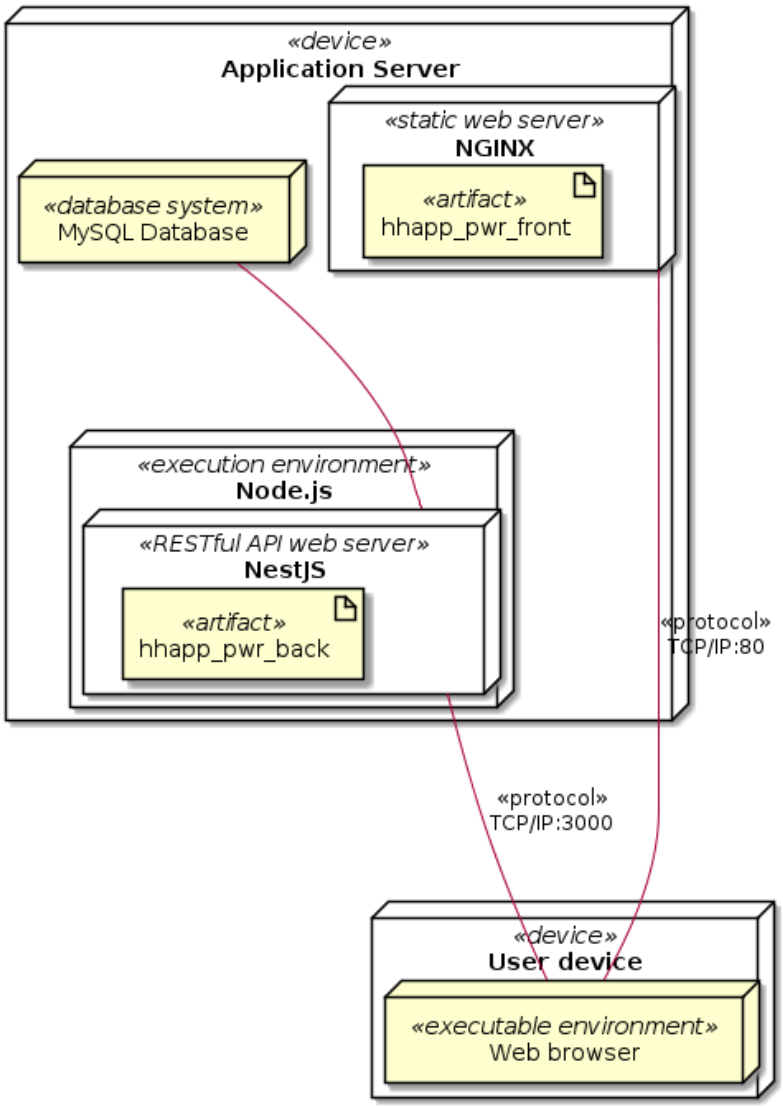
Baza danych aplikacji będzie działać w oparciu o DBMS *MariaDB*[1], który bazuje na i jest kompatybilny z popularnym systemem *MySQL*. Serwerem plików statycznych będzie *NGINX*[2], który jest bardziej przystępny w konfiguracji, od stanowiącego konkurencję serwera *Apache*[3]. Zużywa on mniej zasobów oraz osiąga większą wydajność.

Środowiskiem uruchomieniowym dla aplikacji udostępniającej RESTful API będzie *NodeJS*[4]. Aplikacja ta będzie oparta na frameworku *NestJS*[5], który pozwala na tworzenie wydajnych i ska-

lowalnych aplikacji z użyciem języka *TypeScript*. *NestJS* będzie będzie współpracował z frameworkiem *TypeORM*[6] obsługującym mapowanie obiektowo-relacyjne.

Frontend aplikacji, przesyłany do użytkownika za pomocą serwera plików statycznych, zostanie stworzony z użyciem języka *TypeScript* i frameworka *VueJS*[7]. Za wygląd aplikacji będzie odpowiadać framework *Vuetify*[8], który zaopatruje w gotowy zestaw komponentów, które wyglądem zgodne są ze stylem *Material Design*[9].

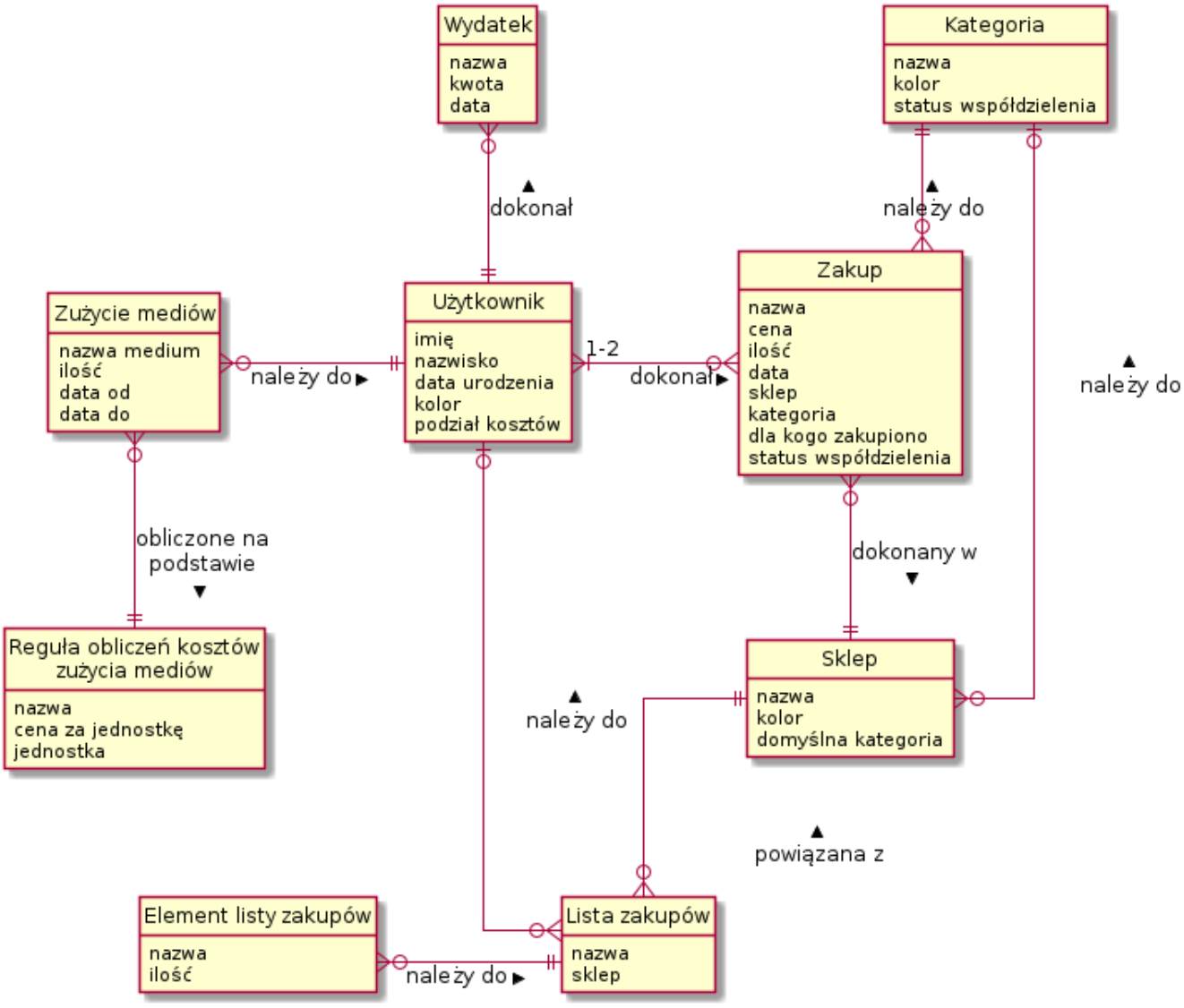
Całość uruchamiana będzie z użyciem wirtualizacji na poziomie systemu operacyjnego, z użyciem środowiska *Docker*[10]. Pozwoli to zminimalizować liczbę kroków potrzebnych do konfiguracji środowiska na różnych maszynach oraz ujednolici je na maszynach deweloperów, co poskutkuje minimalizacją błędów związanych z różnicą konfiguracji w różnych środowiskach.



Rysunek 8: Diagram wdrożenia.

8 Baza danych

8.1 Diagram encji



Rysunek 9: Diagram encji.

8.2 Transakcje i spójność danych

W celu zachowania spójności danych w bazie danych, transakcje są realizowane ograniczeniem pola *"ON DELETE SET NULL"*, *"ON DELETE CASCADE"* lub *"ON DELETE RESTRICT"*.

1. Usunięcie użytkownika powoduje:
    - ustawienie odniesień do niego w encji *Lista zakupów* na wartość *NULL*.
    - Usunięcie zakupów, których dokonał, czyli takich, w których jest wpisany w pole *boughtBy*. Jeżeli usuwany użytkownik jest wpisany w danej krotce w pole *boughtFor*, jest ono ustawiane na wartość *NULL*.
    - Usunięcie jego wydatków.
    - Usunięcie jego wpisów zużycia mediów.
2. Usunięcie listy zakupów powoduje usunięcie przedmiotów znajdujących się w tej liście.
3. Usunięcie sklepu powoduje ustawienie odniesień do niego w encji *Lista zakupów* na wartość *NULL*.
4. Encje *Kategoria* i *Sklep* nie mogą zostać usunięte jeśli powiązane są z encją *Zakup*.
5. Encja *Reguła obliczeń kosztów zużycia mediów* nie może zostać usunięta, jeśli jest powiązana z encją *Zużycie mediów*.
6. Konwersja listy zakupów na zakupy powoduje:
    - Dodanie nowych krotek encji *Zakup*.
    - Usunięcie konwertowanej listy oraz wszystkich jej przedmiotów.

9 Implementacja

9.1 Backend

9.1.1 Framework Nest.js

Backend został zbudowany z wykorzystaniem progresywnego frameworka Nest.js, opartego na języku TypeScript. Łączy on w sobie koncepcje programowania obiektowego, funkcyjnego i funkcyjno-reaktywnego. Do obsługi zapytań wykorzystuje framework Express. Nest.js umożliwia budowanie wysoce skalowalnych aplikacji, co jest dużą zaletą w kontekście realizowanego projektu. W aplikacji Household App został użyty do budowy RESTapi.

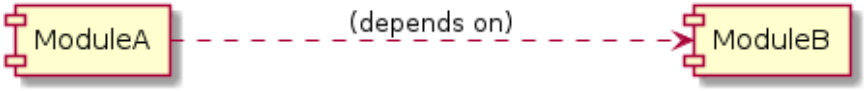
9.1.2 Moduły aplikacji

Aplikacja Nest.js składa się m.in. z modułów, które są wydajnym sposobem podziału aplikacji na niezależne od siebie komponenty. Za pomocą dekoratora `@Module` określa sie z jakich kontrolerów i serwisów korzysta dany moduł. Każdy moduł może importować (wykorzystywać) również inne moduły, które są wymienione w parametrze `imports`.

```
@Module({
  imports: [
    PassportModule.register({ defaultStrategy: 'jwt' }),
    TypeOrmModule.forFeature([Shop]),
    ConfigModule,
    CategoriesModule,
  ],
  controllers: [ShopsController],
  providers: [ShopsService],
  exports: [ShopsService],
})
export class ShopsModule {}
```

Listing 1: Implementacja modułu `ShopsModule`.

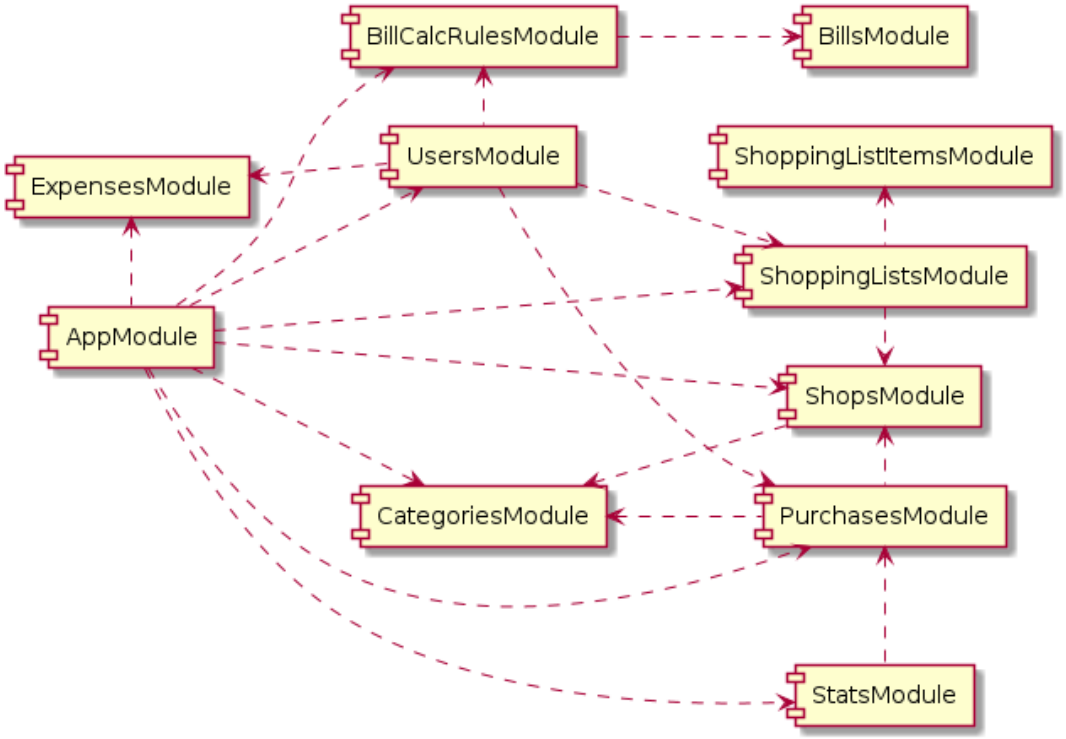
Każde powiązanie (zależność) pomiędzy dwoma modułami reprezentowane jest poprzez połączenie strzałką o przerywanej linii. Na poniższym diagramie, moduł A zależy od modułu B.



Rysunek 10: Legenda do diagramu zależności modułów.

Niektóre moduły są zależnością dla wszystkich innych modułów, które występują na poniższym diagramie. Powiązania z nimi nie zostały pokazane, w celu zachowania czytelności grafu. Są to następujące moduły:

- `PassportModule`
- `ConfigModule`
- `TypeOrmModule`



Rysunek 11: Diagram zależności modułów.

9.1.3 Kontrolery

Kontrolery odpowiadają za obsługę nadchodzących zapytań HTTP i zwracanie odpowiedzi do klienta. Mechanizm routowania, na podstawie ścieżki, decyduje o tym do którego kontrolera zo-

stanie przekazane nadchodzące zapytanie. Każdy kontroler może obsługiwać wiele ścieżek, a każda z nich może realizować inną funkcję aplikacji.

Listing 2 zawiera skrócony kod źródłowy kontrolera dla ścieżek rozpoczynających się od `/bills`. Po przekierowaniu zapytania do odpowiedniej metody kontrolera obsługującej podaną ścieżkę, kontroler wykonuje potrzebne czynności poprzez dostępny mu serwis `billsService`.

```
@Controller('bills')
@ApiUseTags('bills')
@ApiBearerAuth()
@UseGuards(AuthGuard())
export class BillsController {
  constructor(private readonly billsService: BillsService) {}

  @Get()
  async find() { return this.billsService.findAll(); }

  @Get(':id')
  async findOne(@Param('id') id: number) {
    return this.billsService.findById(id);
  }

  @Post()
  async store(@Body() createBillDto: CreateBillDto) {[...]}

  @Put(':id')
  async update(@Param('id') id: number, @Body() updateBillDto: UpdateBillDto)
  {[...]}

  @Delete(':id')
  async delete(@Param('id') id: number) {[...]}
}

```

Listing 2: Implementacja kontrolera dla ścieżki `bills`.

9.1.4 Serwisy

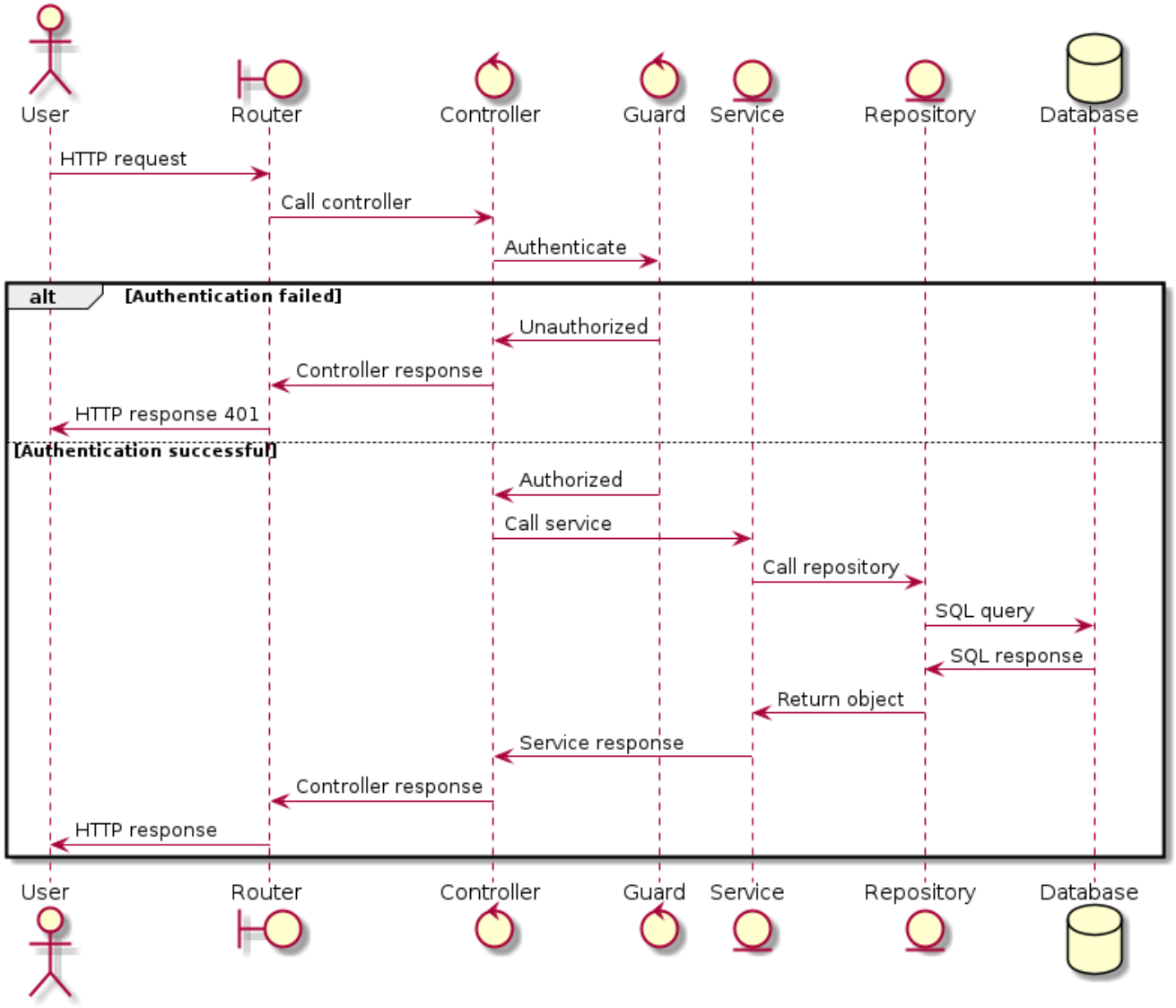
Serwisy enkapsulują i dodają warstwę abstrakcji nad logikę wykorzystywaną w modułach. Głównym celem stosowania serwisów jest wstrzykiwanie zależności do innych klas. Każdy serwis jest klasą odekorowaną dekoratorem `@Injectable()`, implementującą określony zestaw metod.

```
@Injectable()
export class ShopsService {
  constructor(
    @InjectRepository(Shop)
    private readonly shopRepository: Repository<Shop>,
    private readonly categoriesService: CategoriesService,
  ) {}

  async findAll(): Promise<Shop[]> {
    return await this.shopRepository.find({relations: ['defaultCategory']});
  }

  async findById(id: number): Promise<Shop | undefined> {
    const shop = await this.shopRepository.findOne(id);
    if (!shop) {
      throw new NotFoundException();
    }
    return shop;
  }

  async create(dto: CreateShopDto): Promise<Shop> {[...]}

  async update(id: number, dto: UpdateShopDto): Promise<Shop> {[...]}

  async delete(id: number): Promise<Shop> {[...]}
}

```

Listing 3: Implementacja serwisu `ShopsService`.

9.1.5 Obsługa zapytania

Poniższy diagram prezentuje przepływ informacji podczas wywołania zapytania HTTP do RESTapi.



Rysunek 12: Diagram sekwencji przepływu informacji podczas obsługi zapytania.

9.1.6 Struktura RESTapi

RESTapi dla każdej encji pozwaja na operacje pobierania, dodawania, edycji i usuwania. Wykorzystuje do tego metody protokołu HTTP oraz odpowiednio skonstruowane adresy.

- `GET /nazwa-encji`: pobieranie wszystkich encji.
- `GET /nazwa-encji/{id}` : pobieranie encji o podanym id.
- `POST /nazwa-encji`: wprowadzanie encji.
- `PUT /nazwa-encji/{id}`: aktualizowanie encji o podanym id.
- `DELETE /nazwa-encji/{id}`: usuwanie encji o podanym id.
- `GET /users/{id}/payments`: pobieranie należności użytkownika o podanym id.
- `GET /users/{id}/purchases`: pobieranie zakupów użytkownika o podanym id.
- `GET /users/{id}/shopping-lists`: pobieranie list zakupów użytkownika o podanym id.
- `GET /users/{id}/expenses`: pobieranie wydatków użytkownika o podanym id.
- `GET /users/{id}/bills`: pobieranie wydatków użytkownika o podanym id.
- `GET /shopping-lists/{id}/shopping-list-items`: pobranie przedmiotów z listy zakupów o podanym id.
- `GET /shopping-lists/{id}/to-purchases`: zamiana przedmiotów listy zakupów o podanym id, na zakupy.

9.1.7 Walidacja pól zapytania

Pola zapytania HTTP walidowane są poprzez stosowanie klas jawnie opisujących oczekiwanych typów, liczności, jak i konieczności ich podania. Dekorator `ApiModelProperty` dodaje i manipuluje danymi pól modelu RESTapi, tworzonego w oparciu o framework Swagger (9.1.13).

```
export class CreateCategoryDto {
  @ApiModelProperty({ example: 'Obuwie' })
  readonly name: string;

  @ApiModelProperty({ format: 'Hex', example: '#BADA55' })
  readonly color: string;

  @IsBoolean()
  @IsOptional()
  @ApiModelProperty({ example: false, default: false, required: false })
```

```
11   readonly isShared: boolean;
12 }
```

Listing 4: Data transfer object dla operacji tworzenia instancji encji `Category`.

9.1.8 Niestandardowe dekoratory

Nest.js pozwala na tworzenie niestandardowych dekoratorów, które mogą być wykorzystane przy obsłudze zapytań. Listing 5 przedstawia implementację dekoratora `AuthUser`, który umożliwia pobranie użytkownika wykonującego zapytanie.

```
1 export const AuthUser = createParamDecorator((data, req) => {
2   return req.user;
3 });
```

Listing 5: Dekorator `AuthUser`.

9.1.9 Uwierzytelnianie

Podczas pomyślnego logowania, użytkownik otrzymuje token. Wylogowanie użytkownika powoduje umieszczenie jego tokena na czarnej liście, przez co staje się on nieważny (linia 36). Sesja jest aktywna, dopóki token jest ważny, tzn. dopóki istnieje i nie jest umieszczony na czarnej liście.

```
1  @Injectable()
2  export class AuthService {
3    constructor(
4      private readonly jwtService: JwtService,
5      private readonly userService: UsersService,
6      @InjectRepository(JwtBlacklistEntry)
7      private readonly jwtBlacklistEntryRepository: Repository<JwtBlacklistEntry
       >,
8    ) { }
9
10   async validateUser(
11     createSessionDto: CreateSessionDto,
12   ): Promise<User | undefined> {
13     const user = await this.userService.findForAuth(createSessionDto.username);
14     const valid = await bcryptjs.compare(
15       createSessionDto.password,
```

```
      user.password,
    );
    if (valid) {
      return user;
    } else {
      throw new NotFoundException();
    }
  }

  async login(user: User) {
    const payload = {
      username: user.username,
      sub: user.id,
      isAdmin: user.isAdmin,
    };
    return {
      access_token: this.jwtService.sign(payload),
    };
  }

  async logout(user: User, token: string) {
    const entry = this.jwtBlacklistEntryRepository.create({
      user,
      token,
    });
    this.jwtBlacklistEntryRepository.save(entry);
  }
}
```

Listing 6: Serwis `AuthService`.

9.1.10 Autoryzacja

Użytkownik z aktywną sesją może korzystać z dostępnych mu funkcjonalności. Token jest walidowany przed obsługą każdego zapytania za pomocą klasy `JwtStrategy` (listing 7).

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(
```

```
    private readonly configService:ConfigService,
    private readonly userService:UsersService,
    @InjectRepository(JwtBlacklistEntry)
    private readonly jwtBlacklistEntryRepository:Repository<JwtBlacklistEntry>,
  ) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: configService.createJwtOptions().secret,
      passReqToCallback: true,
    });
  }
  async validate(req: Request, payload: any, done: (result: boolean) => void) {
    const token = /.*Bearer *(\S+)$/.exec(req.headers.authorization)[1];
    const blacklisted =
      (await this.jwtBlacklistEntryRepository.count({ token })) !== 0;
    if (blacklisted) {
      done(false);
    } else {
      return {
        id: payload.sub,
        username: payload.username,
        isAdmin: payload.isAdmin,
      }; } } }
```

Listing 7: Klasa `JwtStrategy`.

9.1.11 Mapowanie obiektowo-relacyjne

Mapowanie obiektowo-relacyjne realizowane jest poprzez framework TypeORM. Relacja obiektu z bazą oraz innymi encjami określana jest przez klasę z dekoratorem `@Entity()`. Listing 8 przedstawia implementację encji `Category`. TypeORM umożliwia m.in. zdefiniowanie relacji 1:M, poprzez użycie dekoratora `@OneToMany()`. Ponadto przeprowadza migrację zdefiniowanych encji do bazy danych, co znacząco upraszcza proces tworzenia oprogramowania. Nie jest to jednak zalecane w środowisku produkcyjnym, ze względu na ryzyko przypadkowego naruszenia integralności bazy danych.

```
@Entity()
export class Category {
  @PrimaryGeneratedColumn()
  public id?: number;
  @Column()
  public name?: string;
  @Column({ length: 7 })
  public color?: string;
  @Column({ type: 'bool', default: false, nullable: false })
  public isShared?: boolean;
  @OneToMany(type => Purchase, purchase => purchase.category, {
    onDelete: 'RESTRICT',
  })
  public purchases: Purchase[];
  @OneToMany(type => Shop, shop => shop.defaultCategory, {
    onDelete: 'RESTRICT',
  })
  public shops: Shop[];
}
```

Listing 8: Encja `Category`.

Tak przygotowana encja można zostać zaimportowana jako moduł `TypeOrmModule`.

```
@Module({
  imports: [
    PassportModule.register({ defaultStrategy: 'jwt' }),
    TypeOrmModule.forFeature([Category]),
    ConfigModule,
  ],
  controllers: [CategoriesController],
  providers: [CategoriesService],
  exports: [CategoriesService],
})
export class CategoriesModule {}
```

Listing 9: Moduł `CategoriesModule`.

9.1.12 Wzorzec strażnika jako metoda kontroli dostępu

Nest.js pozwala na użycie wzorca strażnika. Jako że w aplikacji Household App występują dwa rodzaje użytkowników, nie-administrator i administrator, konieczne było ograniczenie dostępu dla tego pierwszego rodzaju. Takie ograniczenie realizuje klasa `AdminGuard` w listingu 10. Wykorzystanie jej w definicji metody kontrolera, powoduje ograniczenie jej dostępności do grupy użytkowników będących administratorami. Przykładowo, metodę `delete` z listingu 11 może wywołać tylko administrator.

```
@Injectable()
export class AdminGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const request = context.switchToHttp().getRequest();
    const user = request.user;
    return user && user.isAdmin;
  }
}
```

Listing 10: Klasa `AdminGuard`.

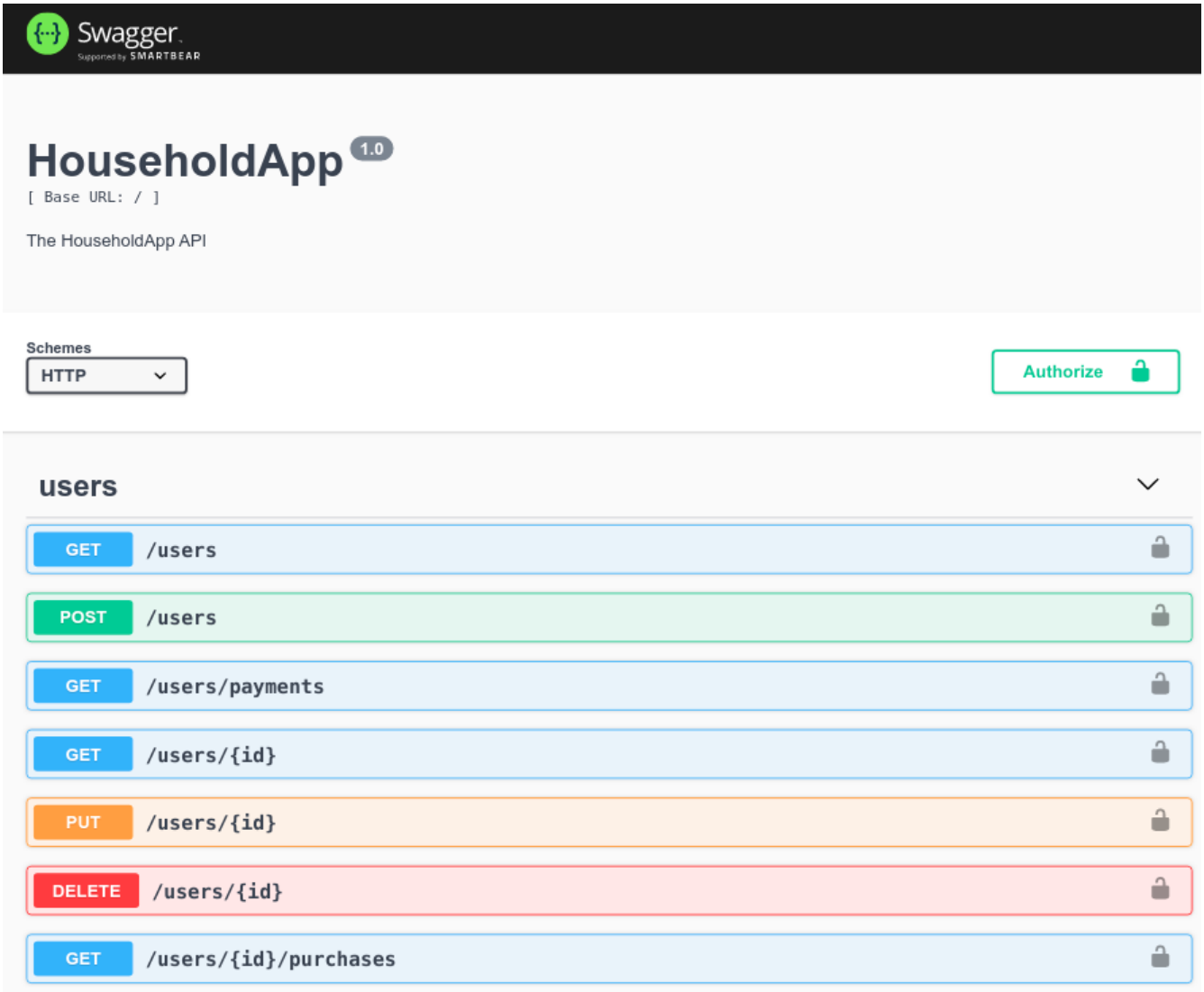
```
@Delete(':id')
@UseGuards(AdminGuard)
async delete(@Param('id') id: number) {
  const user = await this.usersService.delete(id);
  return user;
}
```

Listing 11: Użycie strażnika `AdminGuard`.

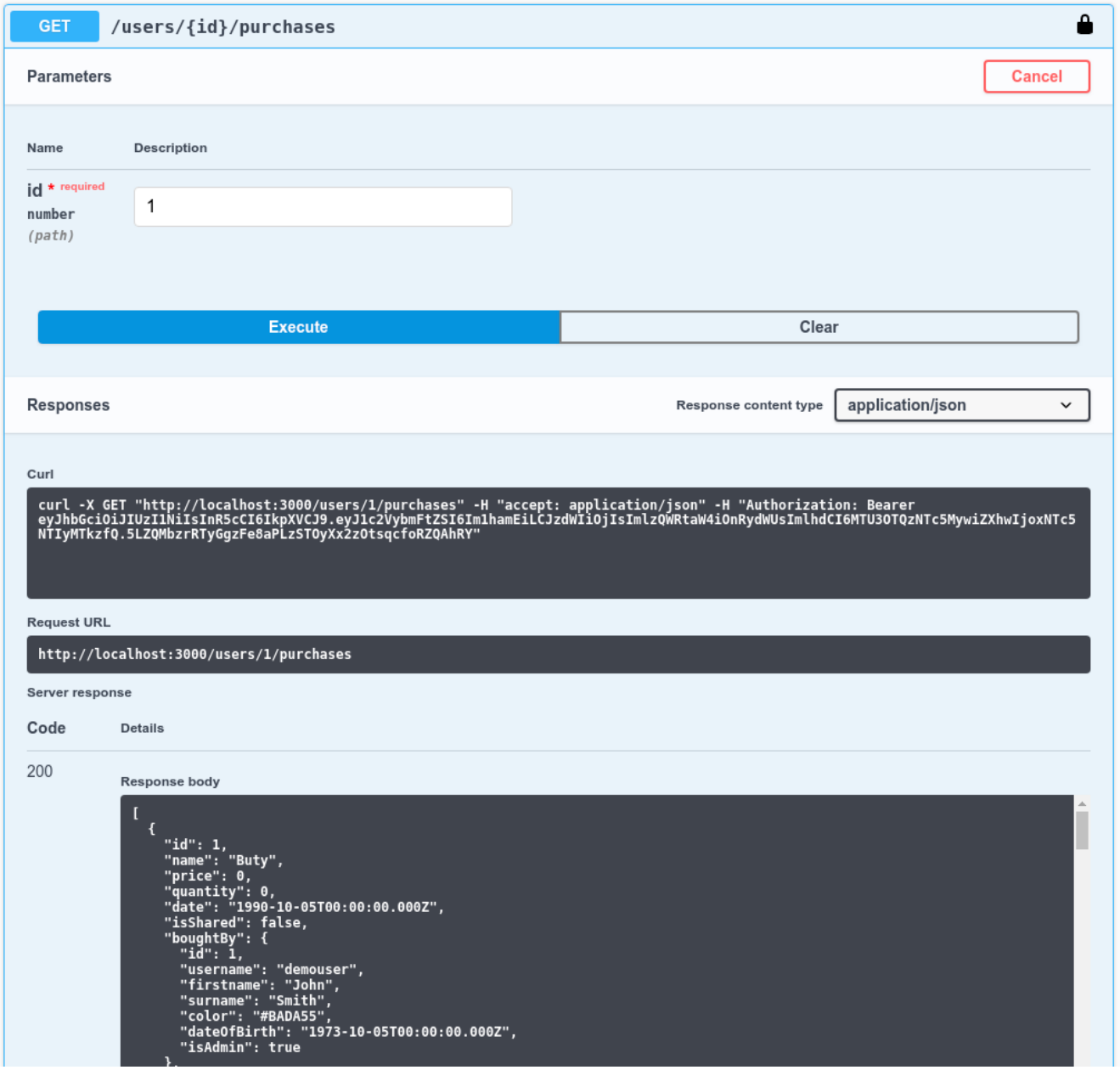
9.1.13 Swagger

Swagger jest frameworkiem wspierającym proces tworzenia i dokumentacji usług webowych RESTapi. Jest on zgodny ze standardem OpenAPI, który określa sposób opisu informacji przekazywanych do/z usługi RESTapi. Pozwala na szybkie testowanie i szczegółowy podgląd działania poszczególnych punktów końcowych. Rysunek 13 przedstawia reprezentację struktury punktów

końcowych kontrolera *users*. Rysunek 14 przedstawia wywołanie punktu końcowego `/users/id/purchases`, dla użytkownika o id=1.



Rysunek 13: Swagger - widok struktury RESTapi.



Rysunek 14: Swagger - wywołanie punktu końcowego.

9.2 Frontend

Interface użytkownika został napisany używając frameworka SPA VueJs. Za wygląd i funkcjonowanie komponentów odpowiedzialny był framework Vuetify, który dostarcza komponenty w stylu Material Design.

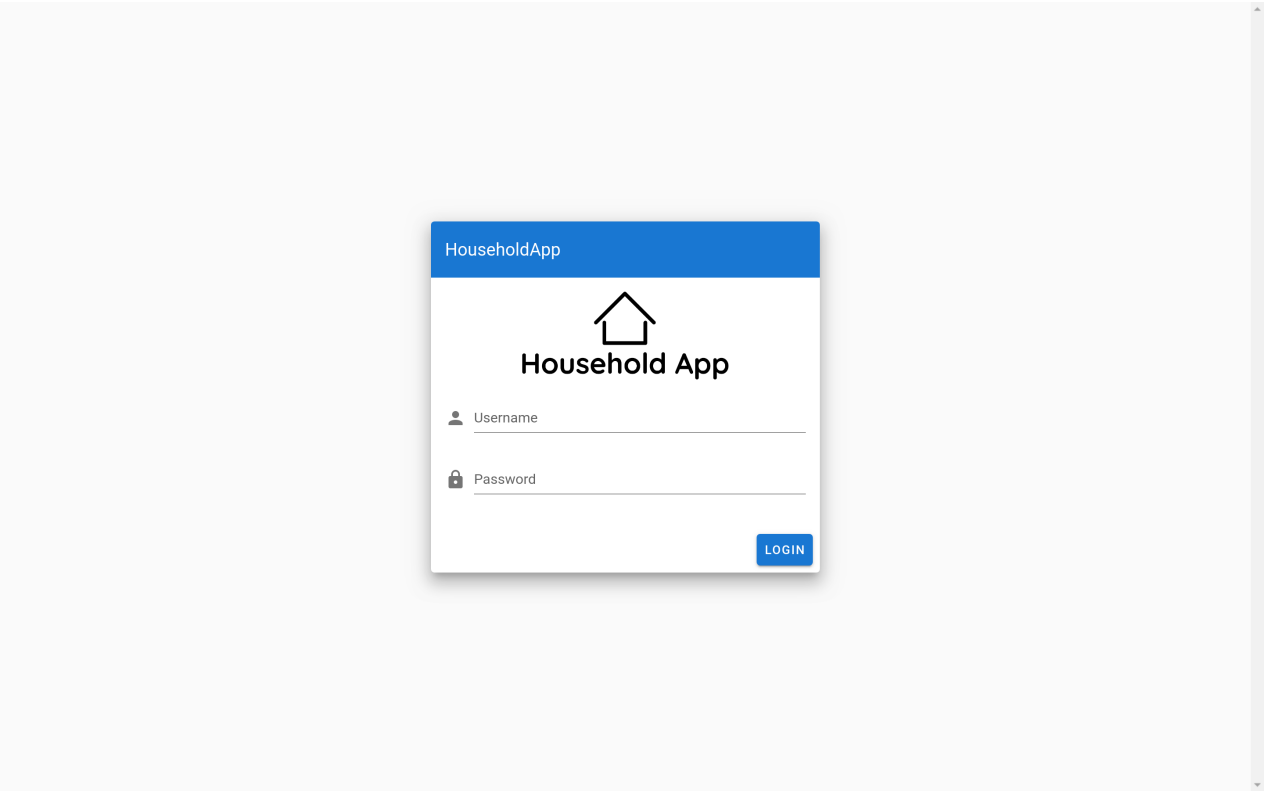
9.2.1 Routing i podstawowe ścieżki

Aplikacja posiada dwa zagnieżdżone routery. Pierwszy steruje głównym widokiem i przełącza pomiędzy widokiem logowania, a widokiem pozostałej części aplikacji. Drugi z nich, zawarty w widoku aplikacji, odpowiada za przełączanie widoku aplikacji pod wpływem wybieranie pozycji z menu. Listing 12 przedstawia kod inicjujący aplikację poprzez podanie obiektów konfiguracyjnych routera, store'a, Vuetify oraz funkcji renderującej, która jako argument przyjmuje główny komponent aplikacji - `App`

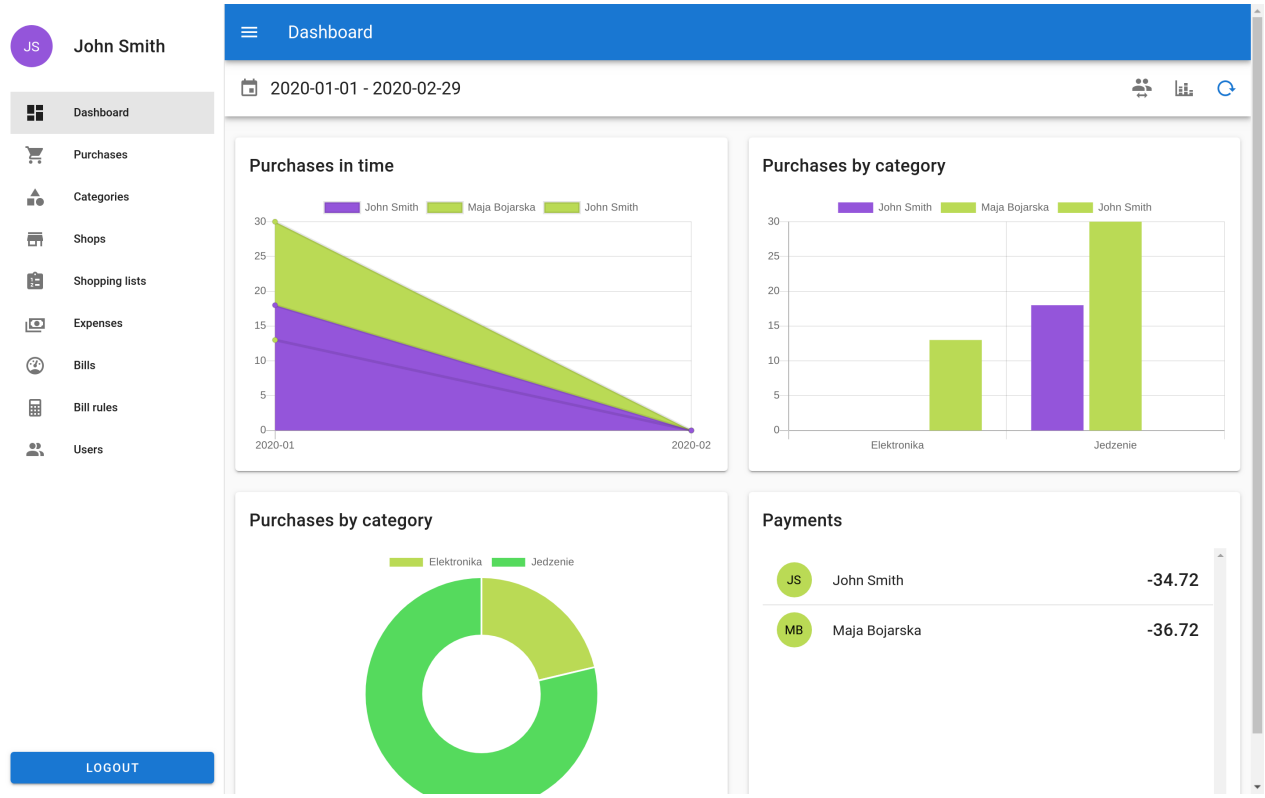
```
1  import App from "./App.vue";
2  import router from "./router";
3  import store from "./store";
4  import vuetify from "./plugins/vuetify";
5  new Vue({
6    router,
7    store,
8    vuetify,
9    render: h => h(App)
10 }).$mount("#app");
```

Listing 12: Punkt wejścia skryptu inicjującego aplikację.

Rysunek 15 i 16 przedstawiają kolejno stronę logowania i pozostałą część aplikacji. Obszarem działania zagnieżdżonego routera jest obszar prawej części ekranu na rysunku 16.



Rysunek 15: Widok strony logowania (`/login`).



Rysunek 16: Widok podsumowania (`/dashboard`).

9.2.2 Komunikacja z serwerem

Do konfiguracji z serwerem wykorzystana została biblioteka `Axios`, która po konfiguracji adresu i portu wysyłała zapytania HTTP. Konfigurację i przykładowe zapytania zamieszczono na listingu 13.

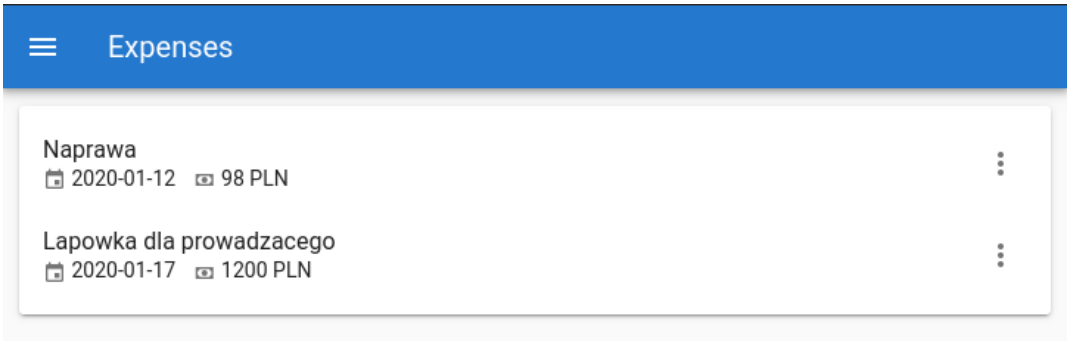
```
1 axios.defaults.baseURL = `http://${process.env.VUE_APP_API_URL}:${process.env.
    VUE_APP_API_PORT}`;
2 ...
3 axios
4   .get("expenses", { headers: { Authorization: this.authHeader } })
5   .then((response: AxiosResponse) => {
6     this.expenses = response.data;
7   });
```

Listing 13: Axios - konfiguracje i przykładowe użycie.

Na listingu 13 widać również przekazywanego z każdym zapytanie wymagającym uwierzytelnienia nagłówka `Authorization` o treści `Bearer <TOKEN>`, gdzie `<TOKEN>` jest tokenem otrzymanym wcześniej w procesie uwierzytelniania.

9.2.3 Wyświetlanie i edycja danych

Dane wyświetlane są w postaci list (rysunek 17). Pozycję z listy można dodać i edytować za pomocą komponentu formularza, który pojawia się na ekranie jako okno dialogowe (rysunek 18).



Rysunek 17: Lista wydatków (`/expenses`).

Rysunek 18: Edycja wydatku (`/expenses`).

Widok listy, z pominięciem danych, definiowany jest w szablonie komponentu pokazanym na listingu 14. Komponenty frameworka Vuetify charakteryzuje prefix `v-`. Na owym listingu widać również osadzony komponent `expense-form`, który wyświetlany jest jako ciało okna dialogowego.

```
<template>
  <v-container>
    <v-card v-if="expenses.length > 0">
      <v-list>
        <v-list-item v-for="expense in expenses" :key="expense.id"
          @click.stop="edit(expense)" >
          <v-list-item-content>
            <v-list-item-title>...</v-list-item-title>
            <v-list-item-subtitle>...</v-list-item-subtitle>
          </v-list-item-content>
          <v-list-item-action @click.stop>...</v-list-item-action>
        </v-list-item>
      </v-list>
      <v-snackbar v-model="snackbarVisible">...</v-snackbar>
    </v-card>
    <v-card v-else>...</v-card>
    <v-dialog
      v-model="dialogVisible"
      max-width="800px"
      :fullscreen="$vuetify.breakpoint.xsOnly"
```

```
    >
      <expense-form
        :expense="dialogExpense"
        @close="dialogVisible = false"
        :type="formType"
        @submit="submit"
      />
    </v-dialog>
    <v-btn @click="create" color="primary" fab fixed right bottom>
      <v-icon>mdi-plus</v-icon>
    </v-btn>
  </v-container>
</template>
```

Listing 14: Struktura widoku listy z pominięciem wyświetlania danych.

9.2.4 Walidacja danych

Za walidację danych odpowiedzialny jest komponent `v-form`, którego obiekt posiada metodę `validate()`. Walidacja wykonuje się również po każdorazowej zmiany wartości pól formularza. Komponent analizuje wartość wszystkich komponentów-dzieci wykonując na nich zdefiniowane przez aplikację reguły. Są one zdefiniowane jako jako funkcje, przyjmujące jako argument wartość pola i zwracające wartość `true` przy poprawnej jego wartości, albo ciąg znaków z komunikatem błędu. Metodę `validate()` odpowiedzialna jest również za wyświetlanie komunikatów użytkownikowi. Listing 15 pokazuje użycie komponentu pola tekstowego, a listing 16 rozwinięcie reguły jego walidacji. Rysunek 19 przedstawia komunikat błędu walidacji.

```
<v-text-field
  v-model="editExpense.name"
  label="Name"
  prepend-icon="mdi-format-text"
  :rules="rules.required"
/>
```

Listing 15: Komponent pola tesktowego

```
readonly rules = {
  required: [(v: string) => !!v || "Field is required!"]
};
```

Listing 16: Obiekt reguł walidacji.

Rysunek 19: Walidacja pola tekstowego danych edycji wydatku (`/expenses`).

9.2.5 MVVM - Model, View, ViewModel

Każdy komponent widoku i formularza w swojej klasie posiada zdefiniowane pola, które zawierają wyświetlane i edytowalne pola obiektów. Przykładem tego jest komponent widoku `Expenses`, który zawiera tablicę obiektów `Expence`. Obiekty te stanowią model widoku (ViewModel). Obiekty klas odpowiedzialne za działanie tych komponentów pośredniczą w wymianie danych pomiędzy widokiem, czyli wyświetlanym formularzem bądź listą, a modelem widoku (View <-> ViewModel). Modelem w tej relacji jest encja zapisana w bazie danych i synchronizowana a modelem widoku w momencie wywołania zdarzeń zapisania i odczytania danych z serwera [11].

Literatura

[1] MariaDB: https://mariadb.org/

[2] NGINX: https://www.nginx.com/

[3] Apache: https://httpd.apache.org/

[4] NodeJS: https://nodejs.org/en/

[5] NestJS: https://nestjs.com/

[6] TypeORM: https://typeorm.io/

[7] VueJS: https://vuejs.org/

[8] Vuetify: https://vuetifyjs.com/en/

[9] Material Design: https://material.io/design/

[10] Docker: https://www.docker.com/

[11] MVVM: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel