Państwowa Wyższa Szkoła Zawodowa w Legnicy
Wydział Nauk Technicznych i Ekonomicznych
Kierunek Informatyka, Rok III, n3PAM1, studia niestacjonarne

Projekt na zaliczenie
ZAAWANSOWANE METODY PROGRAMOWANIA
Car Management

Autorzy:
Sebastian Wiecheć, Karol Rechman, Monika Tuzel, Adrian Mastalerz

Prowadzący:
mgr inż. Marcin Tracz

Spis treści

Opis funkcjonalny systemu i jego części składowych

Temat projektu

Celem projektu naszego zespołu jest stworzenie systemu wspomagającego wynajem wynajem samochodów. Może być wykorzystany w działaniu wypożyczalni pojazdów lub wypożyczaniu samochodów zastępczych w warsztacie samochodowym.

Aplikacja internetowa

Aplikacja internetowa umożliwia korzystanie jako gość, użytkownik oraz administrator. Widok gościa pozwala na przegląd oferty dostępnych pojazdów. Przycisk "Zaloguj się" wywołuje okno logowania lub rejestracji, podobnie jak próba wypożyczenia wybranego pojazdu z widoku gościa. Użytkownik może założyć konto użytkownika, edytować dane użytkownika sprawdzać dostępność pojazdów, przeglądać oraz wypożyczać dostępne pojazdy. Ma również dostęp do historii wypożyczonych pojazdów oraz dotychczasowych wydatków, a także generować zestawienia wydatków dotyczące swojego konta. Administrator może dodawać nowe pojazdy, modyfikować zatwierdzać zwrot pojazdu, wyświetlać koszty, dodawać koszty oraz przebiegi pojazdów a także je kontrolować

Aplikacja desktopowa

Z aplikacji komputerowej może korzystać osoba zalogowana (użytkownik, administrator). Aplikacja umożliwia założenie konta użytkownika osobie niezalogowanej.

Aplikacja mobilna

Aplikacja mobilna skoncentrowana jest głównie na funkcjonalnościach użytkownika, wzbogacona o dostęp do panelu administracyjnego. Tu również nie ma zaimplementowanej funkcjonalności gościa.

API

Stanowi backend całego systemu. Do niego odwołują się zapytania z frontendu wszystkich aplikacji.

Streszczenie opisu technologicznego poszczególnych aplikacji

Aplikacja internetowa

Została stworzona za pomocą frameworku React.js. jest to biblioteka języka programowania JavaScript, która wykorzystywana jest do tworzenia interfejsów graficznych aplikacji internetowych. Najpopularniejsza biblioteka języka Java-Script, z mnóstwem dodatkowych pakietów przyspieszających tworzenie aplikacji frontendowych. Ze względu na ilość pomocnych materiałów jak i dokumentacji wybralśmy właśnie ta technologie.

Aplikacja desktopowa

Powstała na bazie frameworku Electron.js. Electron pozwala na tworzenie aplikacji desktopowych przy pomocy JavaScript, HTML, oraz CSS. Dzięki wbudowanemu silnikowi Chromium i bibliotece Node.js, Electron pozwala stworzyć wieloplatformowe aplikacje działające zarówno na systemach operacyjnych z rodziny Windows, MacOs, jak i Linux. Stanowi narzędzie znacząco ułatwiające tworzenie pierwszych rozbudowanych aplikacji. Przy pomocy Electrona powstały interfejsy graficzne kilku znaczących projektów jak Slack, czy Discord, a także edytor kodu źródłowego Visual Studio Code.

Aplikacja mobilna

Stworzono na bazie biblioteki React Native. React Native to framework (platforma programistyczna, czyli szkielet do budowy aplikacji) stworzony przez Facebook, aby przyspieszyć proces tworzenia aplikacji mobilnych. Pozwala na symultaniczne budowanie aplikacji zarówno na platformę Android, jak i iOS, wykorzystując język JavaScript. Wystarczy napisać kod w języku JavaScript, a w efekcie otrzymamy aplikację mobilną z natywnymi elementami dla iOS oraz Android. Dzieje się tak, ponieważ React Native używa mostów (bridges) do przekładania języka JavaScript na natywne komponenty.

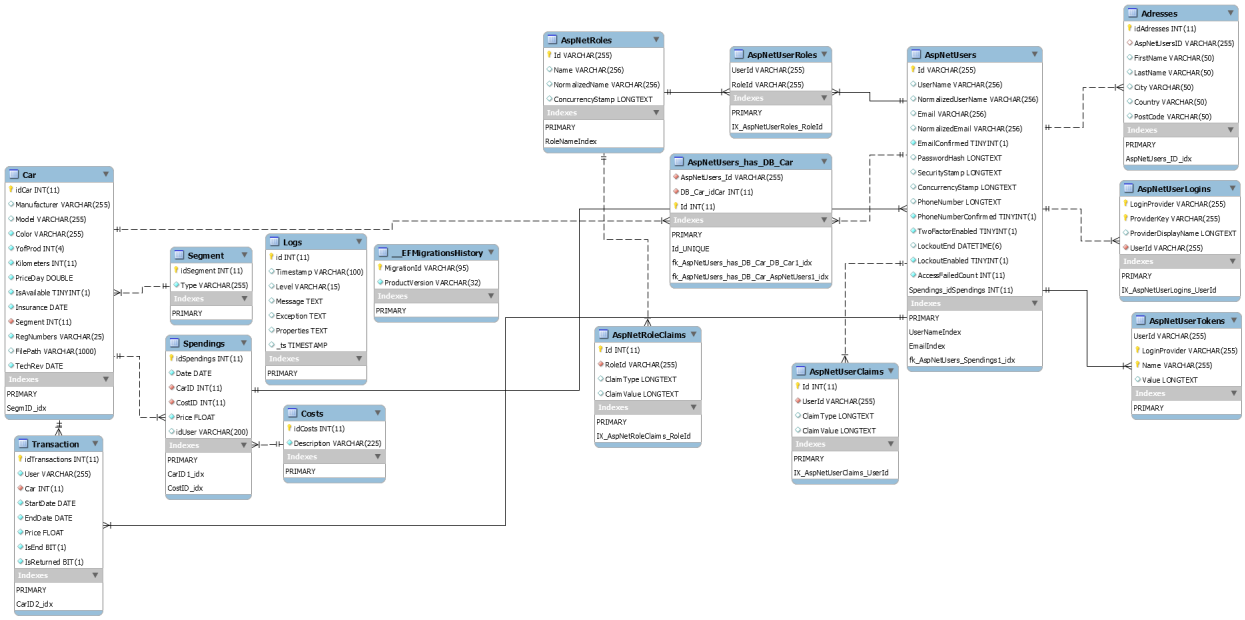
API

API utworzono w ASP.NET Core 3.1 powszechnie znany otwartoźródłowy Framework, wspierany przez firmę Microsoft. Wybraliśmy ten framework, ponieważ jest on najnowocześniejszym i najbardziej rozwiniętym narzędziem do programowania w języku C# dla aplikacji webowych. Posiada obszerna dokumentacje i wsparcie nie tylko od Microsoftu, ale od innych użytkowników pracujących przy nim i programujących za jego pomocą.

Na platformie jest dostępna niezliczona ilość bibliotek, które pomagają w programowaniu aplikacji. Język C# jest nowoczesnym i przyjaznym językiem obiektowym z wieloma udogodnieniami dla programisty.

Baza danych

Baza danych została stworzona i jest zarządzana przy pomocy oprogramowania MySql. Jest to ogólnodostępny, otwartoźródłowy system zarządzania relacyjnymi bazami danych. MySQL rozwijany jest przez firmę Oracle. Biorąc po uwagę fakt, że język SQL jest nauczany na studiach,wybraliśmy właśnie tą bazę danych. Lekki i prosty w obsłudze system MySql'a doskonale sprawdza się w naszej aplikacji. Dodatkowo dostępność serwera za darmo w chmurze sprawiła, iż jest on najlepszym silnikiem baz danych, który mogliśmy wybrać.



Rysunek 2.1: Schemat relacji

Instrukcja lokalnego i zdalnego uruchomienia testów oraz samego systemu

Testy

Do testów aplikacji internetowej, desktopowej oraz mobilnej użyto bibliotek do testowania języka Javascript, między innymi JEST, React testing library oraz MSW. Kilka słów o zastosowanych bibliotekach:

- **JEST** - framework do testowania JavaScript obsługiwany przez Facebook, Inc. zaprojektowany i zbudowany przez Christopha Nakazawę z naciskiem na prostotę i obsługę dużych aplikacji internetowych. Działa z projektami używającymi Babel, TypeScript, Node.js, React, Angular, Vue.js i Svelte.
- **React testing library** - biblioteka została ona stworzona w celu propagowania idei pisania testów, które używają komponentów podobnie jak potencjalny użytkownik aplikacji.
- **Mock Service Worker library** - narzędzie pozwalające na stworzenie testów z wirtualnymi usługami widzianymi jako rzeczywiste. Zawiera definicje dla operacji takich wezwania klienta, prośby oraz wysyłanie symulowanych odpowiedzi.

Uruchamianie testów odbywa się poprzez terminal już zainstalowanej aplikacji. Komendy uruchamiająca test to `npm run test` lub `npm yarn test`. Rezultat zadań zdefiniowanych w testach jest widoczny w postaci wyniku w terminalu.

Uruchamianie aplikacji

Aplikacja internetowa

Uruchamianie lokalne

Do stworzenia interfejsu użytkownika został wykorzystany framework React App (repozytorium dostępne pod adresem `https://github.com/facebook/create-react-app`) i zostaną podane kolejne kroki do stworzenia aplikacji React. Aby rozpocząć tworzenie Na sam początek w terminalu, w folderze projektu zaczynamy od wpisania komend:

```
npm install
```

oraz

```
npm install @material-ui/core @material-ui/data-grid @material-ui/icons axios markdown-to-
    jsx react-material-ui-carousel react-redux react-router-dom redux redux-thunk

```

Pozostając w terminalu, wpisujemy:

```
npm start
```

Ta komenda uruchamia aplikację w trybie developerskim. Aby zobaczyć aplikacje w przeglądarce, należy wpisać adres `http://localhost:3000`. Strona przeładuje się, jeśli dokonasz zmian. Widoczne też będą błędy w pisowni i składni.

Następna komenda uruchamia test aplikacji w trybie interaktywnym.

```
npm test
```

Więcej informacji o testach dostępne w dokumentacji (`https://facebook.github.io/create-react-app/docs/running-tests`).

Poniższa komenda pozwoli "zbudować" i przygotować do uruchomienia naszą aplikację React. Proces zapoczątkowany przez komendę optymalizuje cały pakiet pod kątem jak najlepszego wykorzystania zasobów systemowych i płynności działania.

```
npm run build
```

Aplikacja jest gotowa do uruchomienia. Więcej informacji o uruchamianiu aplikacji React w dokumentacji (`https://facebook.github.io/create-react-app/docs/deployment`).

Jeśli nie chcemy zachowywać zbudowanej wersji projektu lub zmienić konfigurację, możemy skorzystać z polecenia `eject`. Usunie ona tą konkretną wersję projektu i pozwoli na modyfikację plików źródłowych.

```
npm run eject
```

Operacja zapoczątkowana przez komendę `eject` jest nieodwracalna. Samo jej użycie jest całkowicie opcjonalne, ale pozwala lepiej zapanować nad wersjami aplikacji.

Serwer jest uruchamiany po uruchomieniu aplikacji zintegrowanego środowiska programistycznego (IDE) lub edytora:

- Visual Studio: profile uruchamiania mogą służyć do uruchamiania aplikacji i serwera za pomocą IIS Express / ASP.NET Core Module lub konsoli programu.
- Visual Studio Code: aplikacja i serwer są uruchamiane przez omnisharp, która aktywuje debuger CoreCLR.
- Visual Studio dla komputerów Mac: aplikacja i serwer są uruchamiane za pomocą debugera Soft-Mode mono.

Podczas uruchamiania aplikacji z poziomu wiersza polecenia w folderze projektu, `dotnet run` uruchamia aplikację i serwer (tylko Kestrel i HTTP.sys). Konfiguracja jest określana przez `-c|-configuration` opcję, która jest ustawiona na wartość `Debug` (domyślnie) lub `Release`. *launchSettings.jsw* pliku zapewnia konfigurację podczas uruchamiania aplikacji przy użyciu `dotnet run` debugera wbudowanego w narzędzia, takiego jak Visual Studio. Jeśli profile uruchamiania są obecne w `launchSettings.jsw` pliku, użyj `-launch-profile PROFILE NAME` opcji z `dotnet run` poleceniem lub wybierz profil w programie Visual Studio.

Uruchamianie wdrożonej aplikacji

W celu uruchomienia aplikacji w pierwszej kolejności należy uruchomić przeglądarkę internetową oraz wpisać adres `https://sebastianwiechec.github.io/CarsharingFrontend/`. Dostępna jest również aplikacja administracyjna `https://karolrechman.github.io/AdminCarsharing/`.

Aplikacja desktopowa

Uruchamianie aplikacji lokalnie

Do stworzenia interfejsu użytkownika również został wykorzystany framework React App (repozytorium dostępne pod adresem `https://github.com/facebook/create-react-app`) oraz Electron, służący do budowania cross-platformowych aplikacji desktopowych wykorzystujących JavaScript, HTML, CSS oraz darmowy template "Material Dashboard React" (repozytorium dostępne pod adresem `https://github.com/creativetimofficial/material-dashboard-react.git`) Poniżej zostaną podane kolejne kroki do stworzenia aplikacji w trybie developerskim. Aby rozpocząć tworzenie a początek w terminalu, w folderze projektu zaczynamy od wpisania komendy :

```
npm install

```

Zostaną zainstalowane niezbędne składniki dla np. Visual Studio Code. Pozostając w terminalu, wpisujemy:

```
  npm run dev

```

Ta komenda uruchamia aplikacje w trybie developerskim. Otwiera się okno aplikacji Windows. Następna komenda uruchamia test aplikacji w trybie testowym

```
  npm run test-jest

```

Wiecej informacji o testach dostepne w dokumentacji (`https://facebook.github.io/create-react-app/docs/running-tests`). Poniższa komenda pozwoli "zbudować" i przygotować do uruchomienia naszą aplikację. Proces zapoczątkowany przez komendę optymalizuje cały pakiet pod katem jak najlepszego wykorzystania

```
npm run build

```

Jeśli nie chcemy zachowywać zbudowanej wersji projektu lub zmienić konfiguracje, możemy skorzystać z polecenia `eject`. Usunie ona ta konkretna wersje projektu i pozwoli na modyfikacje plików źródłowych.

```
npm run eject

```

Operacja zapoczątkowana przez komendę `eject` jest nieodwracalna. Samo jej użycie jest całkowicie opcjonalne, ale pozwala lepiej zapanować nad wersjami aplikacji.

Aplikacja w trybie developerskim jest uruchamiana w aplikacji zintegrowanego środowiska programistycznego (IDE) lub edytora:

- Visual Studio: profile uruchamiania mogą służyć do uruchamiania aplikacji i serwera za pomocą IIS Express / ASP.NET Core Module lub konsoli programu.
- Visual Studio Code: aplikacja jest uruchamiana przez omnisharp, który aktywuje debuger CoreCLR.
- Visual Studio dla komputerów Mac: aplikacja jest uruchamiana za pomocą debugera Soft-Mode mono.

Aplikacja mobilna

Uruchamianie aplikacji lokalnie

Aplikację w trybie zdalnym można uruchomić np. w terminalu programu Visual Studio Code po zainstalowaniu niezbędnych komponentów. Należy się upewnić, czy na maszynie jest zainstalowany (jeśli nie, to zainstalować) Node.js, React Native CLI oraz EXPO CLI, a także środowisko programistyczne Java. W przypadku JRE należy skonfigurować zmienne środowiskowe. Program uruchamiamy w terminalu komendą:

```
expo start
```

W przeglądarce zostaje uruchomiony serwer Metro, w osobnej konsoli uruchamia się Node.js. Na stronie serwera Metro w przeglądarce wybiera się sposób demonstracji aplikacji. Można skorzystać z emulatora urządzenia z systemem Android pod warunkiem posiadania Android Studio oraz aktualnego SDK. Alternatywę stanowi wygenerowanie kodu QR umożliwiającego ukazanie aplikacji w formie Live APK na urządzeniu z systemem Android, na przykład na prywatnym smartfonie z zainstalowaną aplikacją EXPO. Pozostaje możliwość przewodowego połączenia z urządzeniem w trybie debugowania. Następuje tymczasowa "instalacja" umożliwiająca obsługę oraz debugowanie aplikacji.

Uruchamianie aplikacji na urządzeniu

Na urządzeniu z systemem Android należy pobrać plik z rozszerzeniem .apk. Jeżeli nie zrobiono tego do wcześniej, należy zezwolić na instalację aplikacji spoza Sklepu Play. Po zainstalowaniu, wybranie aplikacji w szufladzie lub na ekranie głównym (w zależności od zainstalowanego launchera) uruchomi ją.

API

Instalacja API odbywa się poprzez uruchomienie edytora lub środowiska programistycznego, które współpracuje z platformą Docker. Należy również uruchomić samą platformę na maszynie i wstępnie skonfigurować parametry startowe poprzez sprawdzenie, jakiego systemu obraz uruchomi platforma Docker. Następnie uruchamia się repozytoria API. Docker samoczynnie dokona konteneryzacji API, dzięki czemu staną się dostępne dla frontendu. Po wypuszczeniu API na chmurę Azure w klastrze usługi Kubernetes (taki rodzaj implementacji przyjęto w projekcie) nastąpi powiązanie z pozostałymi aplikacjami.

Streszczenie wykorzystanych wzorców projektowych w każdej aplikacji

Aplikacja internetowa, desktopowa oraz mobilna

Ze względu na decyzje o sposobie tworzenia aplikacji, we wszystkich aplikacjach można znaleźć analogiczne wzorce projektowe.

Zastosowane wzorce:

- **Obserwator** (ang. *Observer*), znany też jako **Event-Subscriber** lub **Listener**, to czynnościowy (behawioralny) wzorzec projektowy pozwalający zdefiniować mechanizm subskrypcji w celu powiadamiania wielu obiektów o zdarzeniach dziejących się w obserwowanym obiekcie. Przykład w kodzie:

```
//Obserwator

<CustomInput
labelText="Manufacturer"
id="manufacturer"
formControlProps={{
    fullWidth: true,
  }}
  required
  labelProps={{
    shrink: car.manufacturer ? true : false,
  }}
  inputProps={{
    onChange: handleChange, // zaimplementowany wzorzec obserwatora, ktory przy kazdej zmianie pola danych uruchamia funkcje handleChange
    value: car.manufacturer,
  }}
/>;

const handleChange = (event) => {
  const name = event.target.id;
  setCar({
    ...car,
    [name]: event.target.value,
  });
};
```

- **Singleton** jest kreacyjnym wzorcem projektowym, który pozwala zapewnić istnienie wyłącznie jednej instancji danej klasy. Ponadto daje globalny punkt dostępowy do tejże instancji. Przykład w kodzie:

```
//Singleton
const [car, setCar] = useState({ idCar: 0 }); // we farmework React useState jest hookiem, ktory pozwala korzystac ze stanu w komponencie funkcyjnym. Stan zapisywany jest w jednym obiekcie - singletonie.
```

- **MVVM, Model-View-View-Model** opiera się na wydzieleniu odpowiednich warstw w systemie, w celu podziału zadań oraz zmniejszenia zależności pomiędzy klasami. Mamy więc klasy modelu danych, których zadaniem jest przechowywanie danych oraz ich ewentualną walidację. Klasy tej warstwy powinny być jak najprostsze pod względem budowy. Pod żadnym pozorem nie mogą odwoływać się do warstwy modelu widoku a tym bardziej do samego widoku.

- **Mediator**, znany też jako **Intermediary** lub **Controller**. Behawioralny wzorzec projektowy pozwalający zredukować chaos zależności pomiędzy obiektami. Wzorzec ten ogranicza bezpośrednią komunikację pomiędzy obiektami i zmusza je do współpracy wyłącznie za pośrednictwem obiektu mediatora Przykład w kodzie"

```
//Mediator

if (car.idCar != 0) {
    await api.request(API_TYPES.CAR).update(car.id, car);
  } else {
    await api.request(API_TYPES.CAR).create("/", car);
```

```
  }             // wycinek z metody, ktora sprawdza, czy samochod zostal zaktualizowany

  <CustomInput
  labelText="Manufacturer"
  id="manufacturer"
  formControlProps={{
    fullWidth: true,
  }}
  required
  labelProps={{
    shrink: car.manufacturer ? true : false,
  }}
  inputProps={{
    onChange: handleChange,
    value: car.manufacturer,
  }}
/>;             // pole wyboru pojazdu - producenta


  <Button color="primary" onClick={SendData}>
  Update Info
</Button>       // przycisk update

<Modal
open={open}
onChange={handleClose}
txt={"OK"}
title={"Auto wynaj Źte"}
/>              // modal wyswietlajacy informacje o wynajeciu pojazdu
```

- **Conditional Rendering** - jedno z głównych narzędzi w arsenale każdego programisty. Przy pisaniu komponentów, często powstaje potrzeba renderowania konkretnego kodu JSX bazującego na stanie. Jest to do osiągnięcia dzięki warunkowemu renderowaniu.Jest to bardzo pożyteczne narzędzie pozwalające na tworzenie odrębnych komponentów według potrzeb, renderując w danej chwili jedynie te, których wymaga stan aplikacji.
- **Render Props** według oficjalnej dokumentacji React, Render Props definiuje się jako udostępnianie kodu między komponentami używając właściwości, której wartość jest funkcją.
- **Container-View** - najbardziej wydajny i szeroko używany wzorzec strukturalny w środowisku programowania React.
- **Container Component** to punkt wejściowy funkcjonalności/ekranu. Do obowiązków kontenera komponentów należą:
  - pobieranie danych;
  - integracja z redux;
  - obsługa efektów ubocznych , cięższe obliczenia i mapowanie danych;
  - finalne przekazywanie wymaganych właściwości do widoku.
- **View Component** powinien zawierać jedynie część prezentacyjną, czyli logikę interfejsu użytkownika,

API

Wzorce projektowe zastosowane w API:

- **Architektura mikroserwisów**. Mikroserwisy(mikrousługi) to styl architektoniczny, w którym pojedyncza aplikacja jest tworzona jako zestaw małych usług. Każda usługa działa w ramach własnego procesu i jest – albo przynajmniej powinna być – niezależna od innych. Warto podkreślić, że mikroserwis to nie jest zawsze pojedyczny projekt (moduł). W ramach jednego mikroserwisu możemy mieć kilka projektów (modułów), które tworzą jeden niezależny mikroserwis. Jest implementacją wzorca **SOA**(ang. *Service Oriented Architecture*). Termin "Service Oriented Architecture" jest określeniem architektury systemów rozproszonych, w których wyróżnia się:
  - usługobiorcę – klienta korzystającego z usług,
  - dostawcę usług – usługą jest realizacja pewnego przetwarzania z wykorzystaniem dostarczonych danych,
  - rejestr usług – miejsce, gdzie klient uzyskuje informacje o potrzebnych mu usługach.

  Podstawą SOA jest wykorzystanie przesyłania komunikatów do wymiany informacji między uczestnikami przetwarzania. Architektury usługowe odróżnia się od architektur obiektowych i architektur komponentowych.

- **MVC**(ang. *Model-View Controller*) jest jednym z najczęściej stosowanych wzorców projektowych w informatyce. Wiele prac traktuje go jako pojedynczy wzorzec, lecz może on być także traktowany jako złożony wzorzec wykorzystujący idee wzorców prostych, takich jak Obserwator, Strategia czy Kompozyt Głównym założeniem tego wzorca jest podzielenie kodu aplikacji na 3 moduły:
  - Model reprezentujący dane (np. pobierane z bazy danych czy parsowane z plików XML),
  - Widok reprezentujący interfejs użytkownika,
  - Kontroler czyli logikę sterującą aplikacją.

  MVC najczęściej stosowane jest przy tworzeniu dynamicznie generowanych aplikacji internetowych.

- **Wzorzec fabryki**, (ang. *Factory*) Wzorzec Factory pozwala oddelegować tworzenie obiektu do innych klas. Jest to przydatne w momencie gdy mamy do stworzenia obiekt, który jest powiązany z wieloma innymi obiektami.Wyróżnia się poszczególne implementacje wzorca Fabryki: Prosta Fabryka (ang. *Simple Factory*), Metoda Fabryki (ang. *Factory Method*), oraz Fabryka Abstrakcyjna (ang. *Abstract Factory*).

- **Dekorator** znany też jako Nakładka lub Wrapper, to strukturalny wzorzec projektowy pozwalający dodawać nowe obowiązki obiektom poprzez umieszczanie tych obiektów w specjalnych obiektach opakowujących, które zawierają odpowiednie zachowania. Przykład w kodzie:

```
/*
* dekorator
*/
public interface IEmailService
{
    void Send(Email email);
    Task<string> CreateHTMLTableAsync(List<Spendings> spendings);
}

public class EmailServiceDecorator : IEmailService
```

```
{
    protected readonly IEmailService emailService;
    protected ISpendingsService spendingsService;

    public EmailServiceDecorator(IEmailService emailService, ISpendingsService service)
    {
        this.emailService = emailService;
        spendingsService = service;
    }

    public async Task<string> CreateHTMLTableAsync(List<Spendings> spendings)
    {
        string stringHTML = $@"<!DOCTYPE html>...";
        return await Task.FromResult(stringHTML);
    }
```

- **DTO, Data Transfer Object** jest wzorcem projektowym należącym do grupy wzorców dystrybucji. Podstawowym zadaniem DTO jest transfer danych pomiędzy systemami, aplikacjami lub też w ramach aplikacji pomiędzy warstwami aplikacji, modułami lub też w każdej dowolnej sytuacji, w której transfer danych jest konieczny. Przykład w kodzie:

```
/*
* Tworzenie obiektow transferu danych (DTO)
*/
public class Spendings
    {
        [Key]
        public int idSpendings { get; set; }  //numer id wydatk w
        public DateTime Date { get; set; }    //Data
        public int CarID { get; set; }        //numer id auta
        public int CostID { get; set; }       //numer id koszt w
        public double Price { get; set; }     //Cena
        public string idUser { get; set; }     //Id u ytkownika
    }

Task<IList<Spendings>> GetSpendings();
Task<List<Spendings>> GetSpendingsByIdAsync(string id);
Task<string> AddSpendings(Spendings spendings);
Spendings UpdateSpendings(Spendings spendings);
Task<string> DeleteSpendings(int id);
```

- **Wstrzykiwanie zależności** polega na przekazywaniu gotowych, utworzonych instancji obiektów udostępniających swoje metody i właściwości obiektom, które z nich korzystają (np. jako parametry konstruktora). Stanowi alternatywę do podejścia, gdzie obiekty tworzą instancję obiektów, z których korzystają np. we własnym konstruktorze. Jest jednym ze sposobów realizacji paradygmatu IoC(ang. *Inversion of Control*- odwrócenie sterowania) (mogą być to także eventy, czy programowanie aspektowe). W podejściu tym, obiekt nie tworzy obiektów, które wykorzystywane są wewnątrz. Dzięki temu, nie wiążemy się z konkretną implementacją (najlepiej operować na interfejsach), a także nie musimy znać parametrów konstruowanego obiektu. Operując na interfejsach stajemy się niezależni od konkretnej implementacji, a nasz kod zaczyna realizować Open/Closed principle.

  Kod:

```
/*
* wstrzykiwanie zaleznosci
*/
private ISpendingsService spendingsService { get; }
```

```
5  private IEmailService emailService { get; }
6
7  public SpendingsController(ISpendingsService _spendingsService, IEmailService email)
8  {
9      spendingsService = _spendingsService;
10     emailService = email;
11 }
12
```

- **Fluent API**

  Przykład w kodzie:

```
1  * fluent API
2  */
3  public static class MyServices
4      {
5          public static IServiceCollection RegisterMyServices(this IServiceCollection
       services) => services
6                      .AddTransient<ISpendingsService, SpendingsService>()
7                      .AddTransient<IEmailService, EmailService>();
8      }
9
10 /*
11 * fluent API
12 */
13 var spendings = await dbContext.Spendings.Where(s => s.idUser == id).ToListAsync();
```

Wnioski projektowe

Nasz projekt rozpoczął się od utworzenia bazy danych, którą oparliśmy o MySQL, dzięki czemu mogliśmy umieścić naszą gotową już bazę na darmowym hoście, aby każda z osób miała łatwy dostęp do tej samej bazy. Po utworzeniu modelu zaczęliśmy od stworzenia podstawowych metod do obsługi bazy (tzw. CRUD).

Ustaliliśmy podział ról, czyli kto będzie pisał którą aplikację. Wszyscy pracowaliśmy wspólnie. Polegało to na tym, aby się spotykać na czacie głosowym. W momencie, gdy ktoś miał problem, z którym nie mógł sobie poradzić, omawialiśmy go na bieżąco i pozostali podsuwali rozwiązania. To sprawiło, że każdy z nas mógł pracować w takim samym tempie jak reszta, a jednocześnie nikt nie zostawał w tyle.

Po zaprogramowaniu metod obsługujących naszą bazę danych, zaczęliśmy od stworzenia jednego repozytorium do obsługi frontu. Wybraliśmy React jako główny framework. Aplikacja desktopowa powstała przy zastosowaniu Electrona, natomiast wersję mobilną stworzono z użyciem React Native. Powyższe założenia sformułowano aby móc się podzielić większą częścią logiki programowania aplikacji. Dzięki odpowiednio zdawkowanej wiedzy, nauczyliśmy się współpracy, zaangażowania oraz rozwiązywania początkujących problemów.

Całość naszej aplikacji od strony backendu umieściliśmy w dockerze, w architekturze mikroserwisów, za pomocą platformy Docker Compose, który docelowo miał być dostępny na chmurze Azure w klastrze usługi Kubernetes. To pomogło nam rozwiązać problem pracy wewnątrz kontenera.

Wspólna praca nad aplikacjami pozwoliła nam zdobyć wiedzę, jak tworzyć kompatybilne elementy systemu wieloplatformowego.

Nie zabrakło czasu na opracowanie i wdrożenie testów, a dzięki wcześniejszym decyzjom w kwestii spójności logiki programowania, nie wystąpiła konieczność powielania testów we wszystkich aplikacjach.

To wszystko doprowadziło nas do tego miejsca, gdzie każdy zaangażował się w 100%, gdzie wszyscy rozumieli co robili i gdzie każdy wyniósł z tego projektu ważną lekcję na temat zarządzania, współpracy i motywacji. Praca przy pierwszym tak rozległym przedsięwzięciu uświadomiła nam, jak może wyglądać organizacja pracy w prawdziwym projekcie programistycznym.