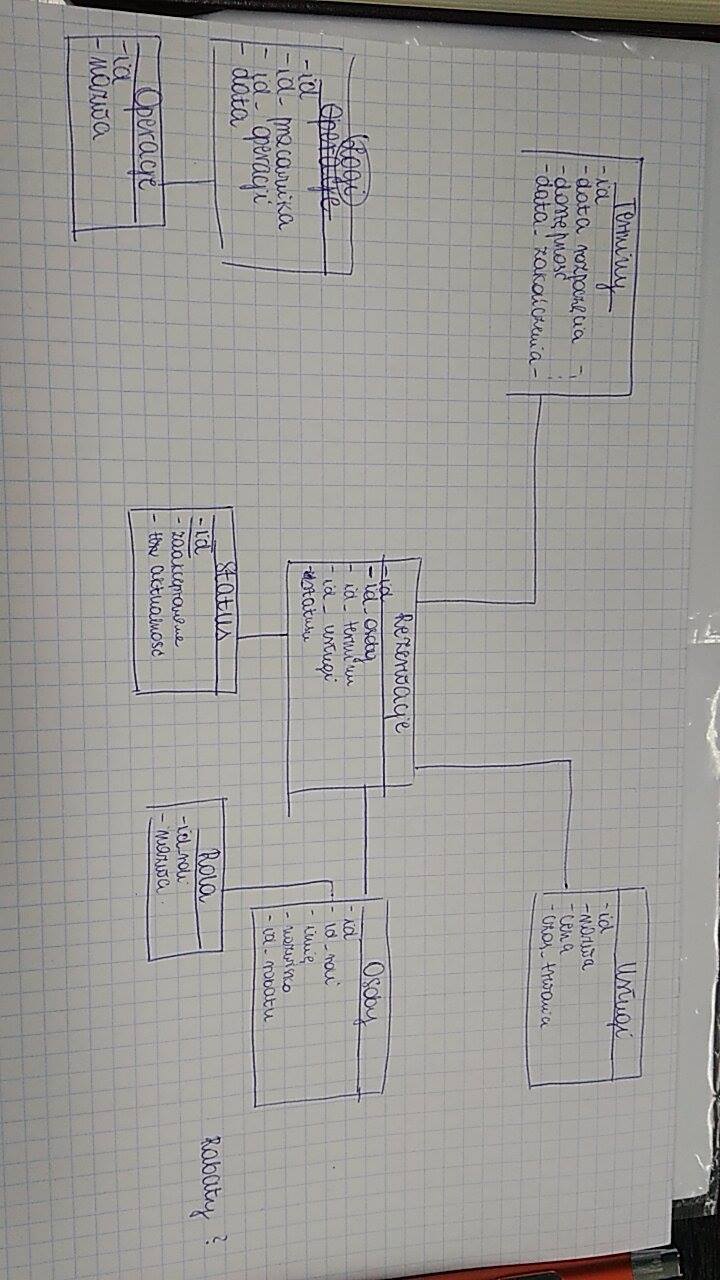
|  |  |
| --- | --- |
|  |  |
| **Bazy Danych**  **Projekt** | |
| **TEMAT:**  **Aplikacja usprawniająca działanie myjni samochodowej** | |
| **Uwagi prowadzącego:** | |
| **Data przyjęcia:**  **Podpis prowadzącego:** | **Imię i nazwisko:**  Krzysztof Skorupa  Rafał Faiński  Jan Kluczewski  Paulina Mrozek |
| **Grupa: IP20PP** |

# Etap projektowania aplikacji.

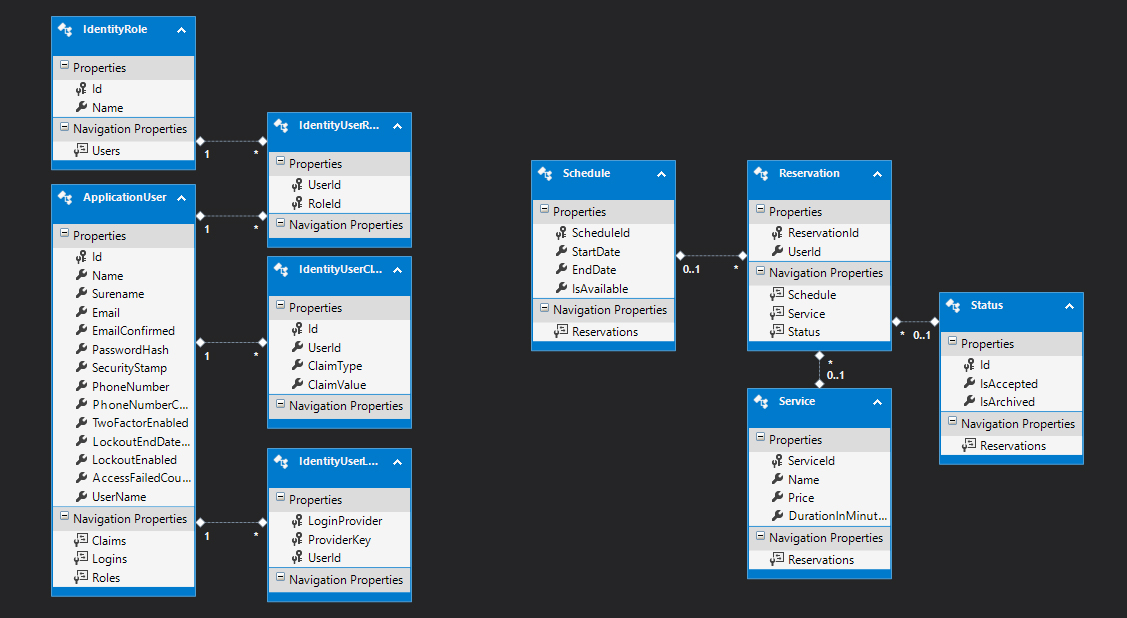
Podczas projektowania przedyskutowany został przebieg zdarzeń w aplikacji oraz zebrane wymagania od klienta. Aplikacja powinna:

1. Posiadać możliwość tworzenia kont użytkownika,
2. Przeglądania dostępnych terminów,
3. Rezerwacji terminu po zalogowaniu się,
4. Zapisywać logi z akcji potwierdzenia bądź odrzucenia rezerwacji,
5. Dla użytkowników o roli pracownika myjni powinny być dostępne następujące czynności:
   1. Potwierdzenie bądź odrzucenie terminu mycia,
   2. Przeglądanie zatwierdzonych terminów oraz logów związanych z wykonanymi akcjami

W pierwszej fazie zaproponowano następujący schemat bazy danych:



W fazie developmentu uległ on niewielkim zmianom. Ze względu na wykorzystanie frameworka ASP.Identity tabele związane z użytkownikami mają generyczne nazwy zgodne z konwencjami zaproponowanymi we frameworku. Dokładny diagram klas modeli przedstawiony został poniżej (jest on również zawarty w folderze z solucją):



Aplikacja została podzielona na 4 projekty umieszczone w jednej solucji:

1. CarWash.Core – zawiera logikę związaną z rezerwowaniem terminów, mapowaniem modeli domenowych na transferowe, wyznaczaniem wolnych terminów oraz z logowaniem,
2. CarWash.Persistence – zawiera wszystkie mechanizmy potrzebne do utrwalania zmian w bazie danych oraz modele reprezentujące obiekty w bazie danych,
3. CarWash.Tests – testy aplikacji,
4. CarWash.UI – Logikę odpowiedzialną za wystawione na zewnątrz API czyli kontrolery WebApi i konfigurację startową

# Wybór technologii

Do rozwiązania przedstawionego problemu zostało wybrane nowoczesne podejście polegające na wystawionym w backendzie API oraz aplikacji SPA na frontendzie, która będzie się komunikować z serwerem poprzez jego API.

**Backend:**

ASP.NET WebApi2 – jest to technologia stabilna oraz najbardziej nam znana ponieważ wykorzystuje język C#,

ASP.NET Identity (z OAuth2 oraz OWIN) – w prosty sposób pozwala zaimplementować zarządzanie kontami użytkowników,

MSSql Server – pozwala na łatwą integrację z ASP.NET,

EntityFramework (ORM) – ma pełne wsparcie dla MSSql Servera oraz ASP.NET

**Frontend:**

Angular – jego podejście komponentowe oraz gotowa powłoka warstwy komunikacji poprzez protoków Http, dodatkowo dobra integracja z TypeScript – ułatwia pracę z Javascriptem poprzez podejście bardziej statycznie typowane.

# Specyfikacja zewnętrzna

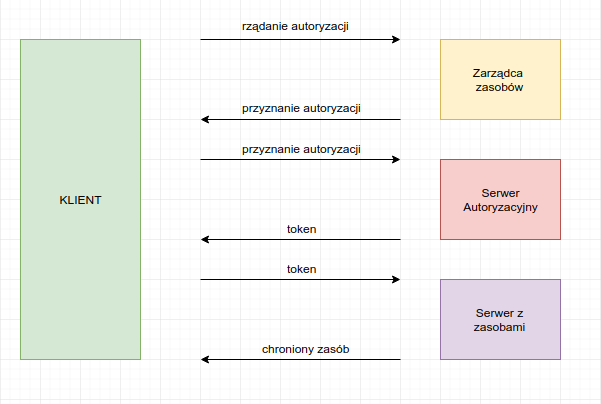
W wersji finalnej do uruchomienia aplikacji potrzebny jest server IIS obsługujący .NET Framework w wersji minimum 4.6. Ze względu na wybrane technologie sugerowana jest platforma Microsoft Azure. Na potrzeby DEMO aplikacji do uruchomienia wymagane są: lokalne środowisko Visual Studio do uruchomienia API oraz platforma Node.js (v7.10.0) z zainstalowanymi: npm’em (v5.3.0) oraz Angular-Cli (v1.1.3).

Po uruchomieniu backendu w Visual Studio należy w consoli Node.js w folderze głównym aplikacji Frontendowej |FolderGłówny|\CarWashFrontEnd uruchomić komendę: ng serve – uruchomi ona server testowy node’a a na nim część frontendową aplikacji. Aplikacja dostępna będzie pod adresem: <http://localhost:4200> z dowolnej przeglądarki.

# Specyfikacja wewnętrzna

W celu autentykacji aplikacji użytkownik wysyła rządanie do zarządcy aplikacji. Po uzuskaniu akceptacji ze strony zarządcy, serwer autoryzacyjny generuje dla użytkownika token, który będzie zezwalał na dostęp do chronionych zasobów na serwerze.

Powyższy proces został zaprezentowany na poniższym schemacie.



Token pozwala każdemu , kto jest w jego posiadaniu na dostęp do chronionych zasobów. W odróżnieniu od OAuth 1.0 nie jest wymagany żadny inny dowód tego,że token został wygenerowany dla konkretnego użytownika (ang. proof of possessioon).

W związku z tym aby bezpiecznie korzystać z Bearer Token należy spełnić następujące założenia:

- podczas nawiązywania połączenia klient musi upewnić się o tym że połączenie jest nawiązywane za pomocą chronionego protokołu np. https,

- niezalecane jest przechowywanie tokenów w cookies. Stwarza to większe możliwości przejęcia tokenu przez osoby nieporządane,

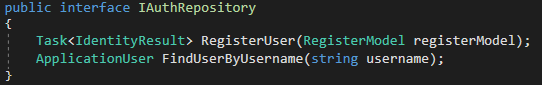
- serwer powienin dysponować możliwością przyznawania krótkotrwałych tokenów w sytuacjach gdy klienci korzystają z obcego komputera, aby zmniejszyć ryzyko przejęcia tokenu

**Szczególy implementacyjne:**

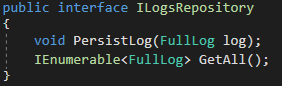
Do zarządzania operacjami zapisu i odczytu z bazy danych wykorzystano wzroce takie jak: UnitOfWork oraz Repository.

Stworzono następujące repozytoria:

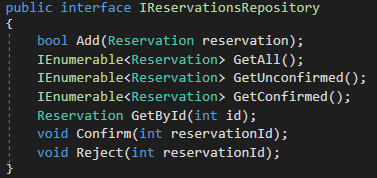
Auth – rejestruje użytkownika oraz nadaje mu domyślnie rolę: Customer, wyszukuje użytkownika po nazwie użytkownika

,

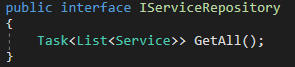
Logs – zapisuje logi w bazie



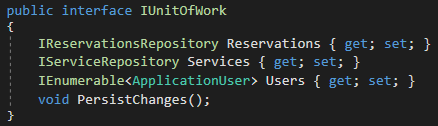
Reservations – pozwala na: dodanie rezerwacji, pobranie wszystkich, niezatwierdzonych, zatwierdzonych, pojedynczej po Id, zatwierdzenia oraz odrzucenia



Service – pozwala na pobranie wszystkich zdefiniowanych programów mycia

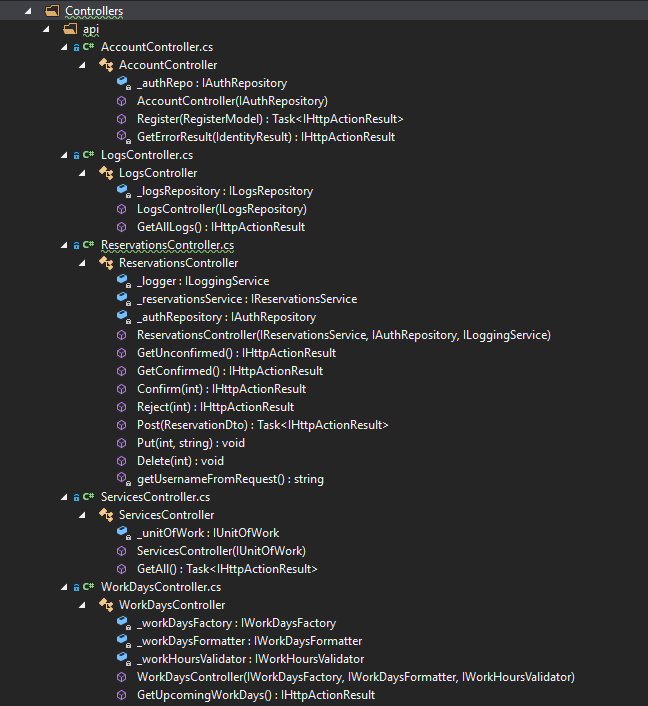


Unit of Work zawiera właściwości dostępu do poszczególnych repozytoriów oraz bazy użytkowników, dodatkowo metoda PersistChanges wykonuje zapis do bazy danych poprzez wywołanie metody SaveChanges na kontekście aplikacji (szczegół implementacyjny EntityFrameworka)



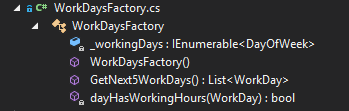
Aplikacja wykorzystuje 2 konteksty odpowiedzialne za persystowanie zmian do bazy danych. Zdecydowano się na rozdzielenie kontekstu osobno dla mechanizmu logowania. Baza danych oczywiście jest jedna, nierozłączona.

Kontrolery mają czytelne sygnatury metod więc nie jest wymagane ich szczegółowe omówienie, poniżej zamieszczono zestawienie wszystkich kontrolerów, wraz z ich metodami:



Mechanizm zwracania wolnych terminów jest bardziej skomplikowany i wymaga wyjaśnienia.

W pierwszym etapie fabryka



Zwraca listę 5 następnych dni roboczych.

1. Sprawdza obecną datę i na jej podstawie tworzy nowy obiekt WorkDay i zapisuje go do przeprocesowania.
2. Sprawdza czy dany dzień ma jakiekolwiek godziny pracy (ponieważ WorkDay podczas domyślnej inicjalizacji do listy swoich dostępnych godzin wpisuje wszystkie te, które mieszczą się w przedziale \_startHour i \_endHour – czyli godzinach pracy myjni fabryka może je sprawdzić) i jeśli ma dodaje ten dzień do listy dni dodanych, dzieje się to do momentu, gdy zostanie dodane 5 kolejnych dni. Fabryka powinna zawierać dodatkowo metodę tworzącą domyślny dzień – dzięki temu możnaby pozbyć się tego szczegółu implementacyjnego z klasy WorkDay ponieważ jego odpowiedzialnością nie powinna być wiedza, jak się zainicjalizować.

Następnie, gdy fabryka zwróci dni kontroler przekazuje je do walidatora – WorkHoursValidator, który:

1. Pobiera listę zajętych terminów,
2. Metoda getListOfScheduledHours przetwarza po kolei każdy z dni i sprawdza, czy jego godziny zawierają, któreś z zajętych,
3. W metodzie CheckAndUpdate po zwróceniu listy godzin do usunięcia z danego dnia są one usuwane,
4. Finalnie walidator zwraca zmodyfikowaną listę najbliższych 5 dni roboczych

W ostatnim kroku kontroler serializuje listę do formatu JSON i zwraca poprzez metodę Get do frontendu.

# Napotkane problemy i ich rozwiązania

Problem ze wstrzykiwaniem zależności do kontrolerów rozwiązano za pomocą DI Contenera – framework Autofac.

Gdzie umieścić modele? – dylemat

Na początku sensownym wydawało się umieszczenie modeli w projekcie Core, ponieważ reprezentowały one również obiekty domenowe. Były one jednak ‘bezmyślnymi’ obiektami więc pasowały również do projektu Persistence ponieważ w zasadzie służyły jedynie do reprezentacji obiektów w bazie danych. Pojawiły się dwa rozwiązania:

1. Wydzielenie osobnych modeli domenowych oraz tych, służących do persystencji a następnie zaimplementowanie mechanizmu mapującego
2. Pozostawienie ich w projekcie Persistence

Ze względu na mały rozmiar aplikacji i brak wyraźnej logiki domenowej obiektów wybrane zostało rozwiązanie drugie.

Rozdzielenie procesu rezerwacji na 3 fazy – wybór usługi, wybór terminu, potwierdzenie:

Przy takim podejściu w angularze potrzebne były 3 oddzielne komponenty, nie miały one jednak ze sobą żadnego powiązania więc nie było możliwe przekazywanie stanu rezerwacji z jednego do drugiego.

Problem został rozwiązany przez wydzielenie serwisu, który zbierał potrzebne elementy i każdy komponent podczas inicjalizacji pobierał stan rezerwacji.