Zaprojektuj aplikację (Web API), której celem będzie zarządzanie danymi pracownika w firmie.

Należy zaprojektować kontroler REST na drugim poziomie dojrzałości wg Richardsona.

Jako użytkownik API chciałbym móc:

- dodawać pracownika
- edytować pracownika (edycja wszystkich danych jednocześnie)

Od strony implementacji należy zadbać o:

1. Prawidłową separację odpowiedzialności
2. Przystosowanie do zastosowania Dependency Injection (nie musi być skonfigurowane)
3. Podział modelu na odpowiednie encje i value objecty
4. Napisanie kompletnych testów w okrojonym zakresie wymienionym w uwagach
5. Walidację modelu względem poniższej tabeli

Model pracownika:

Nazwa pola	**Typ**	**Liczba znaków**	**Czy wymagane**
Nr ewidencyjny	Numer zapisywany jako string z wiodącymi zerami (w sumie 8 znaków)	8	Tak, nadawane automatycznie przy tworzeniu nowego pracownika
Nazwisko	tekstowy	1-50	tak
Płeć	Enum	-	tak

Wymagania dotyczące obiektu pracownika:

- Nr ewidencyjny powinien się nadawać kolejno i automatycznie. Nie może się powtarzać.

Uwagi

Nazwy mogą, ale nie muszą zostać przetłumaczone na j. angielski na potrzeby implementacji.

Warstwa repozytorium może pozostać niezaimplementowana. Nie interesuje nas działający program.

Interesują nas natomiast kompletne i działające testy **jednostkowe** (i tylko jednostkowe) w zakresie:

- Kompletne testy dla obiektu pracownika (konstruktor i metoda aktualizująca dane)
- Jeden wybrany ValueObject (konstruktor)
- Jedna z metod handlera/serwisu: Tworzenie pracownika lub aktualizacja pracownika

Nie interesują nas testy kontrolera API, ewentualnych DTO, klas komend, ani repozytorium (może nie być jego implementacji). Zwracamy jednak **szczególną uwagę** na odpowiednie zamockowanie repozytorium w testach jednostkowych i prawidłowe wykorzystanie mocka.

Można użyć patternu Command/Handler, ale nie to jest najważniejsze. **Nie jest dla nas istotne stworzenie kompletnego rozwiązania mechanizmu obsługi komend.** W razie potrzeby proponujemy użycie np. biblioteki MediatR. W **przypadku nie użycia** tego patternu należy zwrócić szczególną uwagę na zachowanie odpowiedniego podziału odpowiedzialności klas (w szczególności zachowania single responsibility principle).

Będziemy zwracali uwagę na **zastosowanie dobrych praktyk programistycznych** i **prawidłowy podział klas ze względu na odpowiedzialności**.

Sugerujemy użycie value objectów dla danych pracownika i wykorzystanie ich do walidowania swoich wartości. Zwróć uwagę, że samo sprawdzenie czy value object nie jest nullem powinno odbywać się na poziomie obiektu pracownika. Value object jeśli jest prawidłowo stworzony, to na pewno jest prawidłowy, w przeciwnym wypadku jego konstruktor rzuca wyjątek.

Pamiętaj, że numer ewidencyjny to nie to samo co Id! Id zrób najlepiej jako Guid. Numer ewidencyjny ma być nadawany kolejno, na podstawie innych, istniejących pracowników. Celem tego elementu jest sprawdzenie czy potrafisz prawidłowo zaprojektować nadawanie kolejnego numeru. Podpowiem, że za nadawanie tego numeru jest odpowiedzialna domena, a nie baza danych. W dodatku ta operacja jest na tyle skomplikowana, że zasługuje na oddzielną klasę - nie wciskaj tego kodu na siłę do metody tworzącej pracownika. Możesz też zamodelować dodatkowe pomocnicze pola, jeśli będą przydatne przy nadawaniu kolejnego numeru.

Value objecty powinny być tworzone jak najwcześniej i wchodzić do domeny zamiast typów prostych, DTO itp.

Stwórz 2 biblioteki (Assembly): api + domena. W Api powinny być tylko kontrolery, które wywołują metody/serwisy/handlery z biblioteki domeny. W praktyce byłaby potrzebna jeszcze trzecia biblioteka “infrastructure” z implementacją repozytorium, ale to do zadania nie jest potrzebne. Osobno stwórz też oczywiście bibliotekę dla testów.

Im mniej kodu tym lepiej! (w ramach zdrowego rozsądku). **Nie pisz żadnego niepotrzebnego kodu** (który nie wynika z zadania)! Np. nie rób żadnej dodatkowej obsługi błędów. Nie zwracaj w rezultatach obiektów, o które nie prosiliśmy. Jeśli czegoś nie ma w wymaganiu, to ma tego nie być.

Nie projektuj modelu z myślą o mechanizmach dostępu do danych (Entity Framework itp.). Stwórz jak najładniejszy model obiektowy patrząc tylko z punktu widzenia kodu, wykorzystaj wszystkie zalety obiektowości i inne zasady poprawnego, czystego modelowania klas. Przyjmij, że repozytorium oczekuje i zwraca domenowy obiekt pracownika (nie dto). Nie interesuje nas co się dzieje w implementacji repozytorium.

Nie zastępuj konstruktorów statycznymi metodami tworzącymi bez uzasadnienia.

Nie twórz walidacji przez żadne rozwiązania korzystające z atrybutów. Każda klasa odpowiada za swoje dane i je waliduje.