flipcards.pl

DARMOWA PRÓBKA · 2026

Backend Python

na rozmowie technicznej



Python, FastAPI, PostgreSQL i testowanie.

Pytania, odpowiedzi i zadania praktyczne

Zajrzyj do środka

DARMOWA PRÓBKA

Dwa pełne rozdziały. Trzy laboratoria.

Przeczytaj 24 pytania z odpowiedziami i zobacz, jak przejść od kontraktu HTTP do bezpiecznej rezerwacji ostatniego miejsca. Fragmenty zachowują kod, omówienia rozwiązań, testy i podsumowania.

Rozdział 6 · HTTP i pierwsze API w FastAPI

Statusy odpowiedzi, błędy, OpenAPI i paginacja. L09: kontrakt katalogu warsztatów. Strony książki 87–99.

Rozdział 10 · Transakcje i odporność na ponowienia

Wyścigi, izolacja, rollback i idempotencja. L14: ostatnie miejsce. L15: odzyskanie wyniku po utracie odpowiedzi. Strony książki 147–161.

Jak poruszać się po próbce

Dalej znajdziesz wstęp i pełny spis treści. Numery na stronach i w spisie pochodzą z książki; przeskoki są zamierzone. Zakładki czytnika prowadzą do fragmentów próbki. Odsyłacze do pominiętych treści kierują na ostatnią stronę z opisem pełnego wydania.

Próbka ma 42 strony. Osobna zbiorcza paczka kodu jest dodatkiem do pełnej książki; ten PDF zawiera fragmenty kodu i opisy ćwiczeń.

FLIPCARDS.PL

Backend Python

na rozmowie technicznej

Python, FastAPI, PostgreSQL i testowanie.

Pytania, odpowiedzi i zadania praktyczne

Od wyjaśnienia do sprawdzenia
16 rozdziałów i 194 pytań z odpowiedziami.
24 laboratoria, 3 rozmowy próbne i 7 dodatków.
Kod startowy, rozwiązania, testy i materiały do samodzielnej powtórki.

Wydanie 2026

Stan materiałów: 19 listopada 2026

flipcards.pl

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Jak pracować z książką

Na rozmowie technicznej sama nazwa biblioteki rzadko zamyka temat. Trzeba jeszcze wyjaśnić, co robi kod, w jakich warunkach działa i jak sprawdzić jego ograniczenia. Dlatego każdemu większemu zagadnieniu towarzyszy problem z backendu rezerwacji warsztatów: od współdzielonej listy po utraconą odpowiedź, cudzy rekord i powiadomienie wysłane dwa razy.

Książka zakłada znajomość funkcji, warunków, pętli i podstawowych kolekcji. Nie wymaga doświadczenia z FastAPI ani SQLAlchemy. Jeśli nie masz pewności, od czego zacząć, wykonaj krótką diagnozę i L01 w rozdziale 1. Pokażą, czy potrafisz uruchomić projekt, odczytać błąd i wskazać właściwy interpreter. Do ćwiczeń z bazą potrzebujesz także lokalnego Dockera z Compose; szczegółowe wymagania podaje instrukcja danego projektu.

Jedna dziedzina, samodzielne ćwiczenia

Warsztat ma określoną liczbę miejsc. Użytkownik rezerwuje miejsce, anuluje zapis i odczytuje swoje dane. Organizator zarządza katalogiem i raportem. Ten sam kontekst pozwala skupić się na kolejnych mechanizmach, bez uczenia się za każdym razem nowej dziedziny.

Laboratoria są jednak samodzielnymi projektami lub wariantami ćwiczeń, a nie kolejnymi wersjami jednego repozytorium do bezpośredniego scalania. Wczesne przykłady używają pamięci procesu, stałej tożsamości albo jawnych atrap. Późniejsze badają rzeczywiste transakcje, podpis tokenu i trwałe zlecenia. Każdy rozdział wskazuje własną granicę. Przykładowo L11 pokazuje czas życia sesji przez atrapę, L12 używa PostgreSQL, a L14 odtwarza wyścig dwóch operacji. Ich wyniki odpowiadają na różne pytania.

Nie zakładaj, że model danych, nazwa modułu czy sposób uwierzytelnienia z jednego startera obowiązuje we wszystkich następnych. Zachowujemy reguły i rozumowanie, a zakres implementacji wynika z polecenia. L24 jest osobnym porównaniem Django i FastAPI, nie migracją całego backendu.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Jak czytać pytanie i rozwiązanie

Identyfikator **Q9.13** oznacza pytanie trzynaste w rozdziale dziewiątym. Oznaczenia Junior, Mid i Senior opisują zakres odpowiedzi: podstawę, samodzielne zastosowanie albo obronę gwarancji i kompromisów. Nie są oceną stanowiska ani obietnicą wyniku rekrutacji.

1. Przed lekturą odpowiedz własnymi słowami. Podaj mechanizm, mały przykład i warunek, w którym rozwiązanie przestaje wystarczać.
2. Przeczytaj wyjaśnienie i uruchom przykład, jeśli wynik Cię zaskoczył. Zapisz różnicę między przewidywaniem a obserwacją.
3. Otwórz starter laboratorium. Rozwiązanie referencyjne przeczytaj po samodzielnej próbie lub wtedy, gdy umiesz już nazwać miejsce blokady.
4. Po poprawce uruchom testy wskazane dla własnego wariantu. Wyjaśnij jedną asercję, która wykrywa wcześniejszy błąd, oraz ograniczenie testu.
5. Wróć do pytania bez otwartego rozwiązania. Plan kolejnych powrotów i arkusze rozmów zawiera rozdział 16.

Domyślne testy często sprawdzają rozwiązanie referencyjne. Zielony wynik nie dowodzi wtedy poprawności Twojego startera. Czytaj polecenia z opcją `--variant starter`. Testy oznaczone jako `demonstration` potwierdzają celowy błąd wersji początkowej; po naprawie korzystaj z zestawu kontraktu opisanego w danym laboratorium.

Wybierz ścieżkę lektury

Punkt wyjścia	Proponowana kolejność	Pierwszy sprawdzian
Pierwsza rozmowa backendowa	1–4, podstawy 5, następnie 6–9 i 11–12. Rozdział 10 uzupełnia gwarancje zapisu.	L01, L02–L06, potem S01.
Praca z istniejącym API	5, 8–14. Wracaj do języka, gdy problem dotyczy mutacji, iteratora lub typu.	L14–L19, następnie S02 i S03.
Doświadczenie w Java lub Node	1 dla środowiska, 2–5 dla semantyki Pythona, następnie 6–9.	L05–L08; wyjaśnij różnice w wykonaniu i stanie.
Praca z Django	15 oraz wspólne tematy: dane w 9–10, dostęp w 11, testy w 12 i wdrożenie w 14.	L24 i uzasadnienie wyboru frameworka.
Krótka powtórka	Dodatek G: wybór zagadnienia i pytań. Dodatek E: właściwe środowisko ćwiczenia.	Jedno pytanie, jeden kontrprzykład i jedna próba kodu.

Nie trzeba realizować wszystkich ścieżek. Po próbie zapisz lukę konkretnie: „nie odróżniam braku pola od null” jest lepszym punktem wyjścia niż „muszę powtórzyć Pythona”. W pierwszym przypadku wrócisz do L10; w drugim trudno ustalić następny krok.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Mapa laboratoriów

Numer strony w ostatniej kolumnie prowadzi do początku rozdziału. Dokładne polecenia znajdują się przy ćwiczeniu oraz w instrukcji jego projektu.

Zadanie	Wynik pracy	Katalog w practice/	Str.
L01	Odtworzone środowisko i pierwszy test	lab01	13
L02–L03	Deduplikacja, opłata i reguła czasu	structures	25
L04–L05	Import strumieniowy i izolacja stanu	functions	39
L06	Reguła domenowa poza infrastrukturą	models	54
L07–L08	Postęp pętli i limit wywołań	concurrency	69
L09	Sprawdzalny kontrakt HTTP	http	87
L10	Walidacja tworzenia, zmiany i odpowiedzi	validation	100
L11	Zależności i czas życia zasobów	dependencies	113
L12–L13	Trwały zapis, migracje i raport bez N+1	persistence	127
L14–L15	Ostatnie miejsce i odzyskanie wyniku	lab14, lab15	147
L16–L17	Dostęp do zasobu i weryfikacja JWT	lab16, lab17	162
L18–L19	Naprawa testów i sekwencje operacji	lab18, lab19	176
L20–L21	Ważność cache i trwałe powiadomienie	lab20, lab21	191
L22–L23	Próba wydania i pomiar wydajności	lab22, lab23	204
L24	Ten sam odczyt w Django i FastAPI	lab24	217
S01–S03	Próba językowa, zmiana API i incydent	sim01, sim02, sim03	229

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Kod, wersje i granice sprawdzenia

Materiały towarzyszące znajdują się w katalogu `practice/`. L01 ma oddzielne projekty startera i rozwiązania. Laboratoria rozdziałów 2–9 mają własne projekty, pliki `pyproject.toml` i `uv.lock`. L14–L24 oraz S01–S03 korzystają ze wspólnego środowiska w `practice/`. Przed instalacją przejdź do katalogu wskazanego w instrukcji. Samo uruchomienie pytest z dowolnego miejsca nie obejmuje całej książki.

Przykłady sprawdzano na CPythonie 3.14.2. Materiał obejmuje FastAPI 0.128.8, Pydantic 2, SQLAlchemy 2, PostgreSQL 18 oraz Django 5.2 z DRF. To opis przyjętych środowisk, nie lista najnowszych wydań. Dokładne wersje bibliotek określa lockfile wybranego projektu; różnice między jego numerami poprawek są zamierzone. Polecenie `uv sync --locked` sprawdza zgodność deklaracji z blokadą. Nie aktualizuj wszystkich zależności przed pierwszą próbą odtworzenia ćwiczenia.

W poleceniach używamy uv, aby interpreter i narzędzia należały do właściwego środowiska. Instrukcje rozróżniają lokalny test funkcji, komunikację ASGI w procesie, żądanie po TCP i pracę rzeczywistego serwera bazy. To istotna część wyniku: test z atrapą sesji nie dowodzi izolacji PostgreSQL, a pomiar na własnym komputerze nie jest obietnicą opóźnienia produkcji.

Zakres wcześniejszej weryfikacji opisują raporty `VALIDATION-*.md`. Raport rozdziału 10 nazywa się `VALIDATION.md`. Zapisują użyte środowisko, wykonane próby i ograniczenia. Raport `VALIDATION-BOOK.md` dotyczy wspólnego wydania i jego nawigacji. Rozdziały zawierają linki do dokumentacji; pełny wykaz pomocniczy znajdziesz w dodatku F. Kody PY, FAPI i DJ w notach źródłowych oznaczają polskie banki pytań Python, FastAPI i Django; powiązania opisuje `SOURCE_MAP.md` w materiałach książki.

Nawigacja podczas pracy

Spis treści prowadzi do rozdziałów i sekcji. Panel zakładek PDF pozwala rozwinąć także poszczególne pytania. Dodatek G zawiera pełny indeks według rozdziałów, zagadnień i poziomów. Kliknięcie identyfikatora, na przykład Q9.13, prowadzi bezpośrednio do pytania i odpowiedzi. Kolumna „Str.” podaje numer strony w tym wydaniu.

Numeracja stron jest wspólna dla całej książki: obejmuje okładkę, wstęp, spis treści, rozdziały i dodatki. Dzięki temu numer widoczny w dokumencie odpowiada numerowi strony pliku. Osobne podglądy rozdziałów mają własną paginację; identyfikatory pytań pozostają w nich takie same.

Zacznij od rozdziału odpowiadającego Twojej ścieżce. Po każdym ćwiczeniu zostaw krótki zapis: założenie, zmiana, dowód i pozostała granica. Te cztery punkty przydadzą się zarówno w następnej próbie, jak i podczas wyjaśniania decyzji na rozmowie.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Spis treści

Jak pracować z książką **3**

1 Środowisko i sposób pracy z Pythonem **13**

1.1 Interpreter i środowisko 14
1.2 Moduły, pakiety i pochodzenie importu 15
1.3 Deklaracja i odtwarzanie zależności 16
1.4 Trzy różne rodzaje kontroli 17
1.5 Konfiguracja aplikacji i usług 18
1.6 Krótka diagnoza przed laboratorium 19
1.7 L01: odtwórz środowisko i uruchom pierwszy test 19
1.8 Podpowiedzi i rozwiązanie L01 22
1.9 Ocena i powtórka 23
1.10 Źródła i dalsza lektura 24

2 Obiekty i struktury danych **25**

2.1 Nazwy, obiekty i kopiowanie 26
2.2 Kolekcje i koszt operacji 28
2.3 Liczby, pieniądze i chwile czasu 30
2.4 L02: pierwsze zdarzenie i kolejność 32
2.5 L03: opłata i granica czasu zapisów 34
2.6 Sprawdzenie obu laboratoriów i powtórka 37
2.7 Źródła i dalsza lektura 38

3 Funkcje, iteracja i zarządzanie zasobami **39**

3.1 Argumenty i wartości domyślne 40
3.2 Zakresy nazw i domknięcia 41
3.3 Funkcje jako wartości i dekoratory 42
3.4 Iterowalne obiekty, iteratory i generatory 43
3.5 Przepływy strumieniowe i itertools 44
3.6 Wyjątki i granica błędu 45
3.7 Menedżery kontekstu i odpowiedzialność za zasoby 46
3.8 L04: import JSONL z raportem odrzuceń 47
3.9 L05: stan funkcji, domknięcia i metadane 50
3.10 Powtórka przed rozmową 52
3.11 Źródła i dalsza praca 53

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

4 Klasy, typowanie i granice modeli . . . 54
4.1 Instancje, atrybuty i metody . . . 55
4.2 Dataclass, kompozycja i niezmienniki . . . 56
4.3 Dziedziczenie, MRO i metody specjalne . . . 57
4.4 Protocol i klasy abstrakcyjne . . . 58
4.5 Unie, słowniki i parametry typów . . . 59
4.6 Analiza statyczna i walidacja podczas wykonania . . . 61
4.7 Granice domeny, API i przechowywania . . . 62
4.8 L06: oddziel regułę od infrastruktury . . . 63
4.9 Powtórka przed rozmową . . . 67
4.10 Źródła i dalsza praca . . . 68

5 Współbieżność i programowanie asynchroniczne . . . 69
5.1 Współbieżność, równoległość i rodzaj pracy . . . 70
5.2 GIL i wariant free-threaded . . . 71
5.3 Wątki, procesy i executors . . . 72
5.4 Korutyna, zadanie i pętla zdarzeń . . . 73
5.5 TaskGroup i polityka błędów . . . 75
5.6 Timeout, anulowanie i zasoby . . . 76
5.7 Limity i blokujące biblioteki . . . 77
5.8 Wyścigi, blokady i dowód zachowania . . . 78
5.9 L07: znajdź blokowanie pętli zdarzeń . . . 79
5.10 L08: ogranicz liczbę równoczesnych zapytań . . . 82
5.11 Powtórka przed rozmową . . . 85
5.12 Źródła i dalsza praca . . . 86

6 HTTP i pierwsze API w FastAPI . . . 87
6.1 Żądanie HTTP i odpowiedzialność frameworka . . . 88
6.2 Zasoby, metody i statusy . . . 89
6.3 Parametry, dane wejściowe i routing . . . 90
6.4 Grupowanie operacji . . . 92
6.5 Błędy domenowe i walidacyjne . . . 92
6.6 OpenAPI jako sprawdzalny kontrakt . . . 93
6.7 Paginacja i ewolucja API . . . 94
6.8 L09: zaprojektuj kontrakt katalogu warsztatów . . . 95
6.9 Powtórka przed rozmową . . . 99
6.10 Źródła i dalsza praca . . . 99

7 Walidacja i kontrakty z Pydantic . . . 100
7.1 BaseModel, Field i zagnieżdżone dane . . . 101
7.2 Konwersja i tryb ścisły . . . 102
7.3 Walidatory pól i całego modelu . . . 102

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

7.4 Modele tworzenia, zmiany i odpowiedzi . . . 103
7.5 Brak pola, null i PATCH . . . 104
7.6 Kwoty, daty i serializacja . . . 105
7.7 Ochrona pól wewnętrznych . . . 106
7.8 L10: zabezpiecz granicę danych wejściowych . . . 107
7.9 Powtórka przed rozmową . . . 112
7.10 Źródła i dalsza praca . . . 112

8 Zależności i struktura aplikacji . . . 113
8.1 Depends, zależności zagnieżdżone i cache . . . 114
8.2 Zależności z yield i zakres sprzątania . . . 115
8.3 Lifespan, klient HTTP i pula połączeń . . . 116
8.4 Router, usługa i dostęp do danych . . . 118
8.5 Ustawienia aplikacji . . . 119
8.6 Granica transakcji a zamykanie sesji . . . 120
8.7 Fabryka aplikacji i podmiana zależności . . . 121
8.8 L11: udowodnij poprawne życie zasobów . . . 121
8.9 Powtórka przed rozmową . . . 125
8.10 Źródła i dalsza praca . . . 126

9 PostgreSQL i SQLAlchemy . . . 127
9.1 Model relacyjny i projekt schematu warsztatów . . . 128
9.2 Klucze, ograniczenia oraz typy dat i kwot . . . 129
9.3 SELECT, JOIN, agregacje i parametryzacja . . . 130
9.4 SQLAlchemy 2.x, modele, select i relacje . . . 131
9.5 AsyncSession, sesja na zadanie i pula połączeń . . . 132
9.6 Flush, commit, rollback i życie obiektu ORM . . . 134
9.7 N+1, eager loading i koszt pobierania danych . . . 135
9.8 Alembic i aktualizacja istniejącej bazy . . . 137
9.9 Indeksy oraz czytanie planu zapytania . . . 138
9.10 Laboratorium L12: przenieś rezerwacje do PostgreSQL . . . 139
9.11 Laboratorium L13: usuń problem N+1 . . . 143
9.12 Powtórka przed rozmową . . . 145
9.13 Źródła i dalsza praca . . . 146

10 Transakcje i odporność na ponowienia . . . 147
10.1 Reguły biznesowe i granica transakcji . . . 148
10.2 Izolacja i decyzja o przyznaniu miejsca . . . 149
10.3 Błędy i wycofywanie zmian . . . 152
10.4 Idempotencja i utracona odpowiedź . . . 154
10.5 Anulowanie i czas życia wyniku . . . 157
10.6 Granica systemu i testowanie . . . 158

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

10.7 L14: ostatnie miejsce . . . 159
10.8 L15: odzyskaj wynik po utracie odpowiedzi . . . 160
10.9 Powtórka przed rozmową . . . 161

11 Uwierzytelnianie i uprawnienia . . . 162
11.1 Tożsamość i decyzja o dostępie . . . 163
11.2 Sesja, token i dostawca tożsamości . . . 164
11.3 Weryfikacja tokenu . . . 166
11.4 Własność zasobu, role i zakresy . . . 168
11.5 Hasła i przeglądarka . . . 170
11.6 Obsługa odmów i testy . . . 171
11.7 L16: zablokuj dostęp do cudzej rezerwacji . . . 173
11.8 L17: weryfikacja tokenu . . . 174
11.9 Powtórka przed rozmową . . . 175

12 Testowanie backendu . . . 176
12.1 Granica testu i wartość asercji . . . 177
12.2 Pytest, dane i izolacja . . . 178
12.3 Klient asynchroniczny i cykl życia . . . 180
12.4 Podmiany, czas i sytuacje błędne . . . 181
12.5 Własności i sekwencje operacji . . . 183
12.6 Ocena testów i powtarzalne wykonanie . . . 185
12.7 L18: napraw testy, które przepuszczają błędy . . . 186
12.8 L19: sprawdź niezmienniki na sekwencjach . . . 187
12.9 Powtórka przed rozmową . . . 190

13 Integracje, cache i zadania w tle . . . 191
13.1 Granica zewnętrznej usługi . . . 192
13.2 Cache jako świadomie nieaktualna kopia . . . 193
13.3 Zadanie w pamięci i trwały zamiar . . . 195
13.4 Przejęcie, potwierdzenie i drugi odbiorca . . . 197
13.5 L20: określ dopuszczalną nieaktualność cache . . . 199
13.6 L21: zachowaj powiadomienie po awarii procesu . . . 201
13.7 Powtórka i zakończenie pracy . . . 203

14 Wydajność, obserwowalność i wdrożenie . . . 204
14.1 Najpierw wynik i warunki pomiaru . . . 205
14.2 Sygnały, które prowadzą do przyczyny . . . 206
14.3 Cztery różne ograniczenia . . . 207
14.4 Wydanie aplikacji na istniejących danych . . . 210
14.5 L22: wydanie, migracja i powrót do v1 . . . 213
14.6 L23: hipoteza, pomiar i poprawka . . . 214
14.7 Powtórka przed rozmową . . . 216

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

15 Django i wybór frameworka . . . 217
15.1 Co wybieramy razem z frameworkiem? . . . 218
15.2 Dane, zapytania i transakcje . . . 219
15.3 Kontrakt API i dostęp do danych . . . 221
15.4 Wspólne testy i granica async . . . 223
15.5 Decyzja oparta na wymaganiach . . . 224
15.6 L24: ten sam odczyt w Django i FastAPI . . . 225
15.7 Powtórka przed rozmową . . . 228

16 Rozmowy próbne i powtórka . . . 229
16.1 Przygotowanie i sposób pracy . . . 230
16.2 S01: Python i poprawność danych . . . 232
16.3 S02: zmiana API i testy . . . 234
16.4 S03: incydent rezerwacji i powiadomień . . . 236
16.5 Omówienie rozwiązań . . . 238
16.6 Testy, dowody i obrona decyzji . . . 241
16.7 Arkusz oceny i informacja zwrotna . . . 242
16.8 Powtórka na 7, 14 i 30 dni . . . 243
16.9 Zamknięcie próby . . . 245

A Kolekcje, mutowalność i iteracja . . . 246
A.1 Dobór struktury do wymagania . . . 246
A.2 Referencja, kopia i własność danych . . . 247
A.3 Typowe koszty operacji . . . 248
A.4 Iterator i zużycie strumienia . . . 249

B Jak wybrać sposób wykonania . . . 250
B.1 Tabela decyzji . . . 250
B.2 Co uruchamia FastAPI, a co uruchamiasz sam . . . 251
B.3 Właściciel zadania i granice czasu . . . 252
B.4 Plan próby przed optymalizacją . . . 252

C Mapa kontraktu API i dostępu . . . 253
C.1 Granice odpowiedzialności . . . 253
C.2 Metoda, wynik i ponowienie . . . 254
C.3 Walidacja wejścia i publiczny model . . . 255
C.4 Macierz dostępu do zasobu . . . 256

D Dane, transakcje i idempotencja . . . 257
D.1 Sesja i transakcja w skrócie . . . 257
D.2 Reguła i mechanizm jej ochrony . . . 258
D.3 Kontrakt idempotencji na przykładzie L15 . . . 259
D.4 Awaria przed i po zatwierdzeniu . . . 259
D.5 Lista kontrolna dostępu do danych . . . 260

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

E Komendy i mapa środowisk 261
E.1 Katalogi, warianty i usługi 261
E.2 Odtworzenie i diagnoza środowiska 262
E.3 Pytest, Ruff i kontrola typów 263
E.4 Docker Compose i trwałość danych 264
E.5 Alembic: interfejs L12 i zwykły projekt 264
E.6 Skład i utrzymanie indeksu książki 265

F Słownik i źródła 266
F.1 Python i wykonanie programu 266
F.2 HTTP, modele i dostęp 267
F.3 Dane, niezawodność i testy 268
F.4 Źródła: język i współbieżność 270
F.5 Źródła: HTTP, modele i bezpieczeństwo 270
F.6 Źródła: baza, testy i narzędzia 271

G Indeks pytań 272
G.1 Zakres i poziomy 272
G.2 Pytania według zagadnienia 273
G.3 Pełny indeks według rozdziałów 274
G.4 Pytania według poziomu 284
G.5 Od indeksu do sesji powtórkowej 286

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

6

HTTP i pierwsze API w FastAPI

Katalog wyświetla dwa warsztaty, lecz po przejściu na drugą stronę pokazuje je ponownie. Przycisk rezerwacji dostaje 200 z napisem „brak miejsc". Dokumentacja obiecuje 201, a adres nowego zasobu nie daje się odczytać. Każda funkcja może wyglądać rozsądnie osobno. Dopiero kontrakt całego API pozwala rozstrzygnąć, co aplikacja ma rzeczywiście robić.

W rozdziale 5 sprawdzaliśmy, czy inne zadania robią postęp podczas I/O. Teraz określimy znaczenie żądania i odpowiedzi: metodę, adres, dane, status oraz skutek dla stanu. Projekt przewodni nadal dotyczy rezerwacji warsztatów. W tym etapie stan jest demonstracyjny i żyje w pamięci procesu.

> **Po tym rozdziale**
>
> Wyjaśnisz rolę HTTP, ASGI, Starlette i FastAPI. Zaprojektujesz statusy, parametry i publiczną odpowiedź, sprawdzisz OpenAPI oraz ustalisz stabilne sortowanie. Otrzymujesz 12 pełnych odpowiedzi, laboratorium L09, starter, rozwiązanie, 46 testów oraz zapis próby przez TCP.

Projekt `practice/http` instaluje pakiet `workshop_http`. Nie wymaga bazy ani wcześniejszych ćwiczeń. Tożsamość użytkownika jest stała i jawnie demonstracyjna. Nie jest to implementacja logowania. Transakcje i ponowienia rozwija rozdział 10, a uprawnienia rozdział 11.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

6.1 Żądanie HTTP i odpowiedzialność frameworka

Q6.1: Co składa się na kontrakt HTTP i co dzieje się z żądaniem w aplikacji? [Junior]

Żądanie określa metodę, docelowy zasób, nagłówki i ewentualną treść. Odpowiedź zawiera status, nagłówki i ewentualną reprezentację danych. Kontrakt obejmuje również skutek operacji: czy zmienia stan, jakie warunki sprawdza i co klient może zrobić po otrzymaniu odpowiedzi. Samo „zwracamy JSON" nie opisuje zachowania rezerwacji.

W L09 klient wysyła POST pod adres kolekcji rezerwacji warsztatu. Nagłówek Content-Type opisuje format przesłanej treści; Accept wyraża preferencje dotyczące odpowiedzi. Nie są zamienne. Dane JSON zawierają jedno miejsce. Tożsamość nie pochodzi z tego dokumentu. Po sukcesie klient dostaje 201, identyfikator rezerwacji oraz Location wskazujące zasób, który można odczytać.

Serwer Uvicorn obsługuje połączenie i wywołuje aplikację ASGI. Routing wybiera operację. FastAPI odczytuje zadeklarowane parametry, rozwiązuje zależności i korzysta z walidacji danych. Nasza funkcja wywołuje regułę rezerwacji. Następnie powstaje odpowiedź zgodna z publicznym modelem. Walidacja poprawnego int nie odpowiada na pytanie, czy w warsztacie zostało miejsce.

Test kontraktu musi więc sprawdzić więcej niż obecność klucza id. Weryfikuje status, kształt body, nagłówek Location, odczyt utworzonego zasobu i zmianę dostępności. Dla odrzucenia sprawdza brak zmiany stanu. W ten sposób klient i backend mają wspólny, obserwowalny punkt odniesienia.

```
POST /workshops/10/reservations HTTP/1.1
Host: 127.0.0.1:8009
Content-Type: application/json
Accept: application/json

{"seats": 1}
```

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /reservations/2

{"id": 2, "workshop_id": 10, "user_id": 101}
```

Pokazano istotne elementy wymiany; serwer i klient dodają pozostałe nagłówki transportowe. Id 2 wynika z danych początkowych L09.

Q6.2: Za co odpowiadają Uvicorn, ASGI, Starlette, Pydantic i FastAPI? [Junior]

Uvicorn jest serwerem, ASGI opisuje interfejs między serwerem a aplikacją. Aplikacja ASGI otrzymuje opis połączenia oraz funkcje odbierania i wysyłania komunikatów. Nie należy utożsamiać tego interfejsu z formatem JSON. ASGI obejmuje także inne rodzaje komunikacji, w tym WebSocket i cykl życia aplikacji.

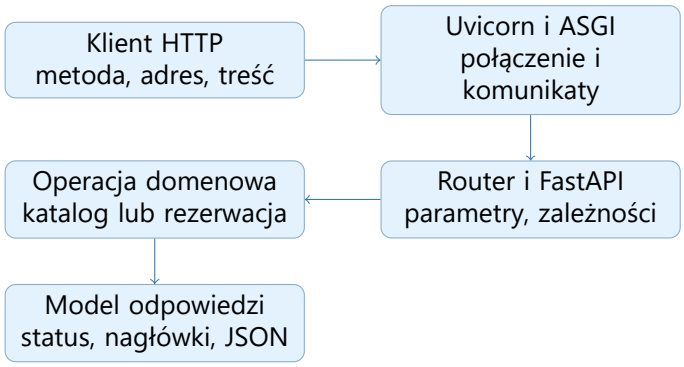
Starlette dostarcza elementy warstwy webowej, między innymi routing, żądania, odpowiedzi i middleware. FastAPI korzysta z tych elementów, łączy deklaracje parametrów z walidacją

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Pydantic, obsługuje zależności oraz generuje dokument OpenAPI. Pydantic opisuje i sprawdza dane; nie przydziela transakcji ani nie gwarantuje dostępności miejsca.

W klasycznym WSGI aplikacja pracuje przez synchroniczny interfejs żądania i odpowiedzi. ASGI pozwala opisywać asynchroniczną wymianę komunikatów. Sam wybór ASGI nie zmienia blokującego SDK w klienta async. Mechanizm z L07 nadal obowiązuje: blokujące wywołanie wewnątrz async def może zatrzymać wątek pętli. L09 wykonuje krótkie operacje w pamięci, więc nie udaje dostępu do asynchronicznej bazy.

Nie przypisuj frameworkowi dowolnej przewagi wydajnościowej. O szybkości konkretnego endpointu decydują także zapytania, serializacja, rozmiar odpowiedzi i zależności. L09 dowodzi poprawności kontraktu na małym zbiorze, nie przepustowości. Jeśli pytanie na rozmowie dotyczy architektury, wskaż odpowiedzialność każdej warstwy oraz to, gdzie znajduje się reguła biznesowa.



6.2 Zasoby, metody i statusy

Q6.3: Jak dobrać metodę HTTP i odróżnić bezpieczeństwo od idempotencji? [Junior]

GET odczytuje reprezentację, POST w naszym kontrakcie tworzy rezerwację w kolekcji warsztatu. PUT zwykle opisuje utworzenie lub zastąpienie stanu zasobu pod wskazanym adresem, a PATCH częściową zmianę według przyjętego formatu. DELETE usuwa wskazaną relację lub zasób. Nazwy metod mają semantykę; nie wybieramy GET do rezerwacji tylko dlatego, że łatwo otworzyć adres w przeglądarce.

Metoda bezpieczna nie prosi o zmianę stanu zasobu, choć serwer może zapisać log techniczny. Idempotencja dotyczy zamierzonego efektu wielokrotnego wykonania, nie identyczności każdego statusu i body. Pierwsze usunięcie może dać 204, a następne 404, mimo że zasób pozostaje usunięty. POST nie jest z definicji idempotentny; można zaprojektować dla niego osobny mechanizm ponowień.

L09 zwraca 409 po drugim żądaniu tego samego użytkownika. Chroni to regułę jednej rezerwacji w wykonaniu sekwencyjnym, ale nie odtwarza pierwszej odpowiedzi po utracie połączenia. Klient nie otrzymuje automatycznie tej samej reprezentacji 201. Nie nazywaj tego obsługą klucza idempotencji. Ten mechanizm budujemy dopiero w L15.

Adres powinien wskazywać zasób lub kolekcję, a metoda operację. Kolekcja /workshops/10/reservations jasno łączy rezerwację z warsztatem. Nie potrzebujemy /createReser-

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

vation jako GET. Test L09 dodatkowo potwierdza, że odczyty katalogu nie przydzielają identyfikatorów ani nie zmieniają rezerwacji.

Q6.4: Kiedy zwrócić 200, 201, 202 lub 204 i jaką rolę ma Location? [Junior]

200 opisuje pomyślną obsługę operacji, na przykład odczyt katalogu. 201 informuje o utworzeniu zasobu. W L09 rezerwacja istnieje już w pamięci, a body przedstawia jej identyfikator, warsztat i użytkownika. Nagłówek Location wskazuje nowy zasób. Względny URI /reservations/2 jest częścią naszego kontraktu i można rozwiązać go względem adresu serwera.

202 oznacza przyjęcie do przetwarzania, którego wynik nie musi być jeszcze gotowy. Nie zastępuje 201 tylko dlatego, że endpoint ma async. Jeżeli operacja ma być wykonywana później, trzeba opisać sposób sprawdzania jej stanu i ewentualnego błędu. Sam status nie zapewnia trwałego zlecenia. Takie problemy rozwija rozdział o zadaniach w tle.

204 oznacza pomyślne zakończenie bez treści odpowiedzi. Nie dodajemy do niego JSON z podsumowaniem. Jeśli klient potrzebuje reprezentacji, wybieramy odpowiedni status z treścią. Statusy sukcesu różnią się znaczeniem, a nie tylko preferencją zespołu.

W FastAPI status sukcesu deklarujemy w operacji. Nagłówek można ustawić przez przekazany obiekt Response, nadal zwracając model danych. L09 sprawdza status i odczytuje adres z Location. To wykryje zarówno przypadkowe domyślne 200, jak i nagłówek wskazujący nieistniejącą trasę. Responses w OpenAPI dokumentuje ten nagłówek, lecz go nie wytwarza.

6.3 Parametry, dane wejściowe i routing

Q6.5: Co umieścić w path, query i body oraz jak ustalić granice parametrów? [Junior]

Path identyfikuje zasób, query opisuje sposób odczytu, a body przenosi dane operacji. W L09 warsztat wybieramy przez workshop_id w ścieżce, stronę katalogu przez limit i offset, a żądane jedno miejsce przez seats w JSON. Jest to decyzja kontraktowa: przeniesienie identyfikatora użytkownika do body nie czyni go wiarygodnym.

Deklaracja parametru określa także wymaganie i wartość domyślną. Limit bez wartości w query dostaje 20. Gdy klient poda 0, nie zastępujemy go domyślnym 20, lecz zwracamy 422. Dopuszczamy 1–50. Offset jest w przedziale 0–10000. Ograniczenia zapobiegają przypadkowemu pobraniu dowolnie dużej odpowiedzi, choć nie są limitem ruchu ani całkowitego kosztu zapytania.

Query i path przychodzą jako tekst. FastAPI może przekształcić /workshops/10 w dodatni int zgodnie z deklaracją. „Typowany parametr" nie oznacza, że transport wysłał obiekt Python. Body ma oddzielny schemat: w tym ćwiczeniu seats jest ścisłym int równym 1. Bool, float, napis oraz dodatkowe pola są odrzucane. Szczegóły konwersji i modeli rozwija rozdział 7.

Nie polegaj na tym, że klient używa przygotowanego formularza. Testuj ujemny limit, zbyt duży limit, brak wymaganego body i dodatkowy user_id. Sprawdź też lokalizację błędu: path, query lub body. Nie musisz utrwalać każdego angielskiego komunikatu biblioteki, jeśli nie jest częścią publicznego zobowiązania API.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

```
class ReservationCreate(BaseModel):
    model_config = ConfigDict(extra="forbid")
    seats: Annotated[int, Field(strict=True, ge=1, le=1)]
```

Model wejścia korzysta z importowanych BaseModel, ConfigDict, Field i Annotated. Pełne importy i wykonywalny kontekst są w schemas.py.

Tak wygląda operacja katalogu z fabryki create_app. Dekorator łączy metodę i adres z funkcją. Query zapisuje ograniczenia oraz wartości domyślne w deklaracji, a response_model określa odpowiedź:

```
@router.get("/workshops", response_model=Page, operation_id="listWorkshops")
async def list_workshops(
    limit: Annotated[int, Query(ge=1, le=50)] = 20,
    offset: Annotated[int, Query(ge=0, le=10000)] = 0,
) -> Page:
    return catalog(data, limit, offset)
```

Router należy do fabryki, a catalog jest wybraną funkcją startera lub rozwiązania. Parametry data i catalog pochodzą z otaczającego zakresu. Na końcu fabryki app.include_router(router) rejestruje te operacje. Ten fragment wymaga pełnego kontekstu app.py; nie jest osobnym plikiem uruchomieniowym. Polecenie python -m workshop_http.app tworzy aplikację i przekazuje ją do Uvicorn.

Q6.6: Dlaczego kolejność tras ma znaczenie mimo adnotacji int? [Mid]

Router dopasowuje ścieżki według zarejestrowanej kolejności. Jeżeli /workshops/{workshop_id} znajduje się przed /workshops/stats, napis stats może trafić do parametru dynamicznej operacji. Dopiero potem walidacja próbuje uzyskać int i odrzuca dane. Nie oznacza to automatycznego szukania następnej trasy.

Dlatego w L09 najpierw rejestrujemy /workshops i /workshops/stats, a potem trasę szczegółów. Test stats oczekuje 200 i liczby warsztatów. Sam test /workshops/10 nie wykryłby błędu kolejności. Adnotacja typu parametru w funkcji i konwerter ścieżki routera to różne mechanizmy; nasz przykład używa zwykłego parametru z walidacją FastAPI.

Zwróć również uwagę na końcowy ukośnik. Projekt deklaruje /workshops bez ukośnika; domyślne zachowanie użytej wersji daje 307 dla /workshops/. W testach wyłączamy automatyczne podążanie za przekierowaniem, aby zobaczyć rzeczywisty status i Location. Inaczej klient testowy mógłby ukryć etap przekierowania.

405 informuje, że pod danym adresem nie obsługujemy żądanej metody. L09 sprawdza DELETE dla szczegółów warsztatu i nagłówek Allow. To inny przypadek niż 404 dla nieistniejącego id lub 422 dla napisu zamiast liczby. Rozróżnienie ułatwia klientowi i zespołowi diagnozę błędnego wywołania.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

6.4 Grupowanie operacji

Q6.7: Co daje APIRouter i gdzie powinna znajdować się logika rezerwacji? [Mid]

APIRouter grupuje operacje, ich metadane oraz wspólne ustawienia. Po dołączeniu przez include_router operacje stają się częścią aplikacji i jej dokumentacji. Router nie jest osobnym serwerem ani izolowaną bazą. Podział na pliki nie tworzy automatycznie granicy transakcji lub bezpieczeństwa.

W małym L09 fabryka tworzy router w app.py. Domena, schematy i operacje mają osobne moduły. To wystarcza, aby testować różne aplikacje z nowym stanem oraz kontrolowanym zegarem. Gdy liczba tras wzrośnie, można przenieść router do osobnego modułu i nadać mu wspólny prefiks. Nie trzeba rozbijać każdej prostej funkcji na kolejne warstwy.

Reserve zna Store, identyfikatory i czas. Zgłasza NotFound lub Conflict, ale nie importuje FastAPI. Adapter decyduje o statusie HTTP i Location. Dzięki temu reguła „nie po rozpoczęciu" nie staje się regułą właściwą wyłącznie jednej trasie. Catalog z kolei zwraca stronę danych; HTTP odpowiada za walidację limitu i offset.

Fabryka otrzymuje opcjonalny Store i zegar. Bez argumentu tworzy nowe dane, zamiast współdzielić globalny słownik pomiędzy testami. Test dwóch fabryk potwierdza izolację. Nie oznacza ona wspólnego magazynu dla procesów produkcyjnych: każdy proces nadal miałby własną pamięć. Zmiana katalogu plików tego nie rozwiązuje.

6.5 Błędy domenowe i walidacyjne

Q6.8: Jak odróżnić 404, 409 i 422 w operacji rezerwacji? [Junior]

404 w naszym kontrakcie oznacza brak wskazanego zasobu. 409 oznacza konflikt z aktualnym stanem: warsztat już się rozpoczął, nie ma miejsc albo użytkownik już ma rezerwację. 422 sygnalizuje odrzucenie danych przez warstwę walidacji FastAPI, na przykład ujemny identyfikator, limit 51 lub brak seats.

Nie sprowadzaj tego do reguły „400 to logika, 422 to poprawny JSON". W użytej wersji FastAPI także uszkodzony dokument JSON trafia do odpowiedzi 422 z typem json_invalid; projekt ma na to test. Rozdziel ogólną semantykę HTTP od zachowania wybranego frameworka. Przy zmianie obsługi błędów sprawdź również zgodność dokumentacji i klientów, zamiast zmieniać tylko numer statusu.

Dla 404 i 409 L09 zwraca detail jako stały kod tekstowy. Dla 422 zachowuje listę błędów z lokalizacjami. To dwa jawnie opisane kształty odpowiedzi. Klient powinien sprawdzać status i kontrakt danego błędu, a nie traktować detail zawsze jako napis. Nie zwracamy 200 z polem error dla odmowy rezerwacji.

Kolejność kontroli również ma znaczenie. Najpierw pobieramy warsztat, potem sprawdzamy czas, duplikat i pojemność. Jeśli warsztat jest pełny, a użytkownik już ma miejsce, dostaje already_reserved. W każdym wariancie odmowy zachowujemy dane i licznik identyfikatorów. To konkretna reguła L09, nie narzucona przez HTTP kolejność biznesowa.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Q6.9: Gdzie mapować wyjątki i dlaczego nie zamieniać wszystkiego na 400? [Mid]

W małym adapterze można przechwycić znany wyjątek domenowy i podnieść HTTPException. Używamy raise, nie return. Gdy identyczne mapowanie pojawia się w wielu trasach, warto rozważyć zarejestrowany handler wyjątku. Nadal jest to odpowiedzialność warstwy webowej, a nie obiektu reprezentującego warsztat.

L09 przechwytuje tylko NotFound i Conflict. Literówka w kodzie, uszkodzony stan lub nieoczekiwany wyjątek nie stają się rzekomo błędnym żądaniem użytkownika. Ogólne except Exception z odpowiedzią 400 ukryłoby przyczynę, utrudniło obserwację awarii i wprowadziło klienta w błąd. W produkcji błąd serwera wymaga kontrolowanego komunikatu oraz bezpiecznej diagnostyki po stronie operatora.

Walidacja odpowiedzi to także inna granica niż walidacja wejścia. Jeśli serwer nie potrafi wytworzyć zadeklarowanego modelu, problem leży w implementacji. Nie należy obwiniać klienta statusem 422 za własny niepoprawny wynik. Sam model wyjściowy pomaga ograniczyć zwracane pola, ale nie zastępuje kontroli, czy użytkownik może odczytać dany rekord.

L09 zwraca 404 zarówno dla brakującej, jak i obcej rezerwacji. Filtr właściciela działa w przykładzie ze stałą tożsamością. Nie dowodzi ochrony przed podszyciem się w rzeczywistym systemie, bo adapter demonstracyjny nie weryfikuje użytkownika. Zachowujemy tę granicę jawnie, zamiast udawać gotowe logowanie.

6.6 OpenAPI jako sprawdzalny kontrakt

Q6.10: Co potwierdza OpenAPI, a czego sama dokumentacja nie gwarantuje? [Mid]

OpenAPI jest dokumentem opisującym operacje, parametry, schematy i odpowiedzi. Swagger UI oraz ReDoc prezentują ten dokument. W projekcie domyślne adresy to /openapi.json, /docs i /redoc. Interaktywny formularz ułatwia wykonanie żądania, lecz nie jest dowodem poprawności reguł biznesowych.

FastAPI generuje schemat na podstawie deklaracji operacji i modeli. Dodatkowe responses opisują 404 i 409, których nie da się wywnioskować wyłącznie z modelu sukcesu. Te deklaracje nie rzucają wyjątku i nie sprawdzają samodzielnie każdej odpowiedzi błędu. Kod nadal musi ustawić status, treść i nagłówki. W L09 OpenAPI opisuje również Location dla 201.

Testy schematu sprawdzają pięć unikalnych operationId, wymagane body, model rezerwacji, ograniczenia limitu, statusy i brak pola wewnętrznego w publicznym modelu. Nie porównujemy całego dokumentu bajt po bajcie: zmiana opisu lub kolejności kluczy nie powinna przysłonić istotnej zmiany kontraktu. Testy zachowania oddzielnie sprawdzają rzeczywiste odpowiedzi i brak zapisu po odmowie.

Wyłączenie /docs nie chroni operacji ani danych. Z kolei dokument może obiecywać zachowanie, którego kod nie realizuje: starter L09 przechodzi testy OpenAPI, a mimo to ignoruje pojemność. W pracy zespołowej przeglądaj zmianę schematu razem z testem konkretnego żądania. Wersja info.version opisuje API, nie wersję samej specyfikacji OpenAPI.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

6.7 Paginacja i ewolucja API

Q6.11: Jak zaprojektować stabilną paginację i czego offset nie gwarantuje? [Mid]

Najpierw ustalamy pełny porządek, potem wycinamy stronę. Sortowanie wyłącznie po starts_at nie rozstrzyga kolejności warsztatów z tą samą datą. Dodajemy unikalny id jako drugi klucz. Na niezmienionym zbiorze kolejne strony mają wtedy przewidywalne granice, niezależne od kolejności wstawiania do słownika.

Page zawiera items, total, limit, offset i next_offset. Total oznacza liczbę wszystkich pozycji w katalogu, przed stronicowaniem. Jeżeli offset jest za końcem, zwracamy 200 z pustą listą i next_offset równym null. Nie jest to 404: kolekcja istnieje, ale wybrana strona nie zawiera elementów. Brak pojedynczego warsztatu jest inną sytuacją.

Gdy między pierwszym i drugim żądaniem ktoś doda rekord na początku, offset może ponownie wskazać już widziany element. Stabilny sort nie tworzy migawki wielu żądań. Kursor oparty na parze starts_at i id może lepiej kontynuować od ostatniego elementu, lecz wymaga własnego kontraktu: kierunku, filtrów, kodowania i zachowania przy zmianach danych. Nie gwarantuje automatycznie niezmiennej historii.

Limit odpowiedzi nie oznacza małego kosztu całej implementacji. L09 sortuje pełną listę w pamięci i liczy dostępność na małym zbiorze. W bazie trzeba dobrać zapytanie, indeks i sposób liczenia total. Nie przenoś demonstracyjnego kodu do milionowego katalogu wyłącznie dlatego, że zwraca tylko 20 pozycji. Wydajność zapytań omawiamy później.

Q6.12: Jak zmieniać API i publiczne modele bez przypadkowego zerwania klientów? [Mid]

Zmiana nazwy pola, typu, wymagania body lub znaczenia statusu może zerwać klienta nawet wtedy, gdy adres się nie zmieni. Zmiana domyślnego sortowania również wpływa na zachowanie stron. Dlatego kontrakt obejmuje nie tylko JSON Schema, ale też semantykę i reguły opisane w tabeli ćwiczenia.

Dodanie opcjonalnego pola często ułatwia kompatybilną ewolucję, ale nie jest bezwarunkowo bezpieczne: klient może rygorystycznie odrzucać nieznane klucze. Najpierw sprawdź znanych odbiorców, ich sposób walidacji i politykę wersjonowania. Jeżeli zmiana jest niezgodna, zapewnij okres przejścia albo osobny wariant API zgodnie z zasadami produktu. Sam prefiks /v2 nie migruje żadnego klienta.

Publiczny model oddzielamy od wewnętrznego rekordu. Dodanie notatki organizatora do Workshop nie powinno ujawnić jej w katalogu. W L09 present jawnie wybiera pola, a model wyjściowy i test sprawdzają publiczny kształt. Nie zwracamy bez namysłu całego słownika obiektu tylko po to, aby uniknąć kilku linii mapowania.

Przy przeglądzie zmiany zadaj trzy pytania: jakie dotychczas poprawne żądanie przestanie działać, jaka odpowiedź zmieni znaczenie i który test pokaże tę różnicę? Na rozmowie taka analiza jest bardziej użyteczna niż deklaracja „API jest REST”. Kolejny rozdział rozwinie modele wejścia, wyjścia i częściowych aktualizacji.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

6.8 L09: zaprojektuj kontrakt katalogu warsztatów

Cel, środowisko i dane

Poziom podstawowy, 45–60 minut bez instalacji i lektury rozdziału. Znasz funkcje, adnotacje i podstawy pytest. W repozytorium wejdź do `practice/http`; po rozpakowaniu paczki do `lab09`. Uruchom:

```
uv sync --locked
uv run --locked pytest -q
uv run --locked pytest -q --variant starter
```

Referencja ma 46 sukcesów. Początkowy starter daje 11 błędów i 35 sukcesów, bez xfail. Domyślny zielony wynik sprawdza solution.py, więc nie potwierdza naprawy starter.py. Dostarczono router, modele, mapowanie wyjątków, zegar i adapter tożsamości. Twoim zadaniem jest naprawa dwóch funkcji oraz wyjaśnienie kompletnego kontraktu HTTP.

Katalog ma cztery warsztaty. Id 30 jest archiwalne, a 10, 20 i 40 mają identyczny czas rozpoczęcia: 1 czerwca 2035 o 10:00 UTC. Warsztat 40 ma jedno zajęte miejsce użytkownika 999. Nowa rezerwacja dostaje id 2. Adapter zwraca User(101). Dane demonstracyjne są celowo małe, lecz zawierają remis sortowania, brak miejsca i przypadek po rozpoczęciu.

> **Granica ćwiczenia**
>
> Tożsamość jest stała, niezweryfikowana. Nie wystawiaj tego serwera jako rzeczywistej usługi dla klientów. Store żyje w pamięci procesu, restart usuwa rezerwacje. Sekwencyjne testy reguł nie dowodzą bezpieczeństwa wielu procesów ani trwałości operacji.

Tabela kontraktu

Wywołanie lub warunek	Wymagany wynik
GET /workshops	200, items/total/limit/offset/next_offset.
limit=2, offset=0	Id 30 i 10; total 4; next_offset 2.
limit=2, offset=2	Id 20 i 40; next_offset null.
Offset za końcem	200, puste items, total 4 i next_offset null.
GET /workshops/stats	200, total 4; trasa statyczna przed dynamiczną.
GET /workshops/10	Publiczne pola, dostępne 2 miejsca, bez internal_note.
POST dla 10, seats=1	201, nowa rezerwacja i odczytywalne Location.
Druga rezerwacja użytkownika	409, detail: already_reserved.
Pełny warsztat	409, detail: workshop_full.
Dokładnie początek lub później	409, detail: workshop_started.
Nieistniejący warsztat	404, detail: workshop_not_found.
Niepoprawne path/query/body	422, detail jako lista błędów walidacji.
GET własnej rezerwacji	200; obca lub brakująca daje 404.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Limit ma zakres 1–50 i domyślną wartość 20. Offset ma zakres 0–10000 i domyślnie 0. Body jest wymagane: dokładnie seats=1 jako int. Nie dopuszczamy dodatkowego user_id. Walidację tych danych dostarcza adapter. Catalog otrzymuje już prawidłowe parametry strony.

Kod startowy i polecenie

Obie funkcje znajdują się w starter.py. Pierwsza ignoruje granice strony:

```
def catalog(store: Store, limit: int, offset: int) -> Page:
    # TODO L09: kolejność, strona i informacja o następnej stronie.
    return Page(
        items=[present(w, store) for w in store.workshops.values()],
        total=len(store.workshops),
        limit=limit,
        offset=offset,
        next_offset=None,
    )
```

Druga sprawdza wyłącznie istnienie warsztatu:

```
def reserve(store: Store, workshop_id: int, user_id: int, now: datetime) ->
↪  Reservation:
    # TODO L09: sprawdź reguły przed jedyną zmianą stanu.
    store.workshop(workshop_id)
    return store.add(workshop_id, user_id)
```

Najpierw zapisz w REPORT-STARTER trzy rozbieżności z tabelą. Potem napraw listowanie: pełny porządek, total przed wycięciem, poprawne items oraz next_offset. Nie zmieniaj kolejności słownika źródłowego. W reserve sprawdź istnienie, czas, duplikat i pojemność, zanim wywołasz jedyną operację zapisującą. Nie zmieniaj sygnatur, testów ani dostarczonej tożsamości, żeby uzyskać zielony wynik.

Podpowiedź 1: sortuj według pary starts_at oraz id. Samo sortowanie po dacie zachowa przypadkową kolejność remisów. Test odwraca kolejność wstawiania tych samych danych.

Podpowiedź 2: koniec strony to offset + limit. Kolejna strona istnieje tylko, gdy ten indeks jest mniejszy niż total. Dla pustego zbioru oraz offset za końcem wpisz null.

Podpowiedź 3: warunek czasu obejmuje równość. Duplikat wymaga zgodności warsztatu i użytkownika. Nie przydzielaj id przed odmową; testy sprawdzają też next_id.

Publiczna projekcja pomija internal_note. Lista katalogu nie zawiera danych uczestników. Taki model otrzymuje klient:

```
class WorkshopView(BaseModel):
    id: int
    title: str
    starts_at: datetime
    capacity: int
    available: int
```

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Rozwiązanie i granice reguł

```
def catalog(store: Store, limit: int, offset: int) -> Page:
    ordered = sorted(store.workshops.values(), key=lambda w: (w.starts_at, w.id))
    total = len(ordered)
    end = offset + limit
    return Page(
        items=[present(w, store) for w in ordered[offset:end]],
        total=total,
        limit=limit,
        offset=offset,
        next_offset=end if end < total else None,
    )
```

Ordered jest nową listą. Strona zawiera jawne publiczne projekcje, a total nadal opisuje cały katalog. Nie filtrujemy automatycznie archiwalnych ani pełnych warsztatów: tabela kontraktu tego nie wymaga. Dwie pierwsze strony mają odpowiednio [30, 10] i [20, 40].

```
def reserve(store: Store, workshop_id: int, user_id: int, now: datetime) ->
↪   Reservation:
    workshop = store.workshop(workshop_id)
    if now >= workshop.starts_at:
        raise Conflict("workshop_started")
    if any(
        r.workshop_id == workshop_id and r.user_id == user_id
        for r in store.reservations.values()
    ):
        raise Conflict("already_reserved")
    if store.used(workshop_id) >= workshop.capacity:
        raise Conflict("workshop_full")
    return store.add(workshop_id, user_id)
```

NotFound powstaje przy pobieraniu warsztatu. Konflikty pojawiają się przed Store.add, więc odrzucone żądanie nie zmienia rezerwacji ani licznika. Kolejność błędów jest świadoma. Jeśli użytkownik już ma miejsce w pełnym warsztacie, dostaje already_reserved.

Kod nie zawiera await ani transakcji. To mały model sekwencyjnych reguł dla aplikacji demonstracyjnej, nie przepis na atomowy zapis do bazy. Późniejsze dodanie zapytania i await pomiędzy odczytem a zapisem zmienia warunki wykonania. Reguły należy wtedy zabezpieczyć w trwałym magazynie, co wykonujemy w L14.

Adapter podnosi HTTPException z 404 lub 409. Po sukcesie ustawia Location na /reservations/{id} i zwraca ReservationView ze statusem 201. Odczyt rezerwacji sprawdza własność względem demonstracyjnego User. Parametr body potwierdza jedno miejsce; nie służy przekazywaniu tożsamości.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Test kontraktu i dokumentacji

Przykład pochodzi bezpośrednio z uruchamianego zestawu testów:

```
def test_create_and_follow_location(client, store):
    before = len(store.reservations)
    response = client.post("/workshops/10/reservations", json={"seats": 1})
    assert response.status_code == 201
    assert response.json() == {"id": 2, "workshop_id": 10, "user_id": 101}
    assert response.headers["location"] == "/reservations/2"
    fetched = client.get(response.headers["location"])
    assert fetched.status_code == 200
    assert fetched.json() == response.json()
    assert len(store.reservations) == before + 1
    assert client.get("/workshops/10").json()["available"] == 1
```

To więcej niż test samego JSON: potwierdza powstanie zasobu, nagłówek, możliwość odczytu, zmianę stanu oraz spadek dostępności. Pozostałe przypadki sprawdzają 422 bez zapisu, 404, konflikty, granicę czasu, brak wycieku pól i obcej rezerwacji, puste strony oraz niezależne aplikacje. Testy OpenAPI sprawdzają wymagane body, schematy, statusy, Location, ograniczenia query i unikalność pięciu operationId.

TestClient wywołuje rzeczywisty kod aplikacji w procesie, ale nie otwiera połączenia TCP z Uvicorn. Osobny probe wykonuje właśnie tę drogę. Nie traktuj obu poziomów jako zamiennych ani nie nazywaj ich próbą obciążeniową.

```
uv run --locked pytest -q --variant starter
uv run --locked ruff check src tests
uv run --locked ruff format --check src tests
uv run --locked mypy
uv run --locked python -m workshop_http.app --variant starter --port 8009
```

Otwórz /docs, wykonaj pierwsze POST, odczytaj Location i ponów żądanie. Po naprawie startera drugie POST powinno dać 409. Nie usuwa to niepewności po utracie odpowiedzi: zwracanie zapisanego wyniku z kluczem idempotencji jest odrębnym zagadnieniem.

Rzeczywista próba i arkusz oceny

```
uv run --locked python -m workshop_http.probe --save output/my-http.json
```

Skrypt sprawdza referencję. Uruchamia własny Uvicorn na wolnym porcie, wykonuje osiem żądań i kończy proces w finally. Dołączony evidence/http.json zawiera faktyczny wynik na Windows i CPythonie 3.14.2: statusy [200, 201, 200, 409, 409, 422, 404, 200]. Odpowiadają kolejno katalogowi, utworzeniu, odczytowi Location, duplikatowi, pełnemu warsztatowi, złemu limitowi, brakowi zasobu i odczytowi OpenAPI. To kontrola startu i kontraktu przez TCP.

Testy mają stały zegar z 2030 roku. Ręczna aplikacja i probe używają rzeczywistego czasu. Po rozpoczęciu danych warsztatów w 2035 roku trzeba przesunąć daty demonstracyjne, aby nadal pokazywać sukces POST. Nie trzeba zmieniać stałego zegara testów.

Ocena: po 2 punkty za stabilne strony, reguły i brak zapisu po odmowie, statusy i Location, granicę tożsamości oraz danych publicznych, zgodność OpenAPI i wyjaśnienie ograniczeń. Łącznie 10 punktów. REPORT-STARTER służy własnym obserwacjom, a REPORT-SOLUTION pokazuje uzasadnienie referencji.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Rozszerzenia: przenieś router do oddzielnego modułu bez zmiany odpowiedzi; opisz zmianę stron po dopisaniu rekordu między żądaniami; zaproponuj kursor z pary starts_at i id. Nie rozszerzaj tego ćwiczenia o własne logowanie ani ORM przed wyjaśnieniem obecnego kontraktu.

6.9 Powtórka przed rozmową

Pytanie kontrolne	Element odpowiedzi
Czy async def oznacza 202?	Nie; status opisuje wynik operacji, a nie składnię funkcji.
Czy int naprawi kolejność tras?	Walidacja parametru nie zastępuje dopasowania trasy.
Czy pusta strona to 404?	Kolekcja istnieje; kontrakt daje 200 z pustymi items.
Czy 409 oznacza idempotencję?	Odmowa duplikatu nie odtwarza pierwotnej odpowiedzi.
Czy responses tworzy błąd?	Dokumentuje go; właściwy status i body wytwarza kod.
Czy sortowanie tworzy migawkę?	Rozstrzyga kolejność, lecz nie zatrzymuje zmian między żądaniami.
Czy demo_user uwierzytelnia?	Nie; zwraca stałą tożsamość do ćwiczenia.

Przejdź przez POST od walidacji body do odczytu Location. Następnie wyjaśnij odmowę bez zmiany stanu oraz ograniczenie pamięci procesu. W rozdziale 7 rozbudujemy kontrakty Pydantic, w tym różnicę między brakiem pola a null w częściowej aktualizacji.

6.10 Źródła i dalsza praca

Zakres zaadaptowano z polskich materiałów FAPI-001–014, FAPI-024, FAPI-051–053 i FAPI-056–059. Połączono opisy podstaw frameworka, parametrów i dokumentacji. Porównanie z Django pozostaje w rozdziale 15. Nowe są tabela kontraktu L09, stabilne strony, rozróżnienie 409 od odtworzenia wyniku, testy zmian stanu oraz rzeczywista próba TCP.

- RFC 9110: semantyka metod, statusów i nagłówków HTTP.
- ASGI 3.0: interfejs serwera i aplikacji.
- FastAPI: parametry ścieżki i kolejność tras.
- FastAPI: jawne ograniczenia liczbowe parametrów.
- FastAPI: APIRouter i grupowanie operacji.
- FastAPI: HTTPException i obsługa błędów.
- FastAPI: modele odpowiedzi i filtrowanie danych.
- FastAPI: deklarowanie statusów operacji.
- FastAPI: dodatkowe odpowiedzi w OpenAPI.
- OpenAPI 3.1: struktura opisu interfejsu.
- FastAPI: testowanie aplikacji przez TestClient.

Źródła sprawdzono 19 listopada 2026. Wersje środowiska, znane ostrzeżenie Starlette/AnyIO i zakres wykonanych prób opisuje `VALIDATION-6.md`.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

10

Transakcje i odporność na ponowienia

Anna i Bartek otwierają stronę warsztatu, na który zostało jedno miejsce. Oboje naciskają „Zarezerwuj". Dwie instancje aplikacji sprawdzają dostępność i obie widzą wartość jeden. Każda zapisuje rezerwację i ustawia licznik wolnych miejsc na zero. Baza nie zgłasza błędu, a organizator ma dwóch uczestników na jedno miejsce.

Druga sytuacja jest mniej widowiskowa. Anna rezerwuje miejsce, baza zatwierdza zapis, lecz połączenie z klientem urywa się przed dostarczeniem odpowiedzi. Przeglądarka ponawia żądanie. Aplikacja odpowiada: „Brak miejsc". Anna nie wie, że to właśnie jej rezerwacja zajmuje ostatnie miejsce.

Ten rozdział wyjaśnia oba przypadki na jednej aplikacji. Najpierw ustalimy, co dokładnie ma pozostać prawdą w bazie. Następnie oddzielimy atomowość od izolacji, naprawimy decyzję o przyznaniu miejsca i zapiszemy wynik operacji tak, aby klient mógł go odzyskać po utracie odpowiedzi.

> **Po tym rozdziale**
>
> Potrafisz wyznaczyć granicę transakcji, wyjaśnić zachowanie PostgreSQL przy dwóch równoczesnych zapisach, zaprojektować klucz idempotencji i przygotować test, który odtwarza błąd przed poprawką. Rozdział zawiera 12 pytań, dwa laboratoria oraz rozwiązania do samodzielnego sprawdzenia.

W przykładach używamy PostgreSQL, SQLAlchemy 2 oraz zwykłych funkcji `def` w FastAPI. Operacje bazy są synchroniczne i każda otrzymuje własną sesję. Dzięki temu granica transakcji jest łatwa do prześledzenia. W wersji asynchronicznej pozostają te same reguły, ale trzeba użyć asynchronicznego sterownika i osobnej `AsyncSession` dla każdego równoczesnego zadania. Zmiana składni na `async` nie naprawia wyścigu.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

10.1 Reguły biznesowe i granica transakcji

Q10.1: Co powinna obejmować transakcja rezerwacji? [Mid]

Transakcja powinna obejmować wszystkie zmiany, które razem tworzą jedną poprawną operację biznesową. W naszym przypadku są to przyznanie miejsca i zapis aktywnej rezerwacji. Jeśli zapis rezerwacji się nie powiedzie, liczba wolnych miejsc musi wrócić do poprzedniej wartości.

Przed wyborem mechanizmu blokowania warto zapisać **niezmienniki**: warunki, które mają być spełnione po każdym zatwierdzonym stanie. Warsztat ma pojemność `capacity` i licznik `remaining_seats`. Liczba wolnych miejsc mieści się między zerem a pojemnością. Dodatkowo liczba aktywnych rezerwacji plus liczba wolnych miejsc jest równa pojemności. Jeden użytkownik może mieć jedną aktywną rezerwację danego warsztatu. Po anulowaniu może dokonać nowej.

Warunek dotyczący samego licznika można zapisać jako CHECK. Warunek unikalności aktywnej rezerwacji wyraża częściowy indeks unikalny. Równość obejmująca wiersze dwóch tabel wymaga jednak spójnego protokołu zapisu. Sam CHECK na tabeli warsztatów jej nie zapewni.

Granica transakcji nie powinna obejmować całej obsługi HTTP. Odczytanie dużego pliku, czekanie na odpowiedź zewnętrznej usługi i przygotowanie treści wiadomości zwykle nie wymagają utrzymywania blokady warsztatu. Przed transakcją przygotowujemy dane; w transakcji podejmujemy decyzję na podstawie aktualnego stanu i zapisujemy jej rezultat.

```
CHECK (remaining_seats >= 0 AND remaining_seats <= capacity);

CREATE UNIQUE INDEX uq_active_reservation
ON reservations (user_id, workshop_id)
WHERE status = 'active';
```

Indeks pozwala zachować anulowane rezerwacje jako historię. Nowa rezerwacja otrzymuje nowy identyfikator, a ograniczenie nadal pilnuje, aby aktywna była tylko jedna. Szczegóły indeksów częściowych opisuje dokumentacja PostgreSQL.

> **Jak odpowiedzieć na rozmowie**
>
> „Zmniejszenie dostępności i utworzenie rezerwacji muszą zatwierdzić się razem. Unikalność zabezpieczę w bazie, a przyznanie miejsca wykonam operacją, która rozstrzyga konflikt równoczesnych żądań. Odpowiedź o sukcesie wyślę dopiero po zatwierdzeniu transakcji.”

Q10.2: Czy ACID oznacza, że wyścigi nie mogą wystąpić? [Mid]

Nie. Atomowość oznacza, że transakcja nie pozostawi części własnych zmian. Izolacja określa natomiast, jak jej działania współistnieją z działaniami innych transakcji. Dwie atomowe transakcje mogą wspólnie naruszyć regułę aplikacji, jeżeli obie podejmą decyzję na podstawie stanu, którego aktualności nie zabezpieczyły.

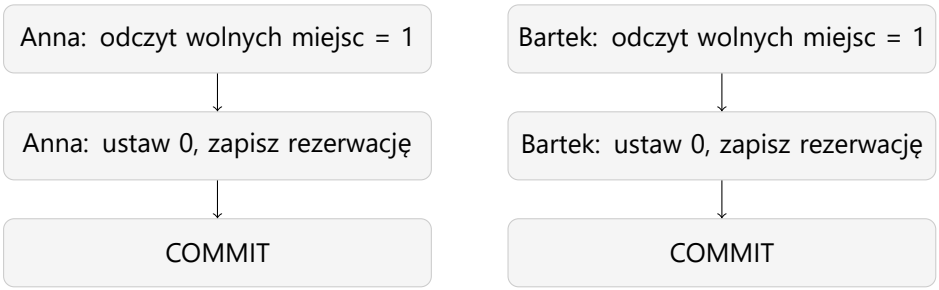
W błędnym wariancie obie transakcje odczytują jeden wolny slot. Następnie każda ustawia licznik na wcześniej obliczoną wartość zero. Oba zapisy spełniają CHECK, a użytkownicy są różni,

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

więc indeks unikalny również nie zgłasza konfliktu. Każda transakcja wykonuje komplet własnych zmian. Błędny jest protokół przyznawania miejsca.

Spójność w ACID nie oznacza, że baza rozumie wszystkie wymagania biznesowe bez ich zapisania. PostgreSQL egzekwuje zadeklarowane ograniczenia. Jeżeli reguła obejmuje sumę danych z kilku miejsc, aplikacja musi dobrać transakcję i mechanizm synchronizacji tak, aby tę regułę utrzymać.

Trwałość również ma określone warunki: ustawienia bazy, zapis na nośniku i sposób replikacji wpływają na zachowanie podczas awarii. W laboratoriach sprawdzamy przede wszystkim atomowość i współbieżność. Kontener testowy trzyma dane tymczasowo, więc usunięcie go nie jest eksperymentem potwierdzającym trwałość produkcyjnego systemu.



Wynik: zero wolnych miejsc i dwie aktywne rezerwacje przy pojemności jeden. Test L14 pokazuje dokładnie ten stan, bez usuwania ograniczenia CHECK z bazy.

10.2 Izolacja i decyzja o przyznaniu miejsca

Q10.3: Co oznacza Read Committed w PostgreSQL? [Senior]

Przy Read Committed zwykły SELECT widzi dane zatwierdzone przed rozpoczęciem danego polecenia. Kolejny SELECT w tej samej transakcji może zobaczyć nowszy stan. Nie należy utożsamiać transakcji z jednym, niezmiennym obrazem bazy.

Inne znaczenie ma modyfikacja wiersza. Jeżeli konkurencyjna transakcja aktualizuje ten sam wiersz, UPDATE może czekać na jej zakończenie. Po zatwierdzeniu konkurencyjnej zmiany PostgreSQL ponownie ocenia warunek aktualizacji względem nowszej wersji wiersza. To zachowanie wykorzystamy w poleceniu z warunkiem `remaining_seats > 0`. Szczegóły i różnice między poziomami opisuje dokumentacja izolacji PostgreSQL.

Nie oznacza to, że dowolna sekwencja zapytań staje się bezpieczna. Jeżeli aplikacja obliczy wartość zero na podstawie wcześniejszego odczytu i wyśle `SET remaining_seats = 0`, baza nie zgadnie, że intencją było warunkowe przyznanie miejsca.

W naszym projekcie Read Committed jest wystarczający, bo konflikt rozstrzyga konkretna operacja na jednym wierszu warsztatu. Przy regule obejmującej wiele niezależnych wierszy trzeba ponownie przeanalizować problem. Przykładem jest zakaz jednoczesnego dyżuru w kilku miejscach: blokada pojedynczego warsztatu nie chroni automatycznie wszystkich rezerwacji użytkownika.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Poziom	Znaczenie w PostgreSQL
Read Uncommitted	Zachowuje się tak samo jak Read Committed. Nie udostępnia brudnych odczytów.
Read Committed	Zwykły SELECT widzi stan zatwierdzony przed rozpoczęciem danego polecenia. Kolejny SELECT może zobaczyć inny stan.
Repeatable Read	Zapewnia stały obraz dla odczytów transakcji i nie dopuszcza fantomów, lecz nadal pozwala na anomalie serializacji. Konflikt zapisu może wymagać ponowienia.
Serializable	Zatwierdzone transakcje mają efekt zgodny z pewną kolejnością wykonania szeregowego. Aplikacja musi obsługiwać odrzucenie transakcji i ponowienie całości.

Brudny odczyt dotyczy niezatwierdzonej zmiany innej transakcji. Fantom to zmiana zestawu wierszy pasujących do ponowionego zapytania. Serializable nie oznacza, że PostgreSQL uruchamia transakcje po jednej; może wykryć niebezpieczny układ zależności i przerwać jedną z nich.

> **Założenia są częścią rozwiązania**
>
> Testy laboratoriów jawnie ustawiają Read Committed. Przeniesienie kodu do innego poziomu izolacji lub innej bazy wymaga ponownego sprawdzenia zachowania konfliktów i obsługi błędów. Nie wystarczy zachować nazw metod ORM.

Q10.4: Jak atomowa aktualizacja rozwiązuje wyścig o ostatnie miejsce? [Mid]

Łączymy sprawdzenie dostępności i jej zmniejszenie w jednym poleceniu. Zamiast wysyłać wartość obliczoną wcześniej w Pythonie, prosimy bazę: „zmniejsz bieżący licznik o jeden, jeżeli nadal jest dodatni". Polecenie zwraca identyfikator zmienionego wiersza.

Pierwsza transakcja zmienia jeden na zero. Druga próbuje zmodyfikować ten sam wiersz. Po zakończeniu pierwszej nie spełnia już warunku dodatniej dostępności i nie otrzymuje wiersza z `RETURNING`. Brak wyniku jest normalnym wynikiem decyzji biznesowej.

Jeżeli miejsce udało się przyznać, tworzymy rezerwację w tej samej transakcji. Nie zatwierdzamy wcześniej licznika. Gdy indeks unikalny odrzuci drugą aktywną rezerwację tego samego użytkownika, cała operacja musi się wycofać razem z wcześniej zmniejszonym licznikiem.

W przykładzie używamy `ON CONFLICT DO NOTHING` wyłącznie dla konkretnego indeksu aktywnej rezerwacji. Brak identyfikatora oznacza konflikt biznesowy; wyjątek `Rejected` opuszcza blok transakcji. Nie ukrywamy wszystkich błędów bazy pod komunikatem „Brak miejsc". Nieprawidłowa migracja lub zerwane połączenie wymagają innej diagnozy.

```
UPDATE workshops
SET remaining_seats = remaining_seats - 1
WHERE id = :workshop_id
  AND remaining_seats > 0
  AND starts_at > :now
RETURNING id;
```

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Poniższa funkcja pochodzi bezpośrednio z działającego rozwiązania L14. Właścicielem transakcji pozostaje funkcja, która ją wywołuje. Punkty `hook` pozwalają testom zatrzymać wykonanie lub wywołać awarię; w zwykłym uruchomieniu nic nie robią.

```
def reserve_in_transaction(session, user_id, workshop_id, note, now, hook):
    """Wywołujący posiada transakcję. Ta funkcja nigdy sama nie zatwierdza zmian."""
    hook("before_reserve")
    claimed = session.execute(
        update(workshops)
        .where(
            workshops.c.id == workshop_id,
            workshops.c.starts_at > now,
            workshops.c.remaining_seats > 0,
        )
        .values(remaining_seats=workshops.c.remaining_seats - 1)
        .returning(workshops.c.id)
    ).scalar_one_or_none()
    if claimed is None:
        exists = session.scalar(select(workshops.c.id).where(workshops.c.id ==
        ↪  workshop_id))
        if exists is None:
            raise Rejected(404, "Warsztat nie istnieje.")
        raise Rejected(409, "Brak miejsc lub warsztat już się rozpoczął.")
    hook("after_seat_change")
    reservation_id = session.execute(
        insert(reservations)
        .values(
            id=uuid4(),
            user_id=user_id,
            workshop_id=workshop_id,
            status="active",
            note=note,
        )
        .on_conflict_do_nothing(
            index_elements=[reservations.c.user_id, reservations.c.workshop_id],
            index_where=reservations.c.status == "active",
        )
        .returning(reservations.c.id)
    ).scalar_one_or_none()
    if reservation_id is None:
        # Wyjątek opuszcza zewnętrzny begin: cofamy również zmniejszenie licznika.
        raise Rejected(409, "Użytkownik ma już aktywną rezerwację.")
    return Result(
        201,
        {
            "id": str(reservation_id),
            "user_id": user_id,
            "workshop_id": workshop_id,
            "status": "active",
            "note": note,
        },
    )
```

Q10.5: Kiedy wybrać SELECT FOR UPDATE zamiast aktualizacji warunkowej? [Senior]

Aktualizacja warunkowa jest wygodna, gdy decyzję da się wyrazić krótkim warunkiem SQL. Blokada wiersza bywa czytelniejsza, gdy potrzebujemy odczytać kilka pól i wykonać bardziej rozbudowaną decyzję przed zapisem. `SELECT FOR UPDATE` blokuje możliwość konkurencyjnej modyfikacji wybranych wierszy do końca transakcji. Dokładne konflikty trybów blokad opisuje PostgreSQL.

Blokadę trzeba pobrać **przed** odczytem, od którego zależy decyzja. Schemat „odczytaj bez blokady, a później zablokuj i zapisz wcześniej obliczoną wartość" nadal wymaga odświeżenia stanu. Samo pojawienie się `FOR UPDATE` w kodzie nie jest dowodem poprawności.

W obu wariantach trzeba uważać na długość transakcji. Wywołanie usługi płatniczej pod blokadą warsztatu może zatrzymać wszystkie kolejne rezerwacje tego wydarzenia. Nawet poprawna blokada potrafi stać się wąskim gardłem.

Alternatywą jest kontrola optymistyczna, np. aktualizacja z warunkiem zgodności numeru wersji. Brak zmienionego wiersza wymusza ponowne odczytanie danych i podjęcie decyzji. Wybór zależy od częstości konfliktów, czasu operacji i tego, jak kosztowne jest jej powtórzenie. W L14 prosta aktualizacja warunkowa ma mało elementów i dobrze wyraża regułę przyznania jednego miejsca.

```
with Session(engine) as session, session.begin():
    row = session.execute(
        select(workshops)
        .where(workshops.c.id == workshop_id)
        .with_for_update()
    ).mappings().one()
    # Dopiero teraz sprawdź aktualną dostępność i wykonaj zapis.
```

10.3 Błędy i wycofywanie zmian

Q10.6: Czym różnią się flush, commit i rollback w SQLAlchemy? [Mid]

`flush` wysyła oczekujące zmiany obiektów ORM do bazy w obrębie bieżącej transakcji. Może ujawnić błąd ograniczenia albo nadać obiektowi identyfikator. Nie stanowi jednak deklaracji, że operacja została zatwierdzona i zakończy się sukcesem.

`commit` zatwierdza transakcję. `rollback` wycofuje jej niezatwierdzone zmiany. Jeśli zapisaliśmy rekord rezerwacji, a potem wystąpił błąd, rollback ma objąć również zmianę dostępności miejsca. Pozostawienie zatwierdzenia w dwóch różnych funkcjach utrudnia utrzymanie tej zasady.

W laboratorium korzystamy z kontekstu `session.begin()`. Normalne wyjście zatwierdza transakcję, a wyjątek powoduje jej wycofanie. Zewnętrzny kontekst `Session` zamyka sesję. Mechanikę tych kontekstów opisuje SQLAlchemy.

Po błędzie bazodanowym nie kontynuujemy bezrefleksyjnie pracy na uszkodzonej transakcji. Najprościej opuścić jej zakres, wycofać całość i rozpocząć nową operację, jeśli błąd rzeczywiście

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

dopuszcza ponowienie. Samo przechwycenie wyjątku w środku bloku i zwrócenie sukcesu ukrywa niepewność zamiast ją rozwiązać.

```
def reserve(engine, user_id, workshop_id, note, now, hook, key=None):
    with Session(engine) as session, session.begin():
        result = reserve_in_transaction(session, user_id, workshop_id, note, now, hook)
        hook("before_commit")
    return result
```

Sukces HTTP dopiero po commit

Nie odkładaj zatwierdzenia operacji biznesowej do sprzątania zależności po wysłaniu odpowiedzi. Jeśli commit się nie powiedzie, klient nie powinien mieć już w ręku odpowiedzi 201. W naszym adapterze HTTP `after_commit` jest wykonywane po powrocie z serwisu.

Q10.7: Jak obsłużyć deadlock i błąd serializacji? [Senior]

Deadlock występuje, gdy transakcje czekają na zasoby trzymane przez siebie nawzajem. Przykładowo pierwsza blokuje warsztat A i czeka na B, a druga blokuje B i czeka na A. PostgreSQL wykrywa taki cykl i przerywa jedną z transakcji. Podniesienie limitu oczekiwania nie usuwa przyczyny.

Pierwsza obrona to spójna kolejność pobierania blokad. Jeżeli operacja obejmuje kilka warsztatów, można blokować je w kolejności identyfikatorów. W rezerwacji i anulowaniu naszego projektu zachowujemy kolejność: wiersz warsztatu, następnie rezerwacja. Transakcje powinny też być krótkie.

Przy błędzie serializacji lub deadlocku może być potrzebne ponowienie **całej transakcji**: razem z odczytami i decyzją biznesową. Nie powtarzamy jedynie ostatniego zapytania na podstawie starych danych. Kody `40001` i `40P01` opisują odpowiednio błąd serializacji i deadlock; ich znaczenie można sprawdzić w katalogu SQLSTATE.

Ponawianie musi mieć limit prób oraz budżet czasu. Po każdej nieudanej próbie kończymy transakcję, a przed następną można zastosować krótki, losowo zróżnicowany odstęp. Nie czekamy z przerwą wewnątrz transakcji trzymającej blokady.

Nie każdy błąd należy powtarzać. Naruszenie reguły „jedna aktywna rezerwacja” jest konfliktem biznesowym. Przy utracie połączenia podczas commit wynik może być nieznany; wtedy potrzebujemy identyfikatora operacji i odtworzenia jej rezultatu. Bezwarunkowe ponowienie wszystkich wyjątków może wykonać efekt drugi raz.

Zakres laboratoriów

Laboratoria ustawiają ograniczony czas oczekiwania na blokadę i polecenie. Nie zawierają automatycznego ponawiania wszystkich błędów ani mapowania każdego SQLSTATE na odpowiedź API. Ćwiczenia skupiają się na regułach rezerwacji i odzyskiwaniu wyniku. Rozbudowa obsługi awarii wymaga odrębnego kontraktu oraz testów.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

10.4 Idempotencja i utracona odpowiedź

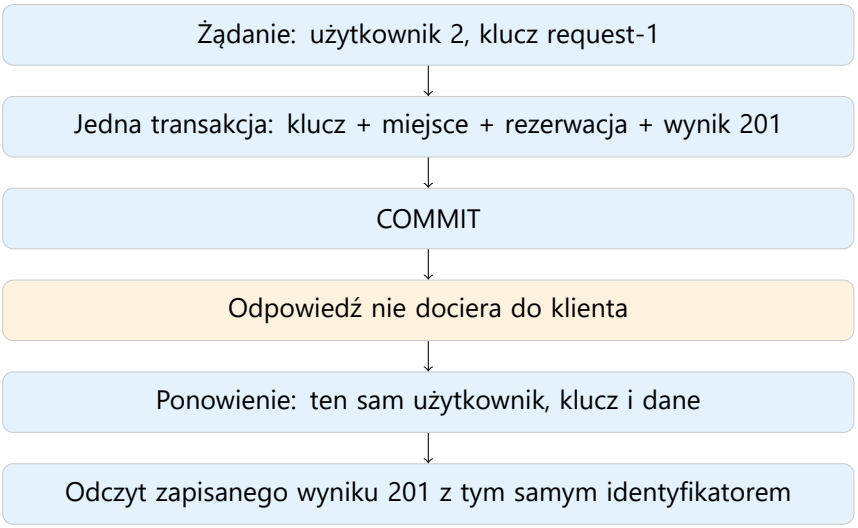
Q10.8: Dlaczego poprawna transakcja nie wystarcza po utracie odpowiedzi? [Mid]

Transakcja zabezpiecza zapis danych, ale nie łączy go atomowo z dostarczeniem odpowiedzi przez sieć. Między zatwierdzeniem w bazie a odczytaniem odpowiedzi przez klienta pozostaje okno niepewności. Proces może się zakończyć, połączenie może zostać przerwane albo pośrednik może przekroczyć własny limit czasu.

Klient widzi timeout i nie potrafi odróżnić dwóch przypadków: „operacja nie została wykonana" oraz „operacja się udała, ale odpowiedź zaginęła". Zapytanie o aktualną dostępność także nie daje pewnej odpowiedzi. Ostatnie miejsce mógł zająć inny uczestnik.

W L14 ponowienie tej samej rezerwacji nie tworzy drugiego aktywnego rekordu, ponieważ chronią nas reguły biznesowe. Może jednak zakończyć się konfliktem 409. To nie jest odtworzenie wyniku pierwszej operacji. Klient potrzebuje sposobu powiązania ponowienia z konkretnym żądaniem.

W L15 klient nadaje operacji `Idempotency-Key`. Serwer zapisuje ten klucz, odcisk danych żądania oraz wynik HTTP razem z rezerwacją. Jeżeli odpowiedź zaginie, ponowienie z tym samym kluczem odczytuje wynik pierwszej operacji. Nie uruchamia ponownie logiki przyznawania miejsca.



Q10.9: Jak zaprojektować klucz idempotencji i porównywać żądania? [Senior]

Klucz nie powinien być globalny dla wszystkich klientów. W laboratorium zakres wyznaczają użytkownik, rodzaj operacji i wartość klucza. Dzięki temu Anna i Bartek mogą użyć tego samego tekstu klucza bez otrzymania cudzej odpowiedzi. W produkcji użytkownik musi pochodzić ze zweryfikowanego uwierzytelnienia.

Przechowujemy również odcisk danych mających wpływ na wynik. U nas są to identyfikator warsztatu i notatka. Najpierw walidujemy dane i uzupełniamy wartości domyślne, a potem tworzymy stabilny zapis JSON. Brak pola notatki oraz jawny pusty napis oznaczają to samo żądanie. Kolejność kluczy w JSON nie może zmieniać odcisku.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Jeżeli ważny klucz wraca z innymi danymi, zwracamy 409. Milczące odtworzenie starej odpowiedzi ukryłoby pomyłkę klienta. Nie wolno też nadpisać wyniku nowym payloadem i potraktować go jako tej samej operacji.

Kontrakt laboratorium przechowuje wyłącznie zakończone sukcesem wyniki 201. Konflikt lub błąd przed commit wycofuje również utworzenie klucza. To świadoma polityka: odrzucone żądanie może zostać ponowione później, gdy pojawi się wolne miejsce. Inne API może utrwalać również wybrane odpowiedzi błędne, ale wtedy musi określić dokładnie ich zakres.

```
def fingerprint(workshop_id, note):
    canonical = json.dumps(
        {"workshop_id": workshop_id, "note": note},
        sort_keys=True,
        ensure_ascii=False,
        separators=(",", ":"),
    )
    return hashlib.sha256(canonical.encode("utf-8")).hexdigest()
```

Czego nie dodawać do odcisku

Losowego identyfikatora żądania, chwili ponowienia ani nowego tokenu dostępowego. Te wartości mogą się zmienić, mimo że klient ponawia tę samą operację. Zmiana semantyki endpointu wymaga natomiast przemyślenia wersji kontraktu i zakresu klucza.

Q10.10: Jak obsłużyć dwa równoczesne żądania z tym samym kluczem? [Senior]

Najpierw próbujemy utworzyć rekord klucza przez `INSERT ON CONFLICT DO NOTHING`. Klucz złożony ma ograniczenie unikalności w PostgreSQL. Wygrane żądanie wykonuje operację i zapisuje wynik przed zatwierdzeniem. Drugie żądanie napotyka konflikt z tym samym kluczem i czeka na rozstrzygnięcie konkurencyjnej transakcji.

Jeżeli pierwsza transakcja zostanie wycofana, drugie żądanie może utworzyć klucz i wykonać operację. Jeżeli pierwsza się zatwierdzi, drugie odczytuje gotowy wynik. Nie zatwierdzamy osobno pustego rekordu „w trakcie", który pozostałby po awarii procesu. Zachowanie `ON CONFLICT` opisuje dokumentacja INSERT.

Istotny szczegół to osobny odczyt po próbie wstawienia. Przy Read Committed polecenie INSERT może rozpoznać konflikt z wierszem, który nie był widoczny w początkowym obrazie tego polecenia. Kolejny SELECT otrzymuje nowy obraz zatwierdzonych danych. Nie zastępujemy tego bez sprawdzenia jednym sprytnym zapytaniem CTE łączącym INSERT i SELECT.

Rekord klucza blokujemy także przy odczycie, ponieważ obsługujemy przejęcie klucza po upływie ważności. Pierwsze żądanie po wygaśnięciu aktualizuje odcisk i wynik; kolejne muszą zobaczyć już ten nowy stan. Spójna kolejność blokad to klucz idempotencji, warsztat, rezerwacja.

Pełna funkcja rozwiązania L15 korzysta z wcześniej pokazanej `reserve_in_transaction`. Wspólna funkcja nie wykonuje commit, więc rekord klucza i operacja biznesowa pozostają w tej samej transakcji. Stałe to `OPERATION = "reserve-workshop-v1"` oraz ważność 24 godziny.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

```
def reserve(engine, user_id, workshop_id, note, now, hook, key=None):
    if not key:
        raise Rejected(422, "Podaj Idempotency-Key.")
    digest = fingerprint(workshop_id, note)
    scope = (
        idempotency.c.user_id == user_id,
        idempotency.c.operation == OPERATION,
        idempotency.c.key == key,
    )
    with Session(engine) as session, session.begin():
        hook("before_claim")
        owned = session.execute(
            insert(idempotency)
            .values(
                user_id=user_id,
                operation=OPERATION,
                key=key,
                fingerprint=digest,
                expires_at=now + TTL,
            )
            .on_conflict_do_nothing(
                index_elements=[
                    idempotency.c.user_id,
                    idempotency.c.operation,
                    idempotency.c.key,
                ]
            )
            .returning(idempotency.c.key)
        ).scalar_one_or_none()

        if owned is None:
            # INSERT czeka na konkurencyjną transakcję. Osobny SELECT w Read Committed
            # widzi jej zatwierdzony wynik. Nie łączymy obu operacji w jeden snapshot
            ↪ CTE.
            previous = (
                session.execute(select(idempotency).where(*scope).with_for_update())
                .mappings()
                .one()
            )
            if previous["expires_at"] > now:
                if previous["fingerprint"] != digest:
                    raise Rejected(409, "Klucz został użyty dla innego żądania.")
                return Result(previous["http_status"], previous["body"])
            # Blokada wiersza serializuje również przejęcie wygasłego klucza.
            session.execute(
                update(idempotency)
                .where(*scope)
                .values(
                    fingerprint=digest,
                    expires_at=now + TTL,
                    http_status=None,
                    body=None,
                )
            )

        hook("after_claim")
        result = reserve_in_transaction(session, user_id, workshop_id, note, now, hook)
        session.execute(
```

```
            update(idempotency)
            .where(*scope)
            .values(
                http_status=result.status,
                body=result.body,
            )
        )
        hook("before_commit")
    # Sukces HTTP może zostać wysłany dopiero po wyjściu z session.begin().
    return result
```

10.5 Anulowanie i czas życia wyniku

Q10.11: Co zwrócić po anulowaniu rezerwacji lub wygaśnięciu klucza? [Senior]

Odtworzenie wyniku operacji i odczyt bieżącego zasobu odpowiadają na różne pytania. Pierwsze mówi: „jak zakończyło się to konkretne żądanie POST?". Drugie mówi: „jaki jest stan rezerwacji teraz?".

Jeżeli Anna zarezerwowała miejsce, a potem anulowała rezerwację, ponowienie pierwotnego POST z nadal ważnym kluczem zwraca historyczne 201 i pierwotne ciało odpowiedzi. Pole status może mieć w nim wartość `active`, chociaż GET pokazuje już `cancelled`. Nie tworzymy kolejnej rezerwacji pod pretekstem odtworzenia wyniku. Ten kontrakt trzeba jasno opisać klientowi.

W laboratorium ważność klucza wynosi 24 godziny. Gdy termin jest równy bieżącej chwili lub już minął, żądanie jest nową operacją. Nadal obowiązują reguły biznesowe: jeśli stara rezerwacja jest aktywna, nie można utworzyć kolejnej aktywnej rezerwacji tego samego użytkownika. Jeżeli została anulowana, nowa rezerwacja może otrzymać nowe id.

Czas żądania pobieramy na wejściu do adaptera HTTP; od niego liczymy ważność nowego klucza i względem niego oceniamy termin istniejącego. Czas w testach dostarcza kontrolowany zegar. Sprawdzamy chwilę sekundę przed terminem i dokładnie na granicy ważności. Nie czekamy dobę ani nie opieramy testu na ustawieniach strefy komputera. Projekt produkcyjny powinien również ustalić źródło czasu, retencję oraz zachowanie procesu czyszczącego stare klucze.

Przypadek	Wynik w L15
Ważny klucz i te same dane	Pierwotny status 201 i pierwotne ciało odpowiedzi.
Ważny klucz i inne dane	409, bez dodatkowej rezerwacji.
Ten sam klucz innego użytkownika	Niezależna operacja w innym zakresie.
Klucz wygasł	Nowa operacja, ponowna ocena reguł biznesowych.
Błąd przed commit	Brak trwałego wyniku; klucz można ponowić.
Utrata odpowiedzi po commit	Wynik czeka w bazie na ponowienie.

Samo anulowanie również musi być odporne na powtórzenie. Zwiększamy dostępność tylko wtedy, gdy UPDATE rzeczywiście zmienił stan z `active` na `cancelled`. Drugie anulowanie zwraca

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

204, lecz nie zwalnia kolejnego miejsca. Kontrola własności rezerwacji poprzedza modyfikację. Przykład adaptera znajduje się w `workshop_lab/api.py`; test obejmuje także dwa równoczesne anulowania.

10.6 Granica systemu i testowanie

Q10.12: Jak sprawdzić awarie i co zrobić z powiadomieniem poza bazą? [Senior]

Test ma wykazać zachowanie, które stanowi problem biznesowy. Dla L14 nie wystarczy wywołać endpoint dwa razy kolejno. Zatrzymujemy oba wykonania po odczycie dostępności w wersji błędnej, aby oba kontynuowały na podstawie wartości jeden. Każde żądanie używa oddzielnej sesji i transakcji PostgreSQL. Po zakończeniu sprawdzamy odpowiedzi HTTP oraz stan dostępny przez API.

Dla L15 wywołujemy kontrolowaną awarię po powrocie z serwisu transakcyjnego, zanim adapter odda odpowiedź. W tym momencie niezależny odczyt widzi już rezerwację. Następnie ponawiamy identyczne żądanie. To model utraty odpowiedzi, a nie test prawdziwego zerwania kabla. Pozwala sprawdzić najważniejszą granicę: commit nastąpił, klient nie poznał wyniku.

Inne punkty awarii znajdują się po utworzeniu klucza, po zmniejszeniu licznika i przed commit. Każdy z nich musi zostawić bazę bez częściowej operacji. Testy nie korzystają z jednej zewnętrznej transakcji obejmującej wszystkie żądania, bo ukryłaby ona rzeczywistą widoczność commitów.

Powiadomienie e-mail jest osobnym efektem. Wysłanie go przed commit może poinformować o rezerwacji, która zostanie wycofana. Wysłanie po commit bez trwałego zlecenia może zostać utracone po awarii procesu. Wzorzec outbox zapisuje zlecenie w tej samej transakcji co rezerwację, a osobny wykonawca dostarcza je później.

Odbiorca musi uwzględniać powtórne dostarczenie. Wykonawca może zakończyć pracę po wysłaniu wiadomości, ale przed zapisaniem potwierdzenia. Idempotencja HTTP i deduplikacja zdarzenia rozwiązują różne granice. Nie obiecujemy pojedynczego efektu w dowolnej zewnętrznej usłudze wyłącznie dlatego, że użyliśmy transakcji PostgreSQL.

> **Dalszy etap projektu**
> Outbox i lokalny odbiorca powiadomień są tematem L21. Laboratoria L14 i L15 nie wysyłają wiadomości. Ich wynik kończy się na zatwierdzonym stanie bazy i odpowiedzi API.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

10.7 L14: ostatnie miejsce

10.7.1 Cel i przygotowanie

Przeznacz 60–90 minut. Uruchom PostgreSQL oraz środowisko opisane w `practice/README.md`. Wariant startowy znajduje się w `lab14/starter.py`, a rozwiązanie w `lab14/solution.py`. Oba korzystają z tego samego schematu, adaptera HTTP i testów kontraktu.

Początkowy warsztat ma jedno miejsce. Użytkownicy 2 i 3 wysyłają rezerwację niezależnie. Testy wstrzykują tożsamość przez lokalny nagłówek `X-Lab-User`; jest to atrapa uwierzytelnienia do ćwiczeń, nie mechanizm dostępu do aplikacji publicznej.

```
uv run pytest tests/test_l14.py --variant starter -q
uv run pytest tests/test_demonstrations.py -q
```

Pierwsze polecenie uruchamia wymagania, których kod startowy nie spełnia. Drugie uruchamia demonstracje: ich sukces oznacza, że udało się odtworzyć błąd, nie że kod startowy stał się poprawny.

10.7.2 Polecenie

1. Odtwórz dwie odpowiedzi 201 dla jednego miejsca. Zapisz stan licznika i liczbę aktywnych rezerwacji.
2. Wskaż, dlaczego CHECK i unikalność aktywnej rezerwacji nie zatrzymały błędu.
3. Napraw funkcję w pliku startowym. Wybierz aktualizację warunkową albo blokadę wiersza; nie zmieniaj oczekiwanych wyników testu.
4. Zachowaj jedną transakcję obejmującą licznik i rezerwację.
5. Sprawdź, czy błąd po zmianie licznika cofa całą operację.
6. Sprawdź powtórną rezerwację tego samego użytkownika oraz anulowanie wykonane dwa razy, również równocześnie.

10.7.3 Wynik kontrolny i podpowiedzi

Przy dwóch różnych użytkownikach oczekujemy odpowiedzi 201 i 409, zera wolnych miejsc oraz jednej aktywnej rezerwacji. Po awarii przed commit oczekujemy jednego wolnego miejsca i braku rezerwacji. Po podwójnym anulowaniu wraca dokładnie jedno miejsce.

Podpowiedź 1. Przenieś warunek dostępności do operacji, która faktycznie ją zmienia. Nie zapisuj liczby obliczonej na podstawie wcześniejszego odczytu.

Podpowiedź 2. Brak wiersza z RETURNING nie jest błędem sterownika. Sprawdź, czy warsztat istnieje, a pozostałe odrzucenia potraktuj jako konflikt biznesowy. Gdy nie uda się utworzyć rezerwacji po przyznaniu miejsca, wyjątek musi opuścić blok transakcji.

Punkt testowy `after_availability_read` usuń razem z błędnym odczytem. Nie umieszczaj bariery oczekującej na drugie żądanie za blokadą wiersza, która zatrzymuje to żądanie. Zachowaj punkt `before_reserve` przed blokadą oraz punkty awarii po zmianie licznika i przed commit.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

10.7.4 Omówienie rozwiązania

Rozwiązanie wcześniej pokazane w Q10.4 aktualizuje bieżącą wartość w bazie. Zwrócony identyfikator stanowi dowód przyznania miejsca w tej transakcji. Funkcja pomocnicza nie zatwierdza danych samodzielnie. Dzięki temu kod wywołujący może dołączyć inne zapisy należące do tej samej operacji. Właśnie tę własność wykorzysta L15.

Oceń rozwiązanie w czterech obszarach: zgodność odpowiedzi HTTP, utrzymanie niezmienników, zachowanie po błędzie oraz wyjaśnienie współbieżności. Lokalny mutex w procesie API nie jest wystarczającym rozwiązaniem, jeśli aplikacja ma kilka procesów lub replik.

10.8 L15: odzyskaj wynik po utracie odpowiedzi

10.8.1 Cel i przygotowanie

Przeznacz 60–90 minut. Kod startowy w `lab15/starter.py` korzysta już z bezpiecznego przyznawania miejsca z L14, ale ignoruje klucz idempotencji. Nie musisz kończyć własnej wersji L14, aby uruchomić L15: punkt startowy odwołuje się do dołączonego rozwiązania referencyjnego.

```
uv run pytest tests/test_l15.py --variant starter -q
uv run pytest tests/test_l15.py --variant solution -q
```

W pierwszym teście zapis się uda, lecz kontrolowany wyjątek przerwie dostarczenie odpowiedzi. Wariant startowy przy ponowieniu zwróci 409. Wariant poprawny odtworzy pierwotne 201 i ten sam identyfikator rezerwacji.

10.8.2 Polecenie

1. Ustal zakres klucza: użytkownik, operacja i wartość nagłówka.
2. Oblicz odcisk zwalidowanych danych, uwzględniając warsztat i notatkę.
3. Przejmij klucz w bazie z użyciem ograniczenia unikalności.
4. Zapisz klucz, rezerwację i pełny wynik HTTP w jednej transakcji.
5. Odtwarzaj wynik ważnego klucza; zmieniony payload odrzucaj przez 409.
6. Obsłuż ważność 24 godziny i ponowne użycie wygasłego klucza.
7. Sprawdź równoczesne ponowienia, awarie przed commit, anulowanie zasobu oraz identyczny klucz należący do innego użytkownika.

10.8.3 Wynik kontrolny i podpowiedzi

Po utracie odpowiedzi niezależny GET pokazuje jedną rezerwację. Ponowiony POST zwraca 201 i jej identyfikator. Licznik nie zmienia się drugi raz. Dwa równoczesne żądania z tym samym kluczem zwracają identyczny wynik. Próba użycia tego klucza do innego warsztatu przed upływem ważności kończy się konfliktem.

Podpowiedź 1. Rejestr kluczy w słowniku procesu nie przeżyje restartu ani nie zsynchronizuje dwóch replik. Rejestr należy do tej samej bazy co rezerwacja.

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

Podpowiedź 2. Nie zatwierdzaj rekordu klucza przed operacją biznesową. Gdy pierwsza próba nie dojdzie do commit, druga powinna móc wykonać całość bez ręcznego odblokowywania „zawieszonego" klucza.

10.8.4 Omówienie rozwiązania

Funkcja z Q10.10 rozdziela dwa przypadki: posiadanie nowego klucza oraz odczyt klucza już istniejącego. Dopiero właściciel nowej lub wygasłej operacji wywołuje logikę rezerwacji. Gotowy wynik zapisuje przed wyjściem z zakresu transakcji.

Zapisujemy status i ciało odpowiedzi. Ten kontrakt nie obiecuje identyczności bajtów transportu ani odtworzenia wszystkich dynamicznych nagłówków HTTP. Gdyby endpoint zwracał istotny nagłówek Location, należałoby określić również sposób jego odtwarzania.

Wersja ćwiczeniowa przejmuje wygasłe klucze przy żądaniu i nie ma osobnego procesu usuwania rekordów. Dodanie takiego procesu wymaga testu wyścigu między usunięciem a odczytem klucza. Nie jest to kosmetyczna optymalizacja: zmienia założenia algorytmu.

10.9 Powtórka przed rozmową

- Potrafię zapisać regułę biznesową przed wyborem poziomu izolacji.
- Odróżniam atomowość transakcji od ochrony przed konfliktem decyzji.
- Wiem, dlaczego odczyt i późniejszy zapis starej wartości są niebezpieczne.
- Potrafię wskazać moment commit i zachowanie po błędzie zapisu.
- Nie mylę braku odpowiedzi z pewnością, że operacja się nie wykonała.
- Rozróżniam odtworzenie wyniku POST od bieżącego stanu zasobu.
- Umiem wyjaśnić zakres klucza, odcisk payloadu, ważność i reguły ponowienia.
- Potrafię pokazać test, który ujawnia błąd przed wprowadzeniem poprawki.

> **Krótka odpowiedź końcowa**
>
> „Miejsce przyznaję atomową operacją w bazie, a licznik i rezerwację zatwierdzam razem. Klucz idempotencji, dane żądania i wynik HTTP zapisuję w tej samej transakcji. Dzięki temu ponowienie może odzyskać rezultat po utracie odpowiedzi. Osobno rozwiązuję trwałość i deduplikację powiadomienia wysyłanego poza bazę."

DARMOWA PRÓBKA · Numeracja stron z pełnej książki · flipcards.pl

FLIPCARDS.PL / BACKEND PYTHON

Ćwicz cały backend

PEŁNA KSIĄŻKA I MATERIAŁY PRAKTYCZNE

Od podstaw Pythona do obrony decyzji na rozmowie

286 stron · 16 rozdziałów · 194 pytania

Python, współbieżność, FastAPI, Pydantic, PostgreSQL i SQLAlchemy, bezpieczeństwo, testy, integracje, wydajność oraz Django.

24 laboratoria i 3 rozmowy próbne

Osobne startery, rozwiązania i testy. W zbiorczym ZIP znajdziesz również instrukcje, zależności, migracje, dane przykładowe oraz arkusze pracy.

Powtórka z konkretnym celem

Dodatki A–G, tabele pomocnicze, indeks pytań z numerami stron i plany powtórki na 7, 14 i 30 dni.

Najpierw odpowiedz własnymi słowami. Potem uruchom test i wyjaśnij, dlaczego rozwiązanie zachowuje regułę również w trudnym przypadku.

Sprawdź ofertę ebooków na flipcards.pl

flipcards.pl/pages/ebooki-pdf

To koniec próbki. Pominięte rozdziały i dodatki są częścią pełnego wydania.