Implementacja protokołu OAuth2

z wykorzystaniem platformy Java oraz Android

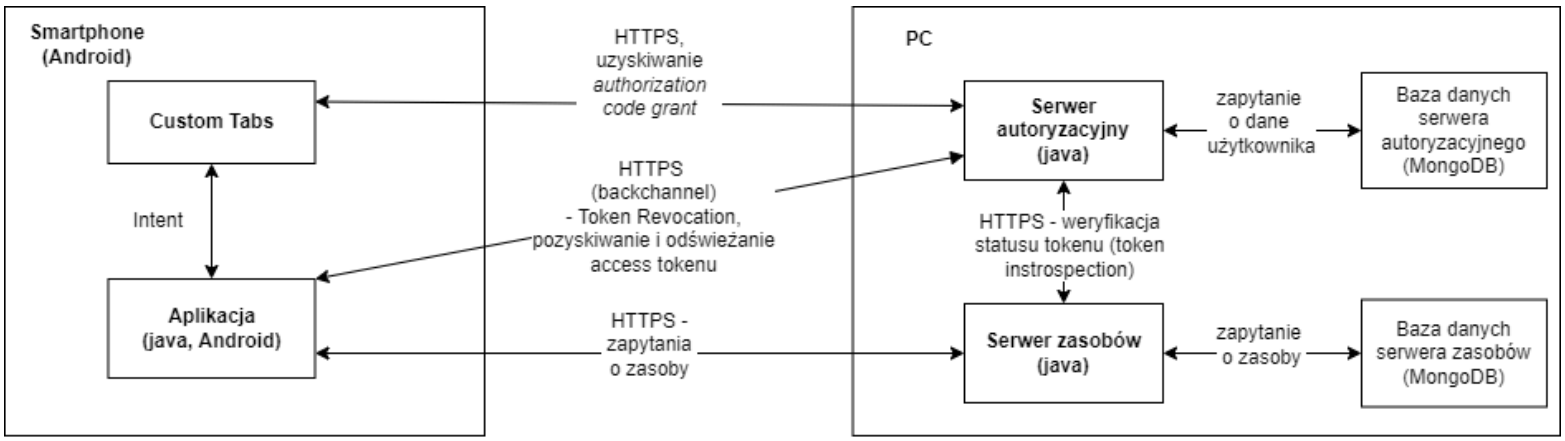
Gabriel Cyganek, Kacper Kafara, Kacper Ślusarczyk, Marcin Pałczak

Wprowadzenie

Celem projektu jest implementacja protokołu OAuth2 z wykorzystaniem technologii Java oraz stworzenie trzech aplikacji klienckich korzystających z funkcji *SingleSignOn* (na platformie Android) pozwalających na demonstrację działania zaimplementowanego protokołu.

Implementacja protokołu obejmuje stworzenie biblioteki <nazwa> pozwalającej na autoryzację za pomocą protokołu OAuth2 zarówno po stronie aplikacji klienckiej jak i serwera zasobów / autoryzacyjnego.

Architektura fizyczna



Architektura fizyczna - z perspektywy pojedynczej aplikacji klienckiej

W skład zaimplementowanego systemu wchodzą:

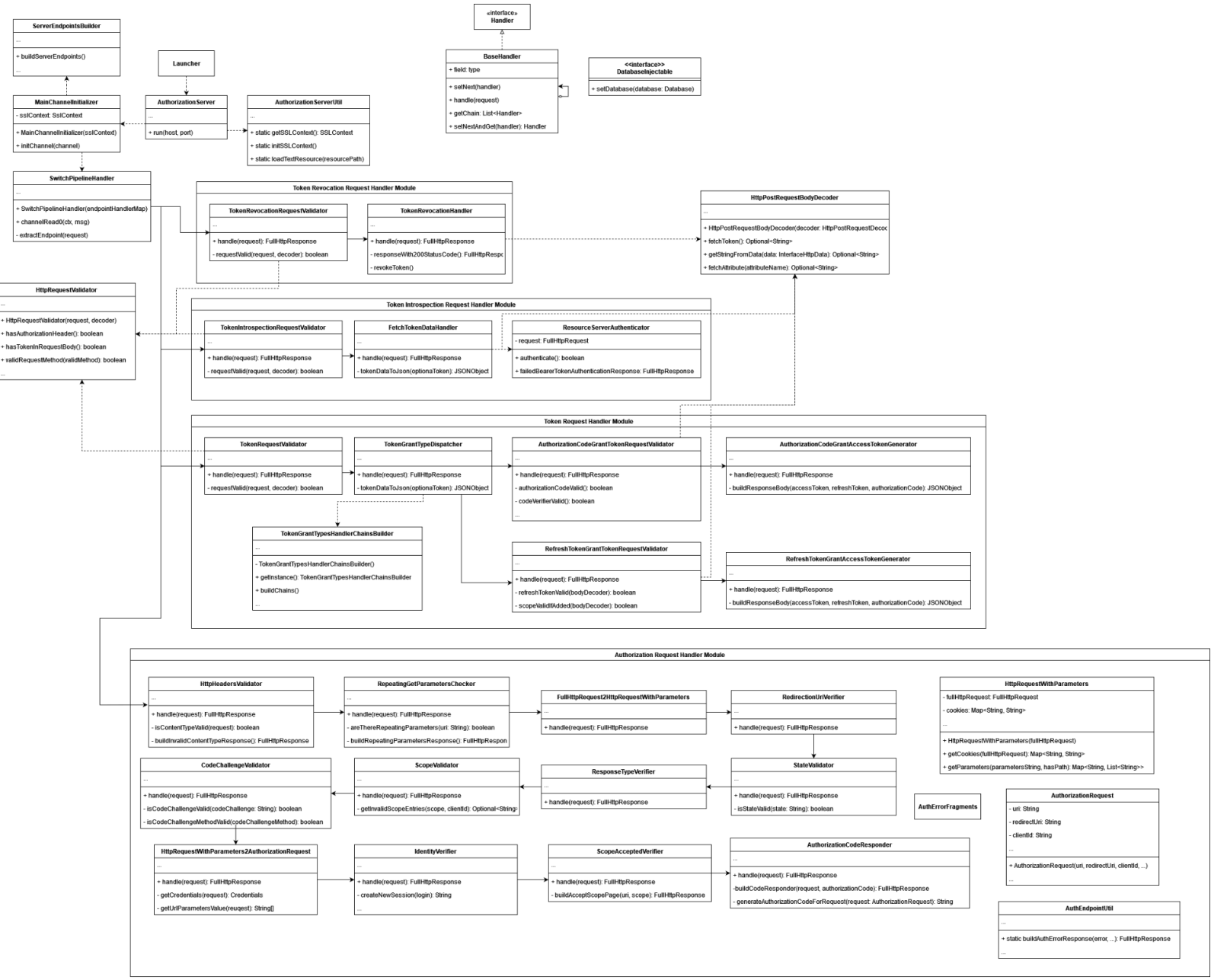
1. Trzy aplikacje klienckie - działające na urządzeniu mobilnym (platforma Android). Wykorzystują funkcjonalność SingleSignOn. Wykorzystują połączenia HTTPS do komunikacji z serwerem autoryzacyjnym (gdzie korzystamy z Custom Tabs do początkowej autoryzacji użytkownika) i serwerem zasobów, aby uzyskać pożądane zasoby.
2. Serwer autoryzacyjny - proces działający na zewnętrznej jednostce PC (technologia Java). Korzysta z własnej bazy danych MongoDB, funkcjonującej na tej samej jednostce PC, do przechowywania danych aplikacji klienckich oraz danych użytkowników. Do komunikacji z serwerem zasobów / aplikacją kliencką wykorzystuje połączenia HTTPS.
3. Serwer zasobów - proces działający na zewnętrznej jednostce PC (technologia Java). Korzysta z własnej bazy danych MongoDB, funkcjonującej na tej samej jednostce PC, do przechowywania zasobów użytkowników. Do komunikacji z serwerem autoryzacyjnym / aplikacją kliencką wykorzystuje połączenia HTTPS.

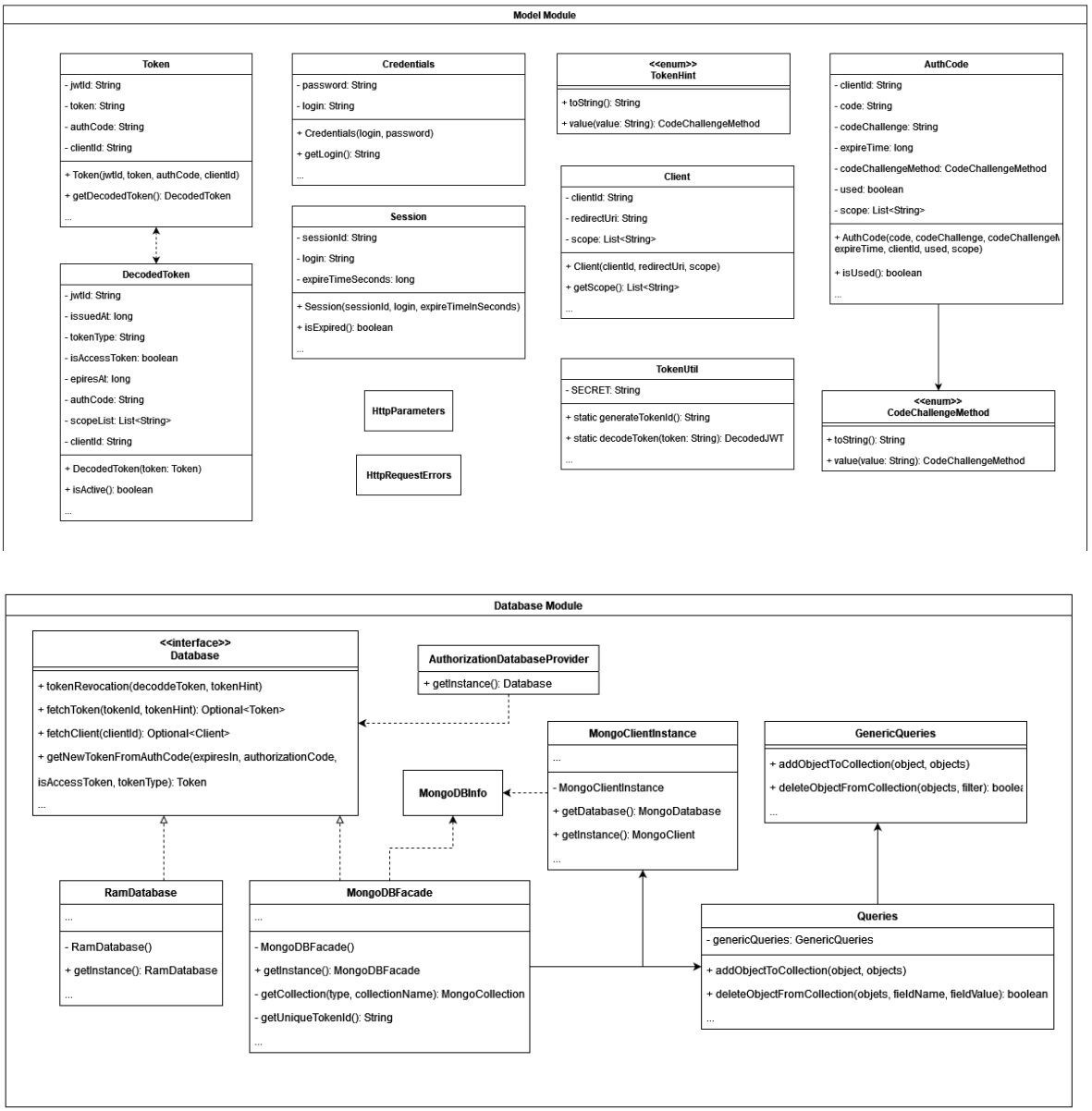
Architektura logiczna

Architektura logiczna

Diagramy klas

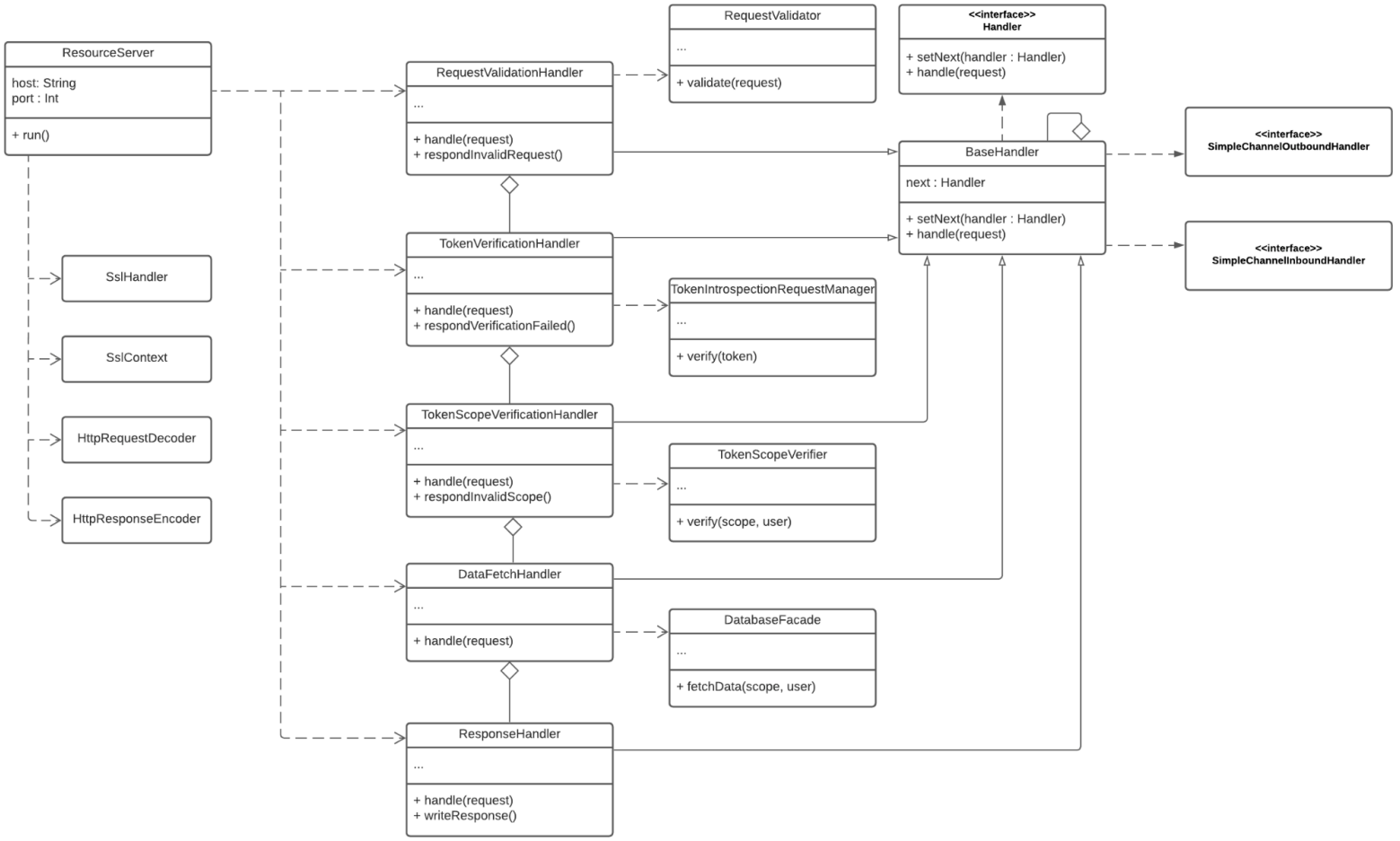
Serwer autoryzacyjny



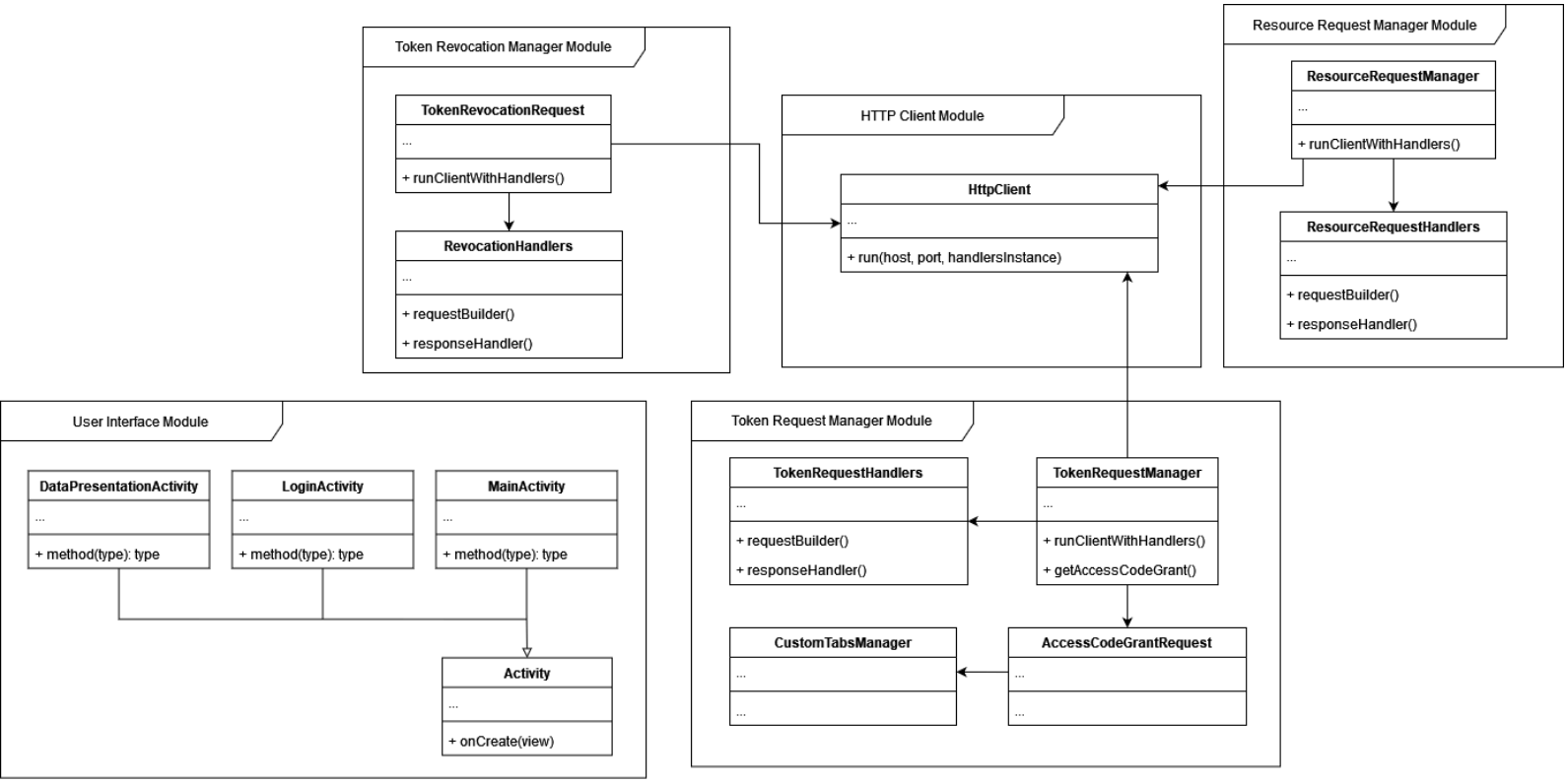


Klasy modułów obsługujących endpointy serwera autoryzacyjnego wykorzystują klasy modułu modelu oraz modułu bazy danych, jednak pominięto ich powiązania na rysunkach dla lepszej przejrzystości diagramów. Dodatkowo klasy obsługujące endpointy i zawierające metodę *handle* rozszerzają klasę *BaseHandler* i są częścią wzorca *Pipeline* używanego do przetwarzania zapytań HTTPS otrzymywanych przez serwer.

Serwer zasobów



Aplikacja kliencka



Authorization - PKCE Flow

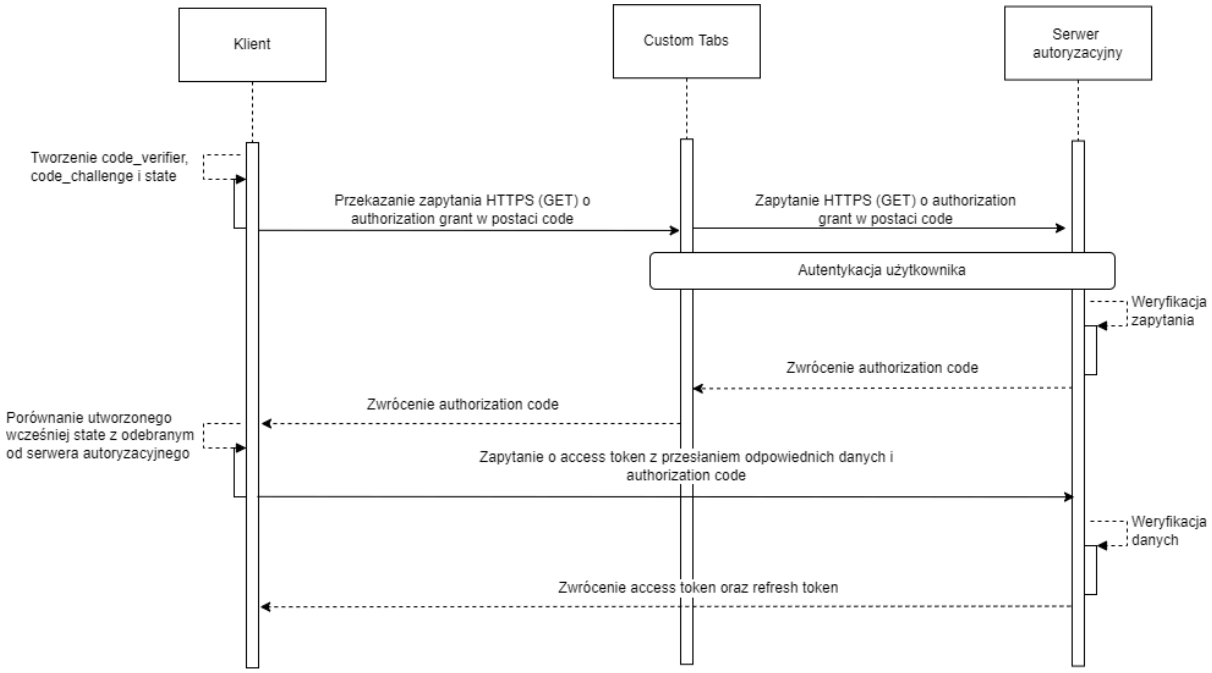


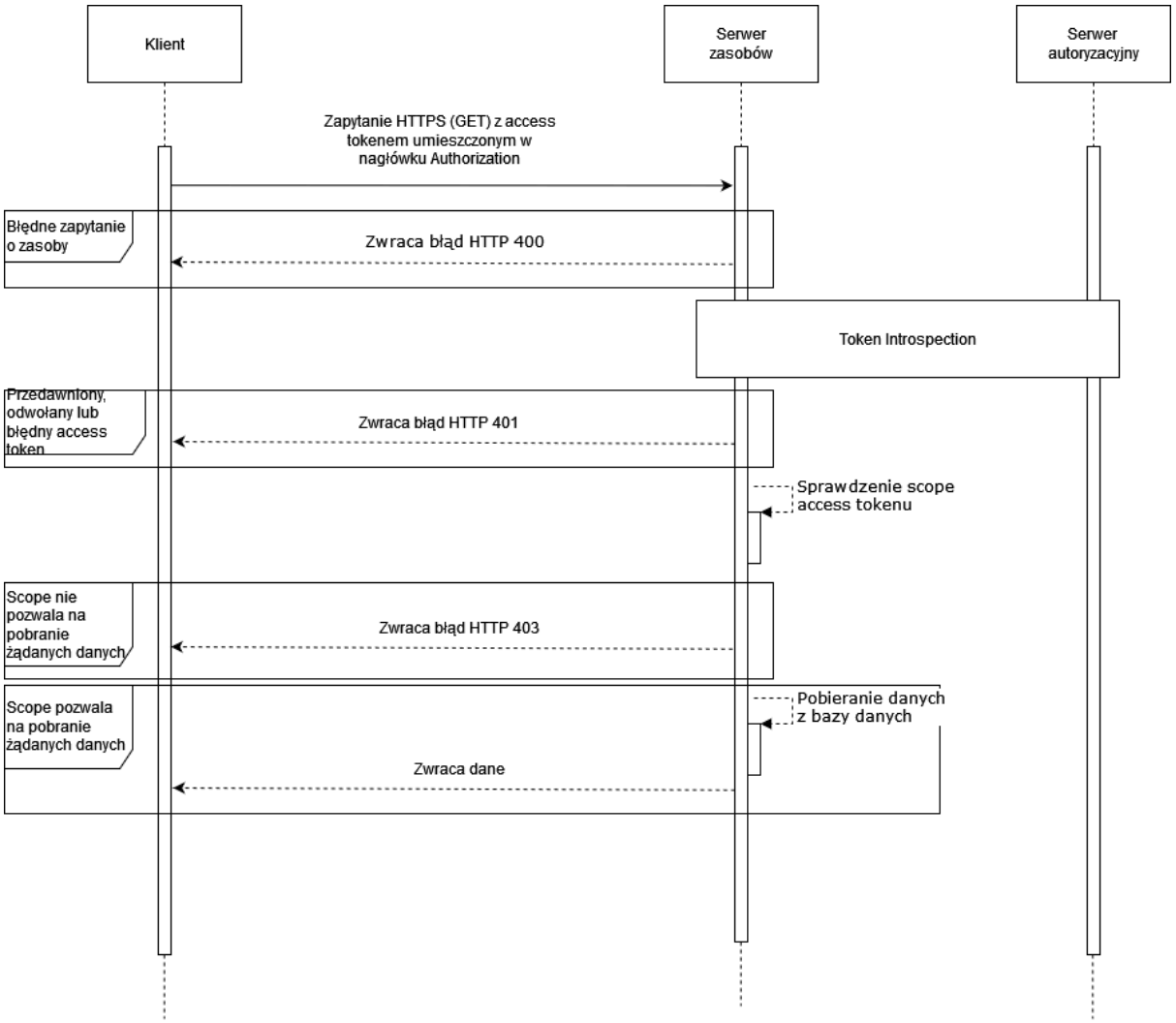
Diagram przedstawiający pomyślny Authorization - PKCE Flow z użyciem Custom Tabs

W celu uzyskania *access tokenu* oraz *refresh tokenu* mobilna aplikacja kliencka korzysta z **PKCE Flow**, na który składają się następujące kroki:

1. Aplikacja kliencka tworzy *code_verifier* (random string o długości 43-128 złożony ze znaków alfanumerycznych oraz znaków -_~.). *(RFC 7636, Section 4.1)*
2. Aplikacja kliencka tworzy *code_challange* jako hash *code_verifier* utworzony algorytmem SHA256. *(RFC 7636, Section 4.2)*
3. Aplikacja kliencka tworzy random stringa *state*, który będzie użyty do weryfikacji autentyczności odpowiedzi od serwera autoryzacyjnego.
4. Aplikacja kliencka przekazuje *code_challange* wraz z *authorization request* i pozostałymi informacjami do serwera autoryzacyjnego przy użyciu pośredniczącego activity Custom Tabs w celu odseparowania wrażliwych danych użytkownika od naszej aplikacji klienckiej, która jest publiczna. *(RFC 7636, Section 4.3, RFC 6749, Section 1.3.3)*
5. W przypadku pomyślnej autoryzacji, serwer odpowiada wysyłając ograniczony czasowy *authorization_code* oraz *state* na *redirect_uri*. *(RFC 7636, Section 4.4)*
    a. W przypadku nieudanej autoryzacji serwer zwraca HTTP 302 na *redirect_uri* z odpowiednim *error* (przykładowo *access_denied*) oraz *state*
6. Aplikacja kliencka porównuje swój *state* wraz z otrzymanym od serwera autoryzacyjnego. Jeśli one się nie zgadzają, to ponawiamy próbę autoryzacji rozpoczynając proces od punktu 1.

7. Aplikacja kliencka przesyła *authorization_code* wraz z *code_verifier* do serwera autoryzacyjnego w celu pozyskania access *tokenu, refresh tokenu*. *(RFC 7636, Section 4.5)*
8. Serwer autoryzacyjny przetwarza zapytanie, porównując hash uzyskany z *code_verifier* z wcześniej uzyskanym *code_challenge* oraz sprawdzając ważność przesłanego przez klienta *authorization_code*. W przypadku pomyślnej autoryzacji odpowiada przesyłając *access token*, *refresh token. (RFC 7636, Section 4.6).* W przypadku braku autoryzacji serwer serwer odpowiada HTTPS 401 (Unauthorized).

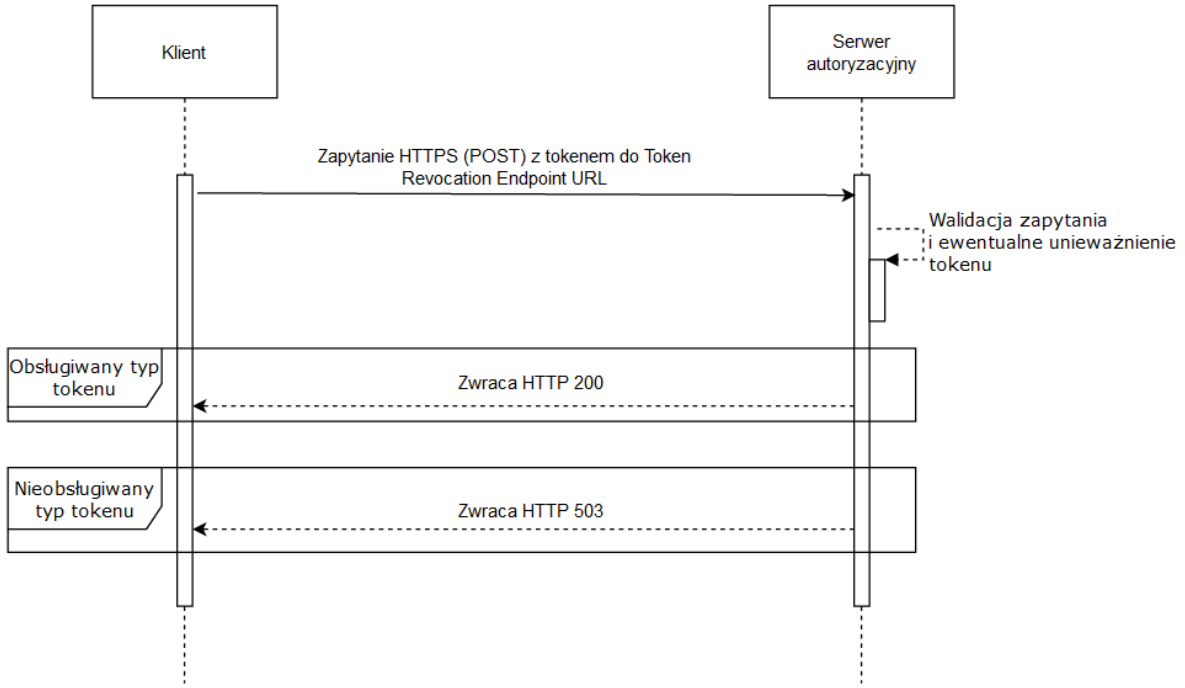
Zapytanie o zasoby do serwera zasobów



Klient chcąc uzyskać dostęp do zasobów wysyła zapytanie HTTPS (GET) z *access tokenem* umieszczonym w nagłówku *Authorization* do serwera zasobów. Serwer zasobów zwraca błąd HTTP 400 z wartością *error = invalid_response* w przypadku błędnego zapytania (przykładowo brak dołączonego *access tokena* w zapytaniu do serwera zasobów), natomiast jeśli zapytanie było poprawne, to dokonuje **Token Introspection** z udziałem serwera autoryzacyjnego.

Jeśli serwer zasobów dostanie informację, że podany *access token* jest błędny, przedawniony lub odwołany, to zwraca błąd HTTP 401 z nagłówkiem *WWW-Authenticate*, w którym zawiera poprawny *scope* do otrzymania żądanego zasobu, string *Bearer* (wskazujący na żądany typ tokenu) oraz *error = insufficient_scope*.
Gdy *access token* jest poprawny, serwer zasobów sprawdza, czy jego *scope* upoważnia klienta do otrzymania żądanego zasobu. Po sprawdzeniu serwer zasobów zwraca HTTP 403 z *error = insufficient_scope* w przypadku niewystarczających uprawnień do uzyskania zasobów, natomiast, gdy klient może otrzymać żądany zasób, to serwer zasobów pobiera dane z bazy danych i zwraca je klientowi.

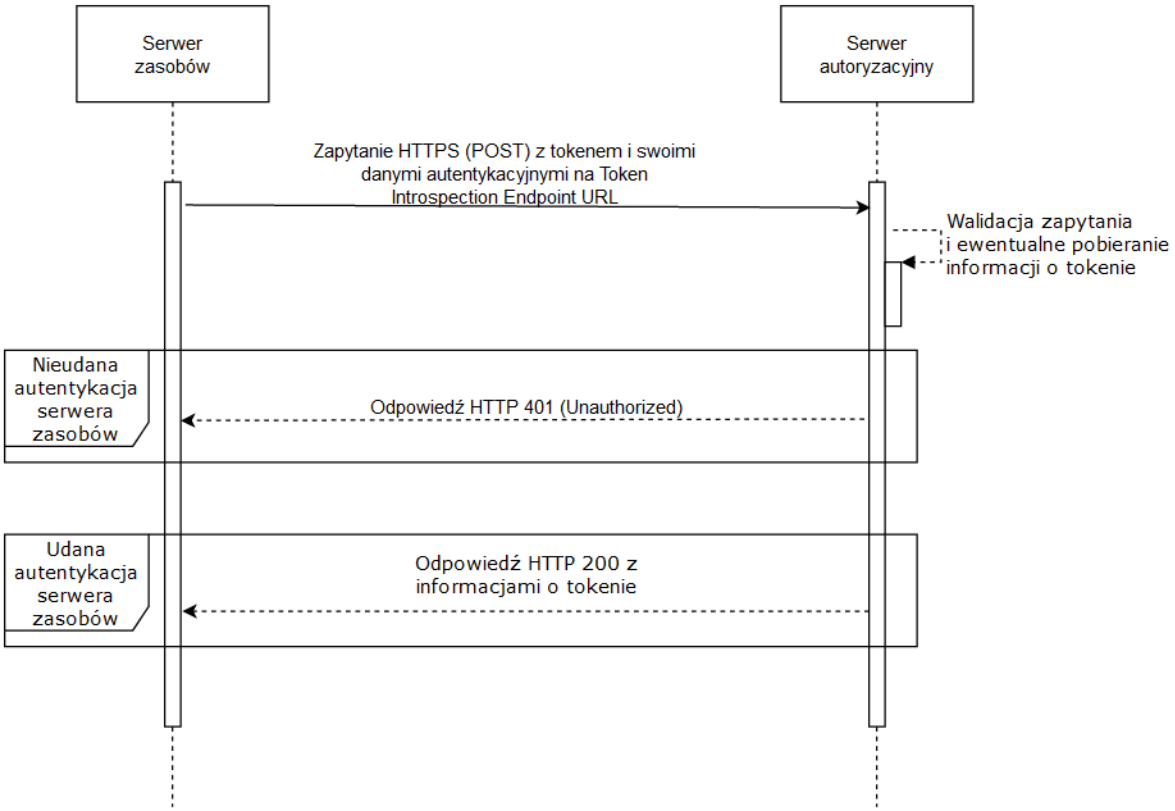
Token Revocation



Kiedy użytkownik chce wylogować się z aplikacji, klient wysyła zapytanie HTTPS (POST) do serwera autoryzacyjnego na *Token Revocation Endpoint URL* używając *Content-Type application/x-www-form-urlencoded* wraz z następującymi parametrami: *token* oraz *token_type_hint* (opcjonalnie, równy *accesss_token* lub *refresh_token*).
Jeśli klient przesyła *refresh token*, to serwer autoryzacyjny unieważnia także wszystkie *access tokeny* przyznane na ten sam *authorization grant* co przesłany *refresh token*.
Serwer autoryzacyjny waliduje zapytanie od klienta i jeśli otrzymuje nieobsługiwany przez ten serwer typ tokenu (serwer w danym momencie nie obsługuje unieważnienia access lub refresh tokenu), to zwraca HTTP 503 z *error code = unsupported_token_type*. Jeśli typ tokenu jest obsługiwany przez serwer autoryzacyjny, to uruchamia on procedurę unieważnienia tokenu i zwraca HTTP 200 do klienta, nawet jeśli unieważniany token był niepoprawny.
Token Revocation może inicjalizować sam serwer autoryzacyjny lub na przykład administrator chcący unieważnić tokeny przypisane do poszczególnych użytkowników aplikacji. Powyższy diagram działania dotyczy sytuacji, w której to użytkownik chce się

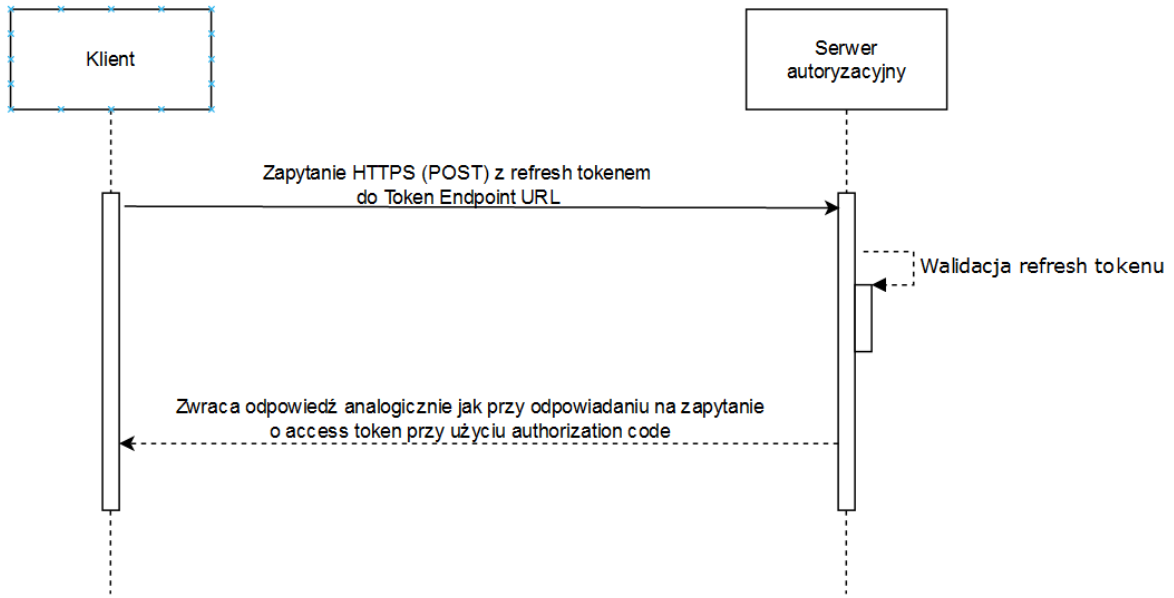
wylogować z aplikacji, wymuszając na kliencie wysłanie odpowiedniego zapytania do serwera autoryzacyjnego.

Token Introspection



Kiedy klient wysyła zapytanie do serwera zasobów, serwer zasobów musi zweryfikować otrzymany *access token*. Do tego służy **Token Introspection** polegający na komunikacji pomiędzy serwerem zasobów, a serwerem autoryzacyjnym.
Serwer zasobów wysyła zapytanie HTTPS (POST) do serwera autoryzacyjnego na *Token Introspection Endpoint URL* używając *Content-Type application/x-www-form-urlencoded* wraz z następującymi parametrami: *token* oraz *token_type_hint* (opcjonalnie), a także dane (*Bearer token*) do autentykacji serwera zasobów w serwerze autoryzacyjnym podane w nagłówku *Authorization*.
Serwer autoryzacyjny odpowiada przesyłając obiekt JSON w HTTP 200 o *Content-Type: application/json*. Przesyłany obiekt JSON zawiera informacje na temat tokenu: *active* (jedyny wymagany atrybut), *scope*, *client_id*, *exp*, *iat*. Wyjaśnienia poszczególnych atrybutów są w *Section 2.2 RFC 7662*
Jeśli sprawdzany token był niepoprawny, nieistniejący na serwerze autoryzacyjnym lub serwer zasobów nie ma uprawnień do sprawdzenia informacji o nim, to serwer autoryzacyjny zwraca serwerowi zasobów informacje o tokenie tak jak wyżej, ale tylko z atrybutem *active* ustawionym na *false*.
Jeśli dane autentykacyjne serwera zasobów są błędne, to serwer autoryzacyjny zwraca obiekt JSON z atrybutem *error* o odpowiedniej wartości zgodnie z *Section 5.2 RFC 6749* w odpowiedzi HTTP 401 (Unauthorized) z *Content-Type: application/json*.

Refresh Token



Aby uzyskać *access token* na podstawie posiadanego *refresh token* klient wysyła zapytanie HTTPS (POST) o *Content-Type: application/x-www-form-urlencoded* z parametrami: *grant_type* (ustawionym na *refresh_token*), *refresh_token* oraz opcjonalnie *scope*.
Serwer autoryzacyjny musi dokonać walidacji otrzymanego *refresh token* i na tej podstawie przyznać lub odrzucić prośbę o przyznanie *access token* w takim sam sposób, jak robi to przy przyznawaniu *access tokenu* na podstawie przysłanego *authorization code* (**Authorization - PKCE Flow**).

Biblioteki:

- auth0/java-jwt
- JUnit oraz Mockito do testów
- CustomTabs - pośredniczy między klientem a serwerem autoryzacyjnym, gwarantuje bezpieczeństwo aplikacji. Separuje aplikację kliencką od danych użytkownika.
- MongoDB Java Driver
- Netty - implementacja serwera autoryzacyjnego i serwera zasobów.
- org.json - JSON in Java

Wzorce projektowe

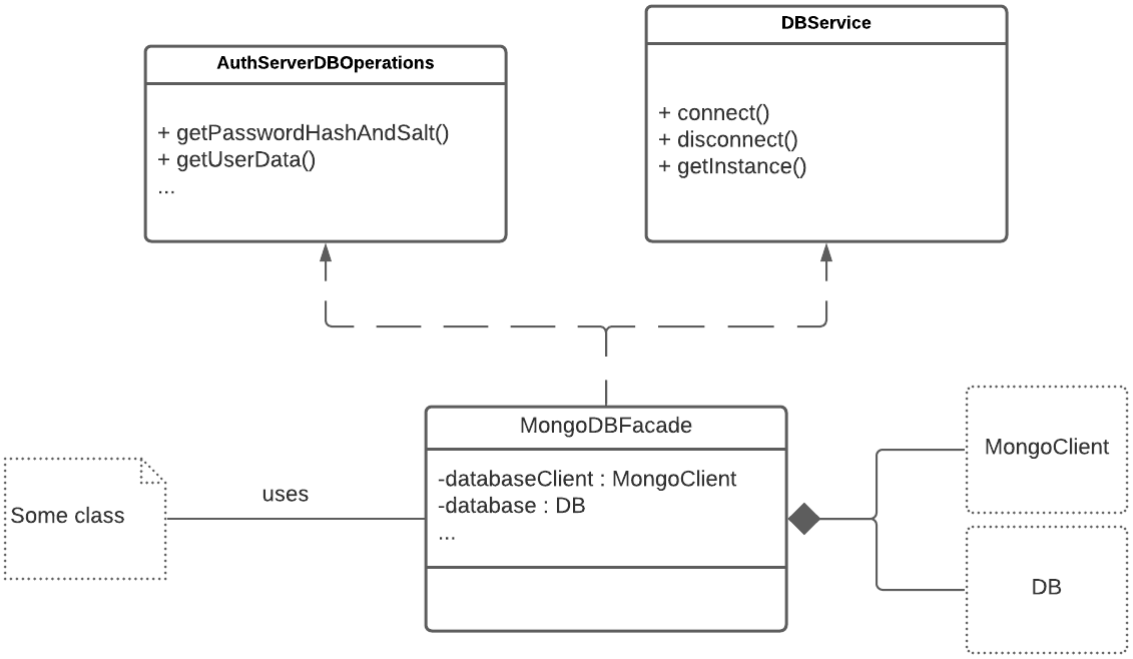
- Fasada

Dlaczego: Chcemy uzyskać prosty interfejs do pobierania informacji z bazy danych
Plusy:

1. Fasada zapewnia prosty interfejs agregujący metody wielu klas. Zmniejsza złożoność kodu w porównaniu z sytuacją, kiedy zamiast skorzystać z jednej metody fasady korzystalibyśmy z wielu klas, z których dana metoda fasady korzysta.
2. Fasada pozwala odpowiednio sterować agregowanymi metodami wielu klas w celu wykonania danego zadania w odpowiedni sposób.

Minusy:

1. Może stać się **god object** powiązanym z dużą liczbą klas - można wtedy pomyśleć o przeniesieniu części jej obowiązków do osobnej fasady
2. Łamie zasadę SRP



- **Singleton**

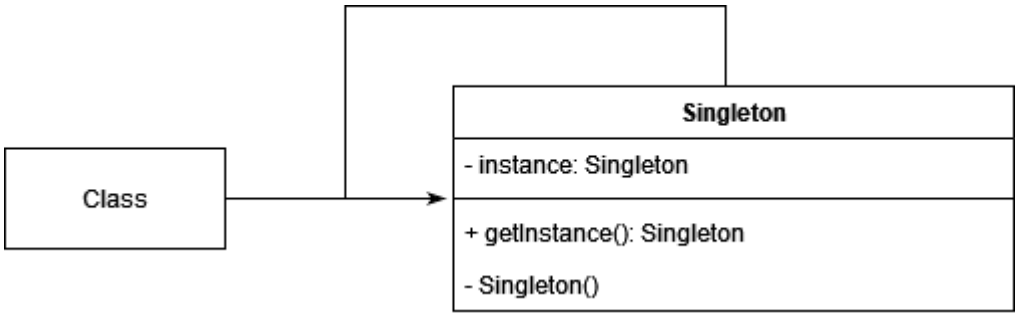
Dlaczego: Chcemy, aby w całej aplikacji była dostępna tylko jedna instancja klasy dającej dostęp do bazy danych
Plusy:

1. mamy gwarancję, że istnieje tylko jedna instancja klasy dającej dostęp do współdzielonych zasobów

Minusy:

1. łamie SRP (Single Responsibility Principle)

2. można doprowadzić do dużej zależności (coupling) w systemie, szczególnie przy stosowaniu wielu singletonów
3. utrudnia testy jednostkowe



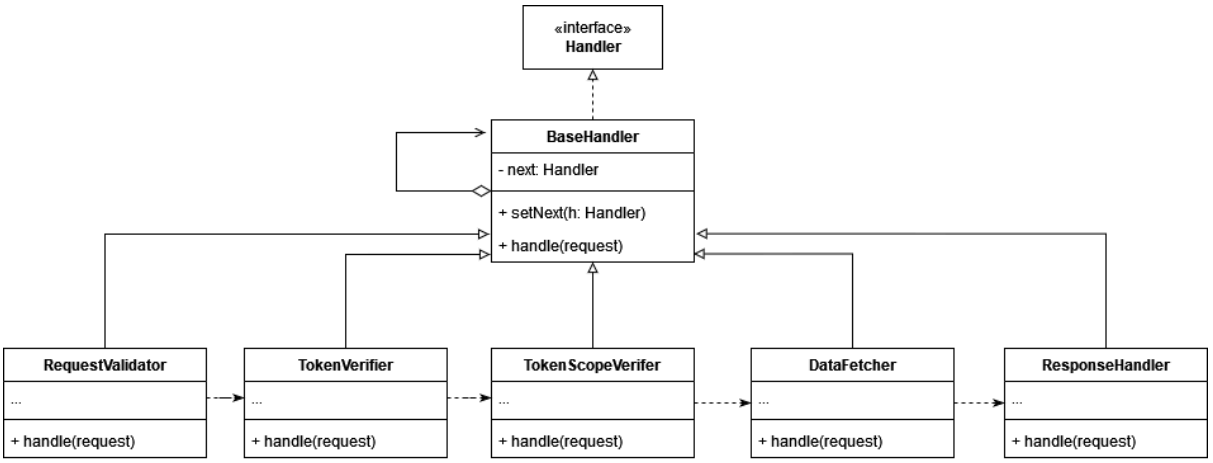
- Pipeline

Dlaczego: Chcemy uzyskać łańcuch obiektów przetwarzających sekwencyjnie dane zapytania HTTP / HTTPS.
Plusy:

1. pozwala uporządkować obsługę zadania (kolejność przetwarzania zapytania)
2. realizuje SRP (Single Responsibility Principle) - można łatwo podzielić obsługę całego zadania na etapy (np. weryfikacja zapytania HTTP, weryfikacja tokenu, pobranie danych z bazy, etc.)
3. realizuje OCP (Open / Closed Principle) - w razie potrzeby można dodać nowe klasy obsługujące zapytanie, bez konieczności zmiany już istniejących

Minusy:

1. trudno jest modyfikować pipeline w czasie działania programu



- Builder

Dlaczego: Chcemy uzyskać odseparowany od logiki serwera, prosty mechanizm tworzenia zróżnicowanych odpowiedzi HTTP.
Plusy:

1. pozwala na pozbycie się konstruktorów teleskopowych.

2. umożliwia tworzenie zróżnicowanych obiektów tego samego typu
3. wykorzystuje ten sam kod do tworzenia różnych obiektów
4. izoluje logikę tworzenia obiektów od logiki biznesowej aplikacji

Minusy:

1. w wyniku zastosowania wzorca możemy otrzymać wiele klas, zwiększa się złożoność kodu