Serwer:

main.cpp
Funkcja main odpowiada za utworzenie wątku podrzędnego - aplikacji serwera (*ConnectionManager*) oraz za uruchomienie konsoli admina (*ConsoleManager*).
Funkcji (*handle_client*) która tworzy obiekt ConnectionManagera - serwer, podawane jako argumenty są deskryptory pipe (deskryptor do odczytu i zapisu) tworzone zaraz po rozpoczęciu wykonania funkcji main, oraz port dla serwera, który jest podawany jako argument wywołania programu.
Po pomyślnym utworzeniu wątku podrzędego i przekazaniu mu 3-elementowej tablicy (deskryptory pipe oraz port), wątek nadrzędny uruchamia konsole admina (*ConsoleManager*).
Funkcja main kończy swoje wykonanie w momencie kiedy konsola admina zostaje zamknięta, wtedy zamykany jest deskryptor do pisania (pipe) oraz zamykany jest wątek podrzędny - serwer.

ConnectionManager
Główny moduł serwera odpowiadający za: skonfigurowanie gniazda nasłuchującego, odbiór nowych połączeń przychodzących do serwera, obsługę komend przesyłanych z konsoli admina oraz odbieranie wiadomości od już połączonych klientów.
Podczas utworzenia obiektu w ramach inicjalizacji przyjmowana jest 3-elementowa tablica argumentów (deskryptory pipe oraz port), tworzone jest gniazdo nasłuchujące (*create_listener*) oraz tworzony jest obiekt *Waiter* - odpowidzialny za zarządzanie deskryptorami (w tym funkcją select operująca na tych deskryptorach).
Swoje wykonanie rozpoczyna od funkcji *manage_connections,* która obsługuje główna pętlę reagującą na zmiany na odpowiednich deskryptorach zwracanych przez funkcję *make_select* obiektu *Waiter.*
Zmiana na deskryptorze:
- gniazda nasłuchującego uruchamia funkcję *handle_new_connection*
- do odczytu z pipe uruchamia *handle_console_request*
Natomiast każdy inny deskryptor zwrócony przez funkcję select (deskryptory połączonych klientów) powoduje uruchomienie funkcji *handle_client_request.*

handle_new_connection - przyjmuje nowe połączenie, wypełnia strukture adresu *sockaddr_in* korzystając z funkcji *accept,* a następnie za pomocą obiektu *Waiter* dodaje deskryptor nowego klienta oraz tworzy strukturę kliencką (*ClientStructure*) dodając ją w mapie *cli_struct.*

handle_console_request - pozwala na zarządzanie serwerem od strony administratora. Administrator może zamknąć wszystkie aktualne połączenia i zamknąć serwer. Dzięki tej funkcji także może wyświetlić aktualnych użytkowników, liderów i wszystkie istniejące grupy. Dodatkowo administrator ma możliwość usunięcia użytkownika nie mającego status lidera i możliwość usunięcia dowolnej grupy. Funkcja do komunikacji z serwerem używa 1 znaku, w odpowiedzi na taki znak otrzymuje znak powodzenia operacji ‘s’ (success) lub znak błędu ‘e’ (error).

handle_client_request - zleca odpowiedniej strukturze klienckiej pobranie części pakietu. Gdy ewentualnie połączenie zostaje zakończone, funkcja usuwa strukturę kliencką (*ClientStructure*) oraz zleca zamknięcie deskryptora obiektowi *Waiter*. Po odebraniu całej wiadomości wywoływana jest funkcja *selectAction* z klasy *ServerController,* która obsłuży requesta od klienta, wysyłając mu odpowiedniego responsa.

ClientStructure
Moduł odpowiadający za odbieranie/kolekcjonowanie kolejnych pakietów wysyłanych przez klienta. Do każdego deskryptora odpowiadającemu klientowi przypisany jest taki obiekt struktury, który składa w całą wiadomość pakiety przychodzące od klienta. Obiekt podczas inicjalizacji (która zachodzi także po każdym zebraniu całej wiadomości) ustawia parametry na domyślne, w tym parametr określający ile bajtów maksymalnie pobrać za pierwszym razem. Główną funkcją klasy jest *set_part_message* (opisana poniżej).

receive_part_message - korzysta z funkcji *recv* aby pobrać pakiet składający się z tylu bajtów ile aktualnie potrzebuje struktura kliencka (może być mniej). Następnie wywoływana jest funkcja *set_part_message,* która jako argument przyjmuje liczbę bajtów odebranych za pomocą *recv.*

set_part_message - zależnie od tego ile bajtów wiadomości udało się pobrać, przekopiowuje odebraną paczkę do bufora w strukturze klienckiej tym samym składając docelową wiadomość.
Jeśli całkowita liczba odebranych bajtów >= 4, wtedy funkcja odczytuje 4 pierwsze bajty złożonej dotychczas wiadomości (nagłówek - header) i przekształca zawartość w liczbę odpowiadającą długości całej wiadomości. Od tej pory struktura kliencka ustawia parametr potrzebnych bajtów na równowartość różnicy całej wiadomości i dotychczas odebranych bajtów - parametr ten jest wykorzystywany w funkcji *receive_part_message,* aby określić ile maksymalnie bajtów pobrać za pomocą funkcji *recv*.

Waiter
Moduł przechowujący wszystkie deskryptory używane w programie. Obiekt tej klasy tworzy sety deskryptorów i aktualizuje je na bieżąco oraz zarządza vectorem deskryptorów będących aktualnie w użyciu. *Waiter* jest jedynym obiektem korzystający z funkcji *select.* Posiada funkcje dodającą, zamykającą dany deskryptor oraz zamykającą wszystkie deskryptory, a główną funkcją jest *make_select* (opisana poniżej).

make_select - funkcja wywoływana z poziomu *ConnectionManager*, z funkcji *manage_connections.* Korzystając z funkcji *select* otrzymuje zbiór deskryptorów, a następnie przechodzi przez vector wszystkich deskryptorów będących w użyciu i sprawdza czy dany deskryptor jest "set", jeśli tak to dodaje dany deskryptor do wynikowego vectora, którego następnie zwraca.

PackageSizeParser
Zawiera 2 funkcje: *parse_int_32* oraz *serialize_int_32,* służace do przekształcania headera w liczbę i odwrotnie.

parse_int_32 - przekształca tablicę 4-elementową zawierającą bitowo zapisaną liczbę (rozmiar wiadomości) w liczbę typu integer.

serialize_int_32 - przekształca liczbę typu integer w tablicę 4-elementową.

MessageParser
Zawiera funkcję, która dostaje jako argument strukturę klienta CientStructure i parsuje zawarta w niej wiadomość na jsona przy użyciu biblioteki *nlohmann.*

ServerController
Wyższa warstwa komunikacji client-server odpowiadająca za poprawna obsluge requestów klienta oraz wyslanie mu właściwie zbudowanego responsa.

Główna funkcja modułu jest funkcja *selectAction* która na podstawie flagi zawartej w otrzymanym jsonie decyduje, którą funkcje klasy *ServerController* wywołać. Flaga otrzymywana jest za pomocą funkcji convert, która zamienia stringa zawartego w jsonie na flagę *RequestFlag*.

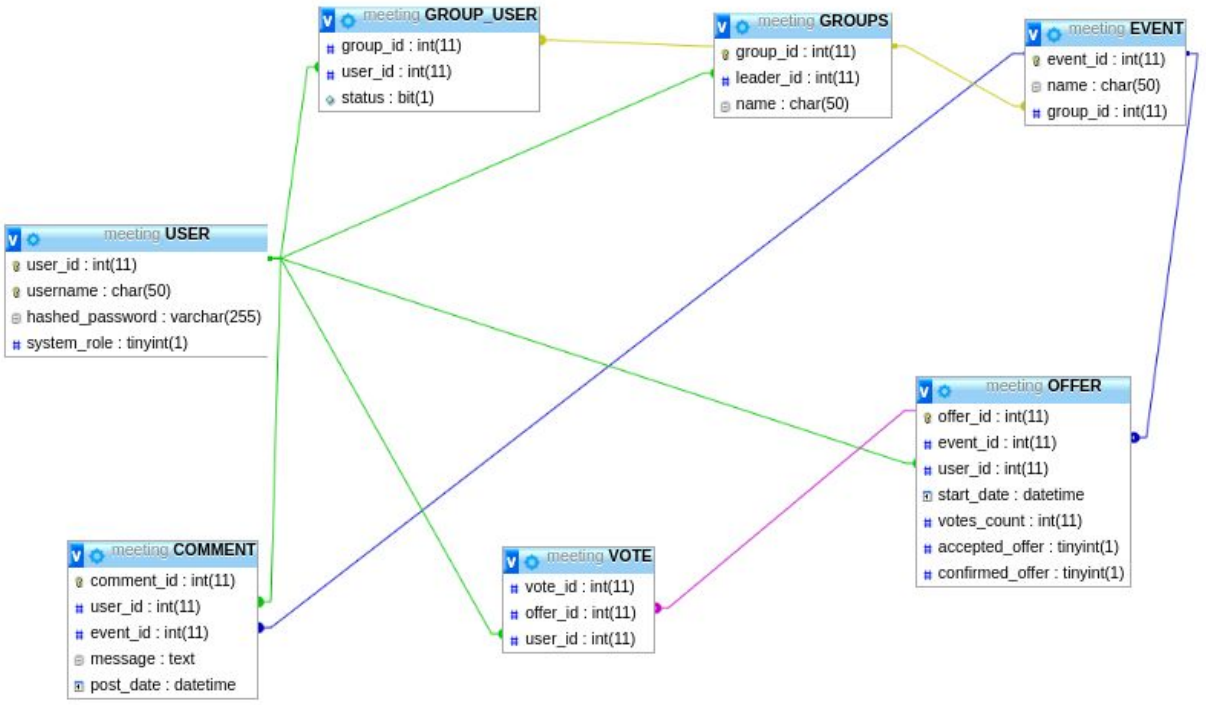
Klasa *ServerController* zawiera wiele funkcji odpowiedzialnych za wywołanie odpowiednich dla danego requesta funkcji klasy *DataBaseConnection* w celu stworzenia responsa dla klienta. Stworzenia responsa wymaga wykonania zapytań w bazie danych i ewentualne odesłanie pobranych informacji do klienta.

Funkcja *sendResponse* odpowiedzialna jest za wysłanie wiadomości do klienta, który przysłał requesta do servera. Zamienia ona otrzymanego stringa na tablice znaków, a następnie przy użyciu funkcji *send_all* zawartej w niższej warstwie komunikacji *ConnectionMannager* wysyła response do klienta.

DataBaseConnection:
Moduł odpowiedzialny za tworzenie połączenia z baza danych (Connector/C++) oraz wykonywanie zapytań do bazy w celu wykonania zleconej akcji przez moduł *ServerController*. Funkcje tej klasy są odpowiedzialne za poprawne działanie w bazie danych oraz tworzenie na podstawie wyników zapytań treści wiadomości wysyłanej później do klienta.

Baza danych:

W projekcie wykorzystana została baza danych MySQL. Model bazy danych został przedstawiony poniżej.



Klient

Warstwy:

- Warstwa aplikacji
- Warstwa serializacji
- Warstwa komunikacji

Warstwa aplikacji

Interfejs graficzny użytkownika aplikacji został wykonany za pomocą biblioteki JavaFX. Interfejs składa się z 7 widoków:

1. AllGroupsWindow.fxml
2. EventsWindow.fxml
3. GroupsWindow.fxml
4. LoginWindow.fxml
5. OffersWindow.fxml
6. RegistrationWindow.fxml
7. RequestsReviewWindow.fxml

Za obsługę widoków odpowiadają kontrolery. Kontrolery zawierają logikę aplikacji, mają dostęp do klas modelowych i wykonują operacje na obiektach tych klas. Poprzez kontrolery warstwa aplikacji komunikuje się z warstwą serializacyjną. Tworzone są obiekty Request, które są wypełnianie danymi i wysyłane do warstwy serializacji. Warstwa serializacji zwraca obiekty typu Response, zawierające żądane dane.
Aplikacja zawiera 6 klas modelowych:

1. Comment
2. Event
3. Group
4. Offer
5. User
6. Vote

Klasy modelowe są klasami encyjnymi, czyli mają swoje odwzorowanie jako tabele w bazie danych. W aplikacji obiekty tych klas są tworzone podczas ładowania odpowiednich widoków, a później danymi tych obiektów są wypełniane tabele.

Warstwa serializacji

Warstwa serializacji pośredniczy między warstwami aplikacji i komunikacji. Zapisuje obiekty Request przekazywane przez kontrolery aplikacji jako łańcuchy znaków w formacie JSON i podaje je do warstwy komunikacji. Z warstwy komunikacji pobiera odpowiedź w takiej samej postaci i parsuje je na obiekty Response. Takie obiekty zwraca do kontrolera.

Do serializacji służy klasa Serializer.java, która zawiera parser (com.google.gson.Gson) oraz metody, przyjmujące jako argument obiekt Request i zwracające obiekt Response.

Warstwa komunikacji

Do komunikacji z serwerem służy klasa ConnectionManager.java. Moduł realizujący najniższą warstwę komunikacyjną po stronie klienta wysyła requesta i odbierania responsa od serwera, główną funkcją jest *sendRequestRecResponse* (opisana poniżej).
Podczas tworzeniu obiektu rozstawiane jest gniazdo które łączy się pod wskazany adres i port. Następnie inicjalizowane są wszystkie paramety (inicjalizacja odbywa się także po odbiorze całej wiadomości) w tym parametr określający ile bajtów potrzeba jeszcze do uzbierania pełnej wiadomości.
Dodatkowo klasa posiada funkcje konwertujące tablice 4-elementową (header) na liczbę typu integer i odwrotnie: liczbę typu ineteger na tablicę 4-elementową, te funkcje realizują odpowiednio funkcje *convertHeaderToInt* i *convertIntToHeader*.

sendRequestRecResponse - przyjmuje jako argument wywołania string będący requestem. Daną wiadomość "opakowuje" (dołącza tablice 4-elementową na poczatek stringa) w nagłówek, następnie korzysta z funkcji *sendRequest* do wysłania wiadomości do serwera i funkcji *receiveResponse* do odebrania odpowiedzi od serwera. Następnie usuwa pierwsze 4 bajty - nagłówek i zwraca docelowy response - wiadomość otrzymaną od serwera bez tablicy 4-elementowej będącej nagłówkiem.

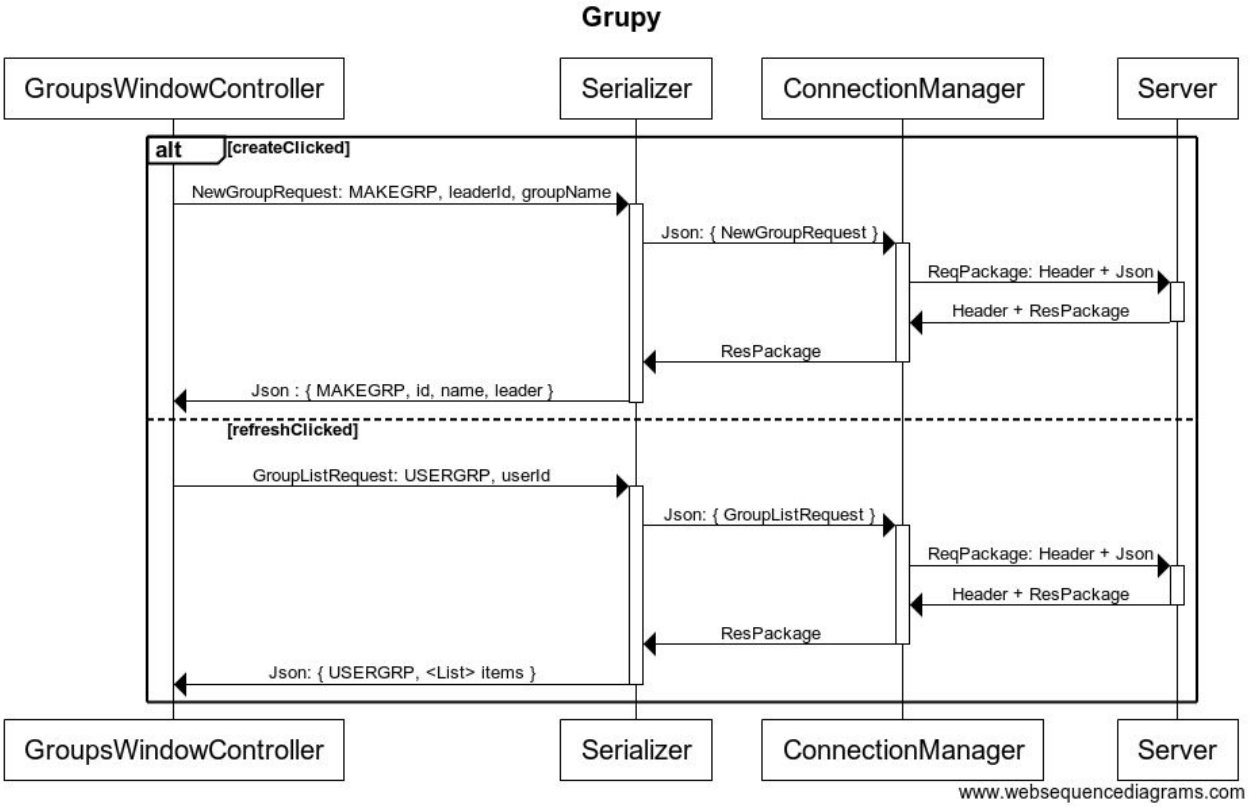
sendRequest - korzystając z obiektu klasy *OutputStream* wysyła daną w argumencie wiadomość na adres związany z gniazdem podczas wykonania konstruktora.

receiveResponse - mechanizm identyczny jak ten w *ClientStructure* po stronie serwera: funkcja za pomocą obiektu klasy *DataInputStream* pobiera maksymalnie zadaną ilość bajtów a następnie wywołuje funkcję *handlePortion.*

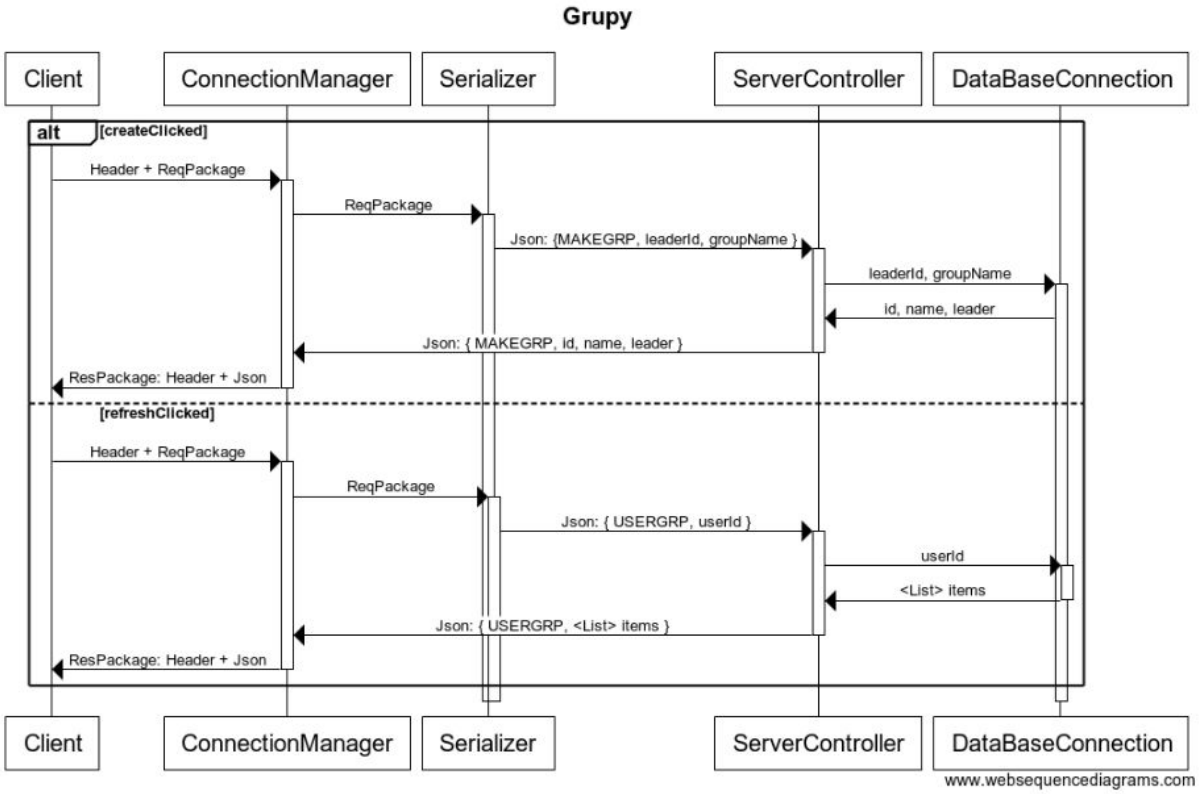
handlePortion - zależnie od tego ile bajtów wiadomości udało się pobrać, przekopiowuje odebraną paczkę do bufora tym samym składając docelową wiadomość.
Jeśli całkowita liczba odebranych bajtów >= 4, wtedy funkcja odczytuje 4 pierwsze bajty złożonej dotychczas wiadomości (nagłówek - header) i przekształca zawartość w liczbę odpowiadającą długości całej wiadomości. Od tej pory parametr określający ilość potrzebnych bajtów na równowartość różnicy całej wiadomości i dotychczas odebranych bajtów - parametr ten jest wykorzystywany w funkcji *receiveResponse,* aby określić ile maksymalnie bajtów pobrać za pomocą funkcji *read* obiektu klasy *DataInputStream*.

Diagramy sekwencji:

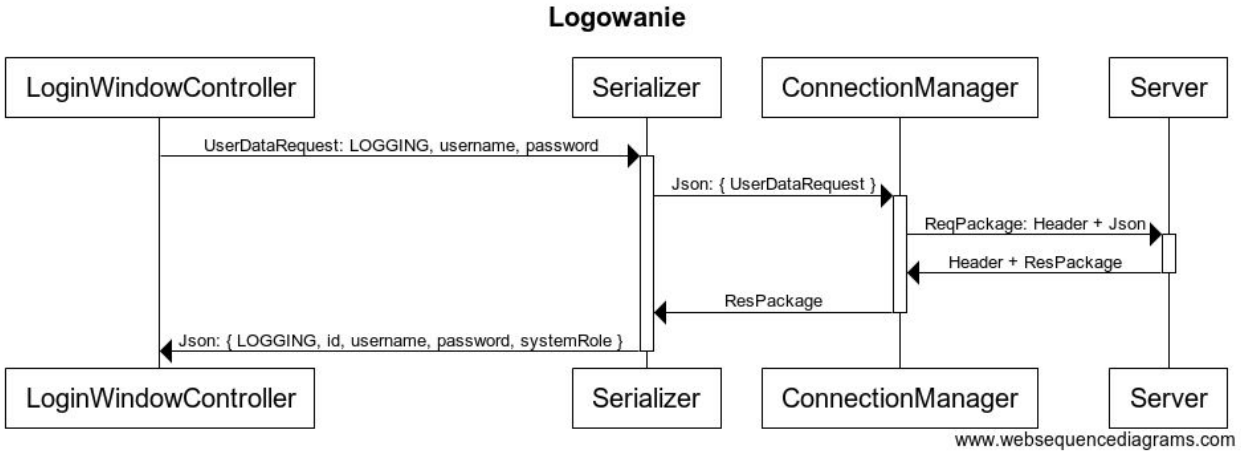
Klient:



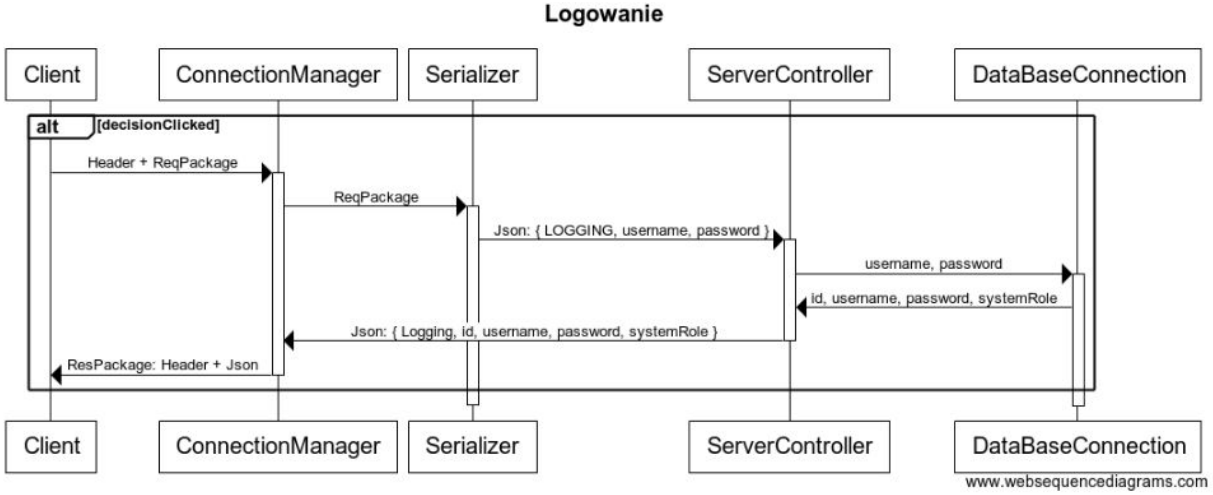
Serwer:



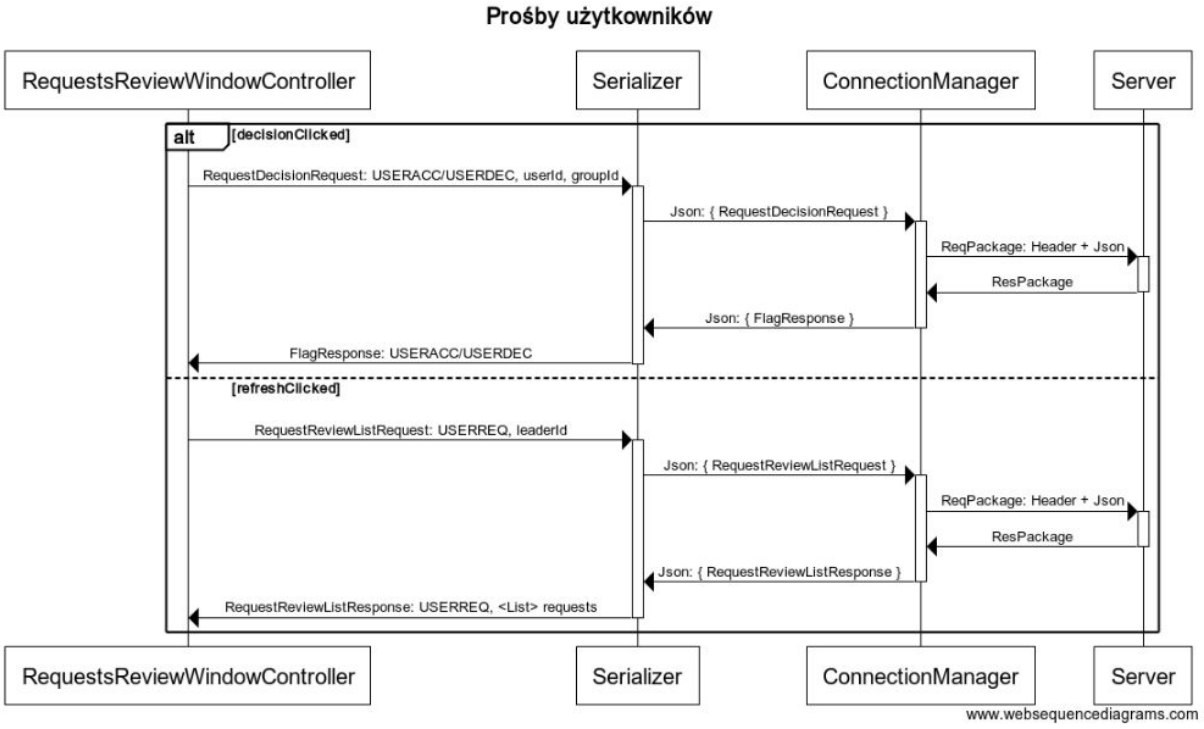
Klient:



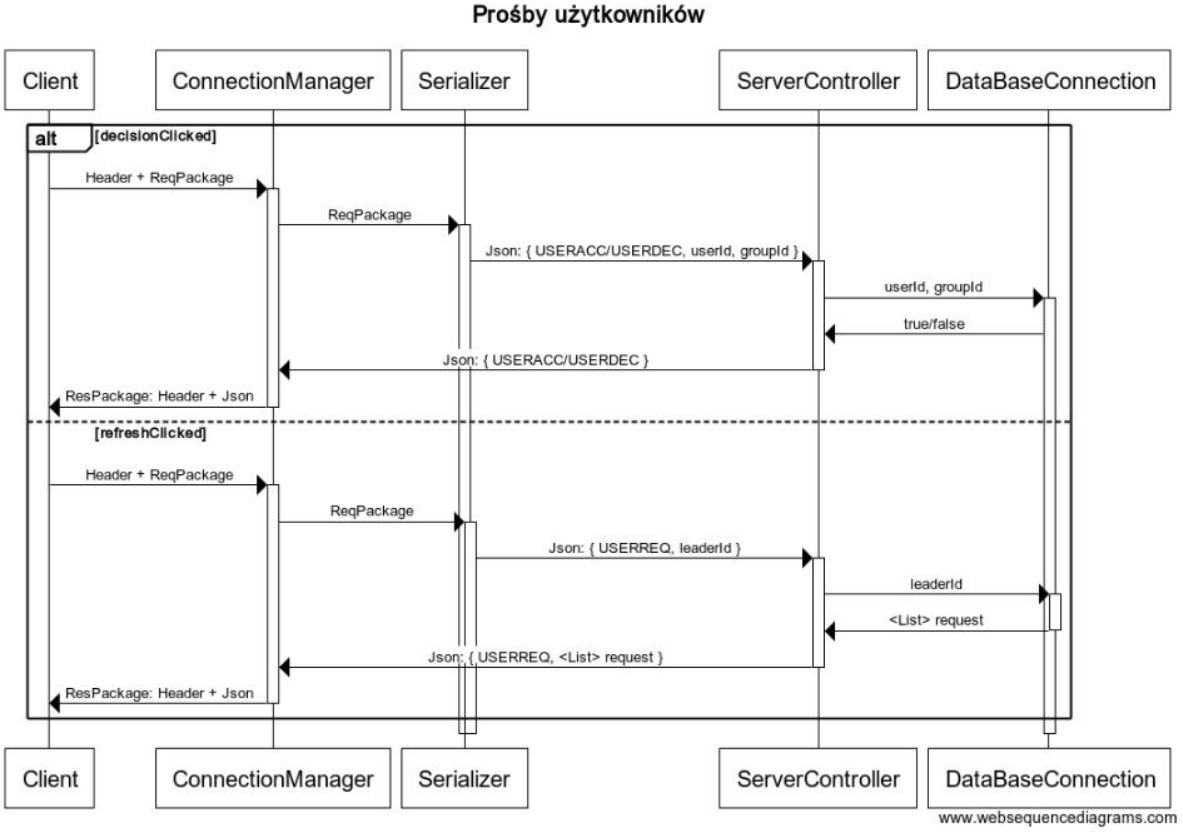
Serwer:



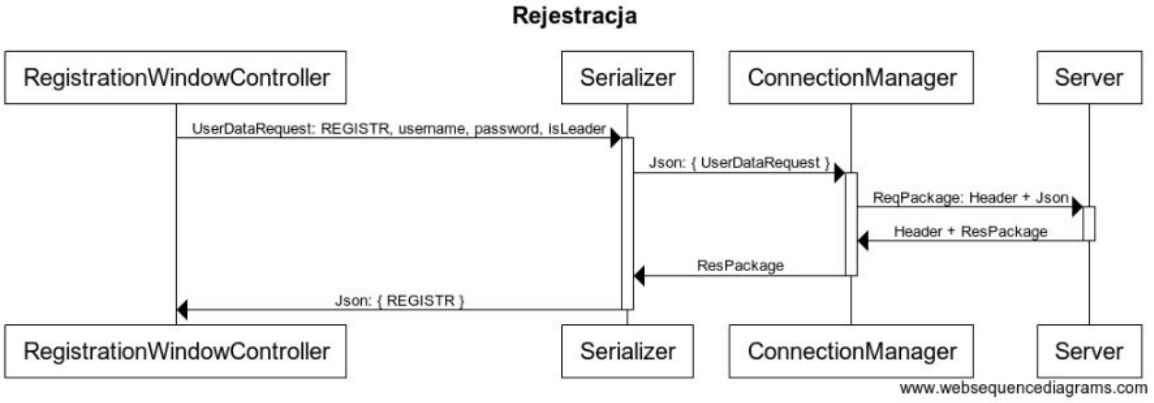
Klient:



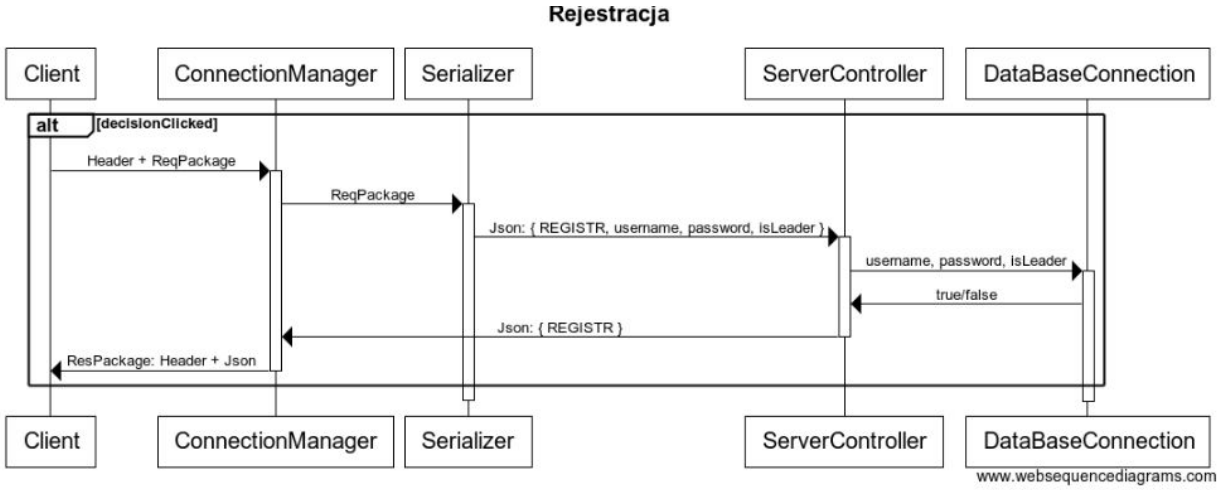
Serwer:



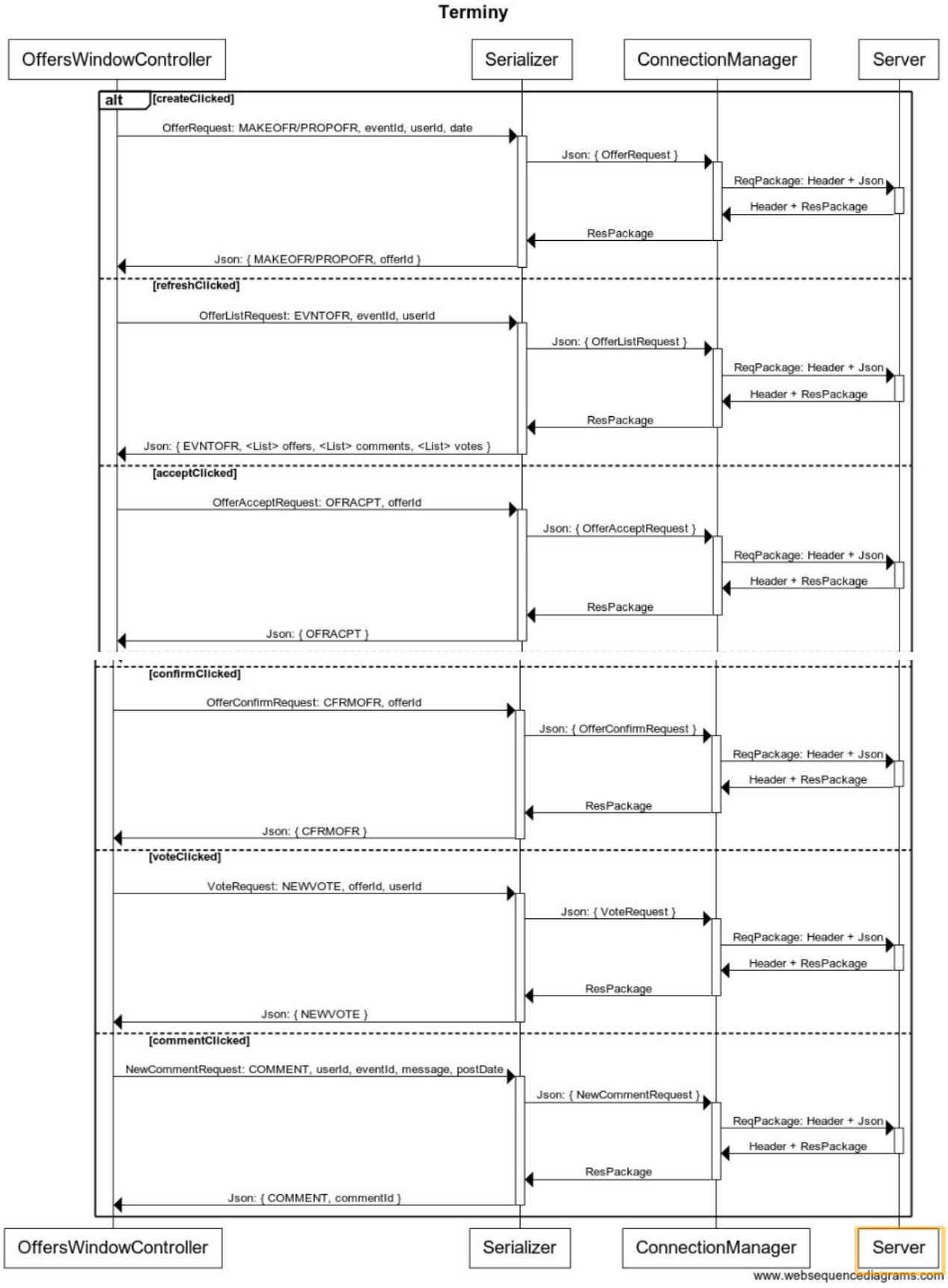
Klient:



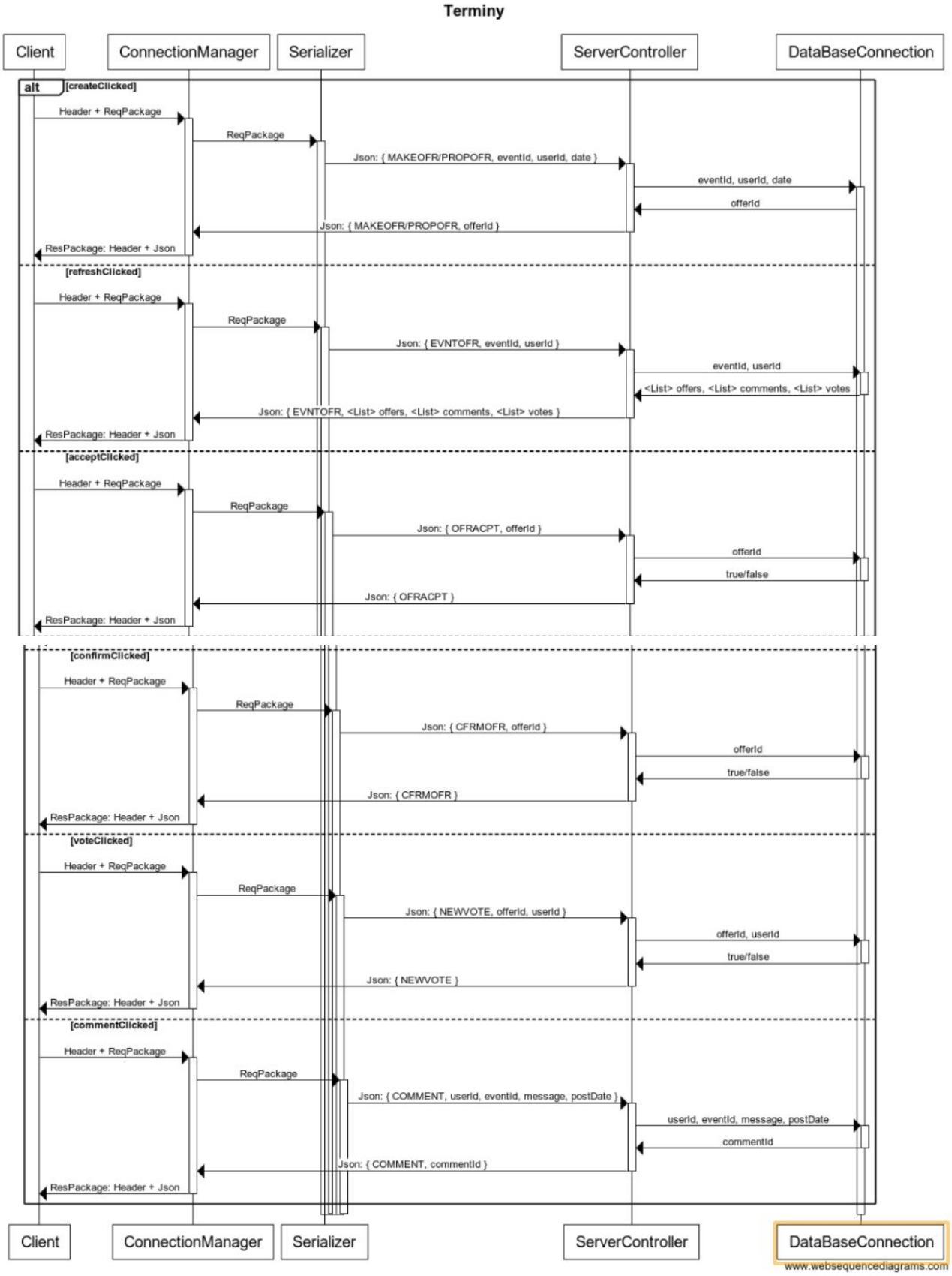
Serwer:



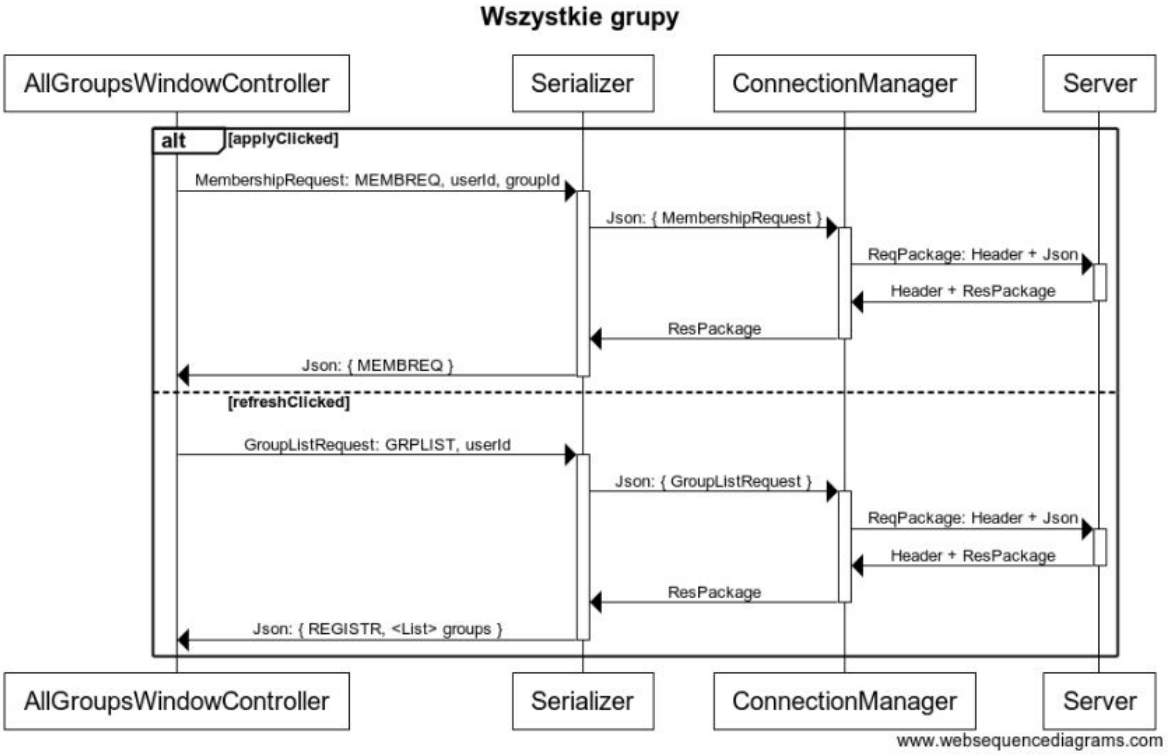
Klient:


Terminy
OffersWindowController
Serializer
ConnectionManager
Server
alt
[createClicked]
OfferRequest: MAKEOFR/PROPOFR, eventId, userId, date
Json: { OfferRequest }
ReqPackage: Header + Json
Header + ResPackage
ResPackage
Json: { MAKEOFR/PROPOFR, offerId }
[refreshClicked]
OfferListRequest: EVNTOFR, eventId, userId
Json: { OfferListRequest }
ReqPackage: Header + Json
Header + ResPackage
ResPackage
Json: { EVNTOFR, <List> offers, <List> comments, <List> votes }
[acceptClicked]
OfferAcceptRequest: OFRACPT, offerId
Json: { OfferAcceptRequest }
ReqPackage: Header + Json
Header + ResPackage
ResPackage
Json: { OFRACPT }
[confirmClicked]
OfferConfirmRequest: CFRMOFR, offerId
Json: { OfferConfirmRequest }
ReqPackage: Header + Json
Header + ResPackage
ResPackage
Json: { CFRMOFR }
[voteClicked]
VoteRequest: NEWVOTE, offerId, userId
Json: { VoteRequest }
ReqPackage: Header + Json
Header + ResPackage
ResPackage
Json: { NEWVOTE }
[commentClicked]
NewCommentRequest: COMMENT, userId, eventId, message, postDate
Json: { NewCommentRequest }
ReqPackage: Header + Json
Header + ResPackage
ResPackage
Json: { COMMENT, commentId }
OffersWindowController
Serializer
ConnectionManager
Server
www.websequencediagrams.com

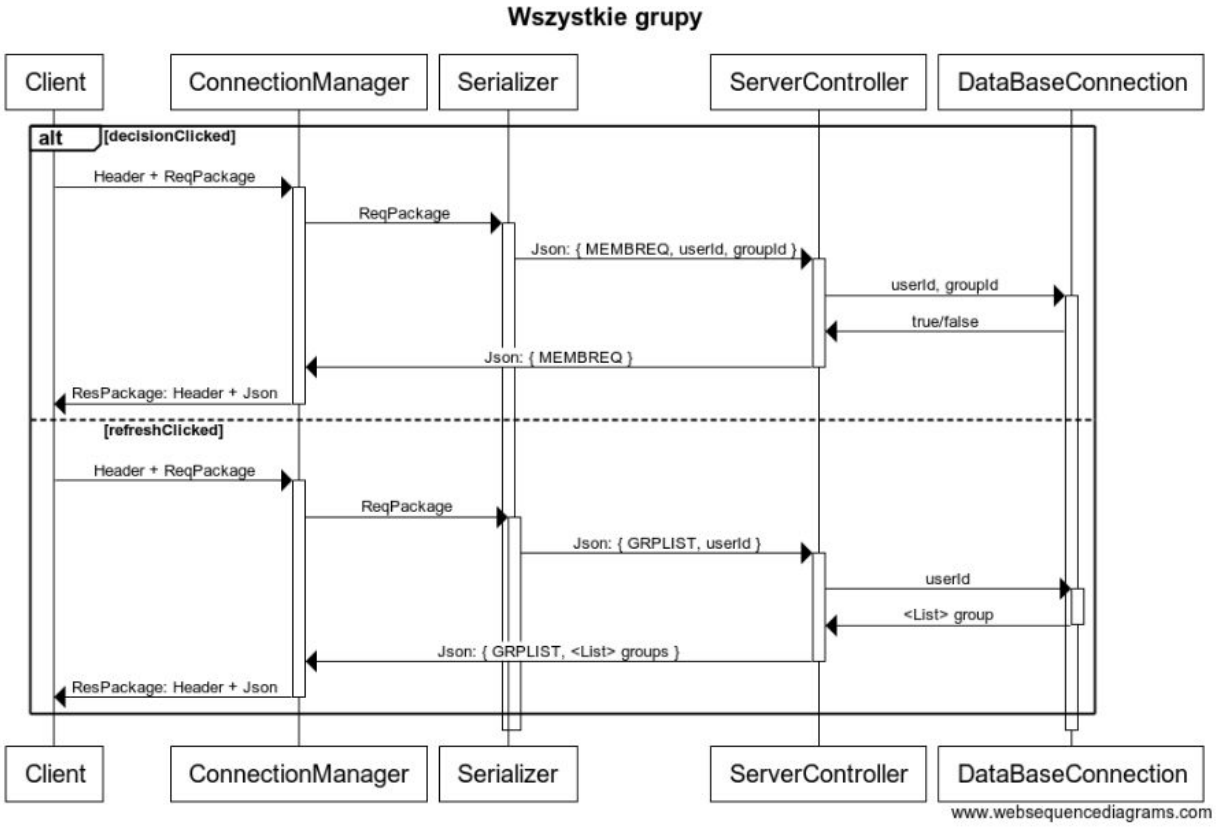
Serwer:


Terminy
Client
ConnectionManager
Serializer
ServerController
DataBaseConnection
alt
[createClicked]
Header + ReqPackage
ReqPackage
Json: { MAKEOFR/PROPOFR, eventId, userId, date }
eventId, userId, date
offerId
Json: { MAKEOFR/PROPOFR, offerId }
ResPackage: Header + Json
[refreshClicked]
Header + ReqPackage
ReqPackage
Json: { EVNTOFR, eventId, userId }
eventId, userId
<List> offers, <List> comments, <List> votes
Json: { EVNTOFR, <List> offers, <List> comments, <List> votes }
ResPackage: Header + Json
[acceptClicked]
Header + ReqPackage
ReqPackage
Json: { OFRACPT, offerId }
offerId
true/false
Json: { OFRACPT }
ResPackage: Header + Json
[confirmClicked]
Header + ReqPackage
ReqPackage
Json: { CFRMOFR, offerId }
offerId
true/false
Json: { CFRMOFR }
ResPackage: Header + Json
[voteClicked]
Header + ReqPackage
ReqPackage
Json: { NEWVOTE, offerId, userId }
offerId, userId
true/false
Json: { NEWVOTE }
ResPackage: Header + Json
[commentClicked]
Header + ReqPackage
ReqPackage
Json: { COMMENT, userId, eventId, message, postDate }
userId, eventId, message, postDate
commentId
Json: { COMMENT, commentId }
ResPackage: Header + Json
Client
ConnectionManager
Serializer
ServerController
DataBaseConnection
www.websequencediagrams.com

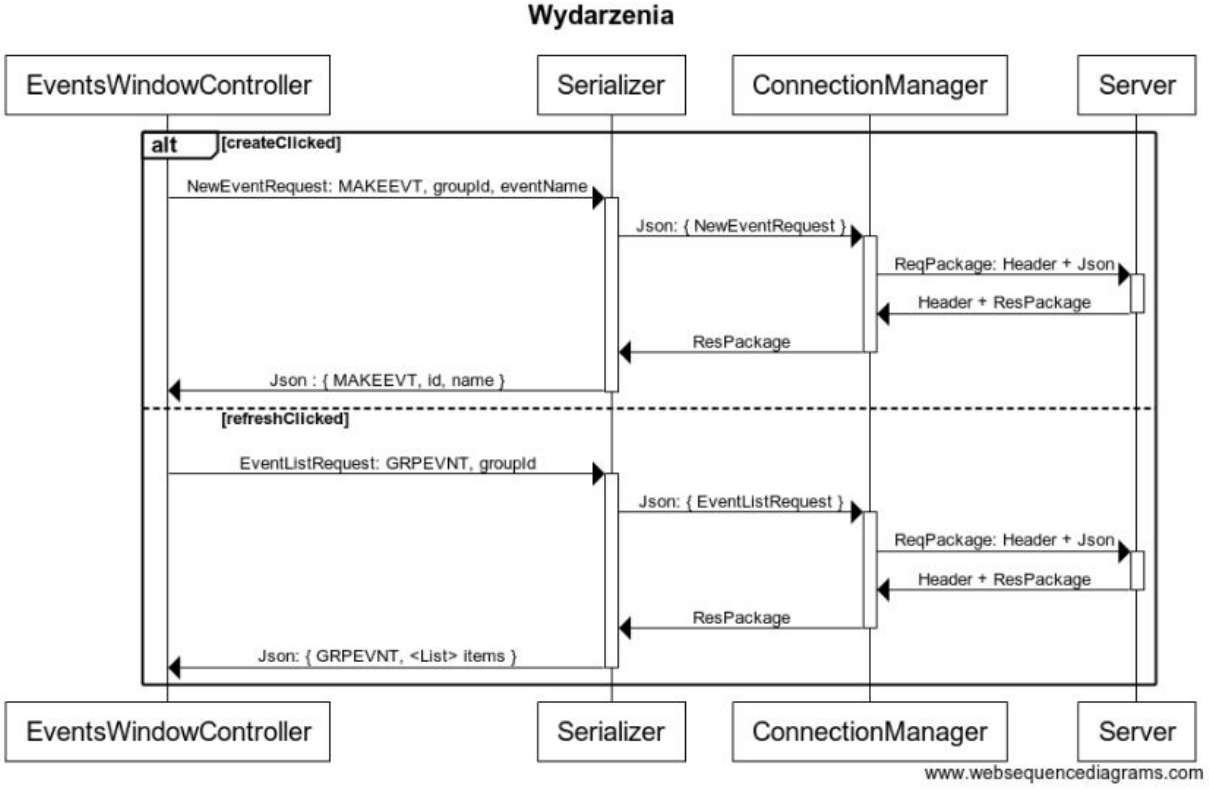
Klient:



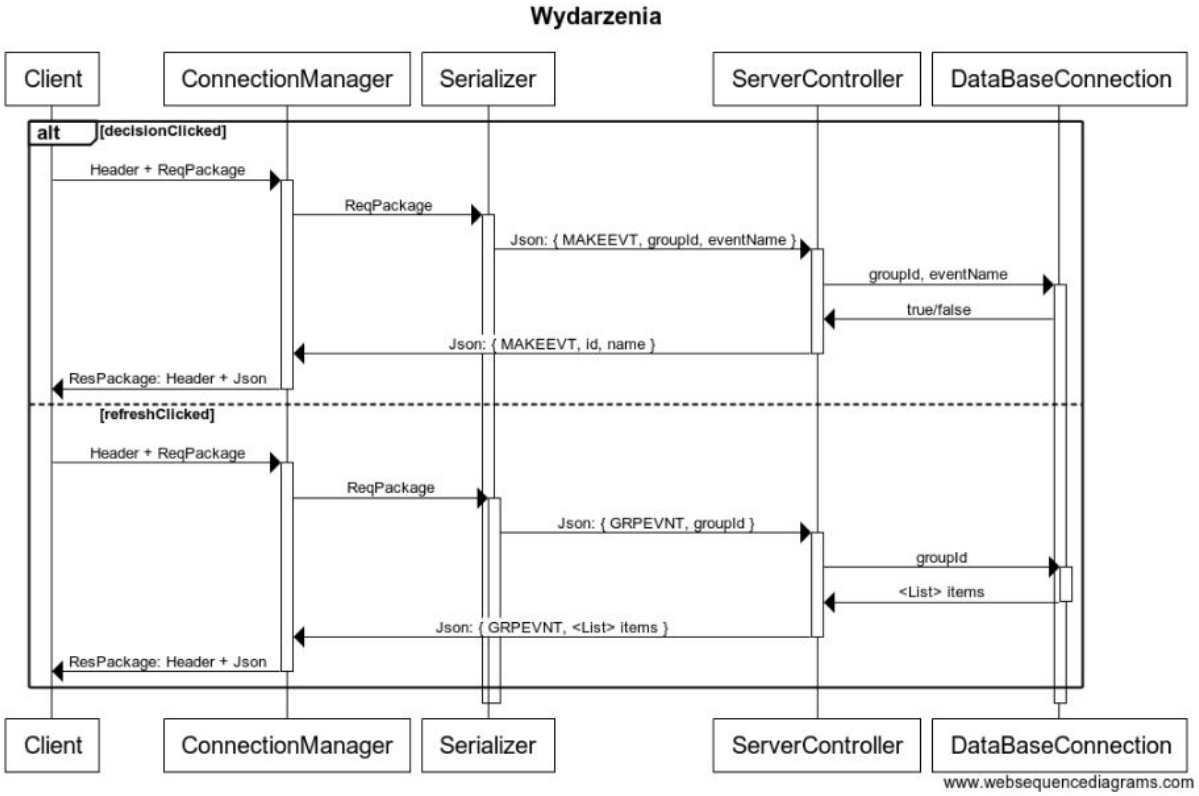
Serwer:


Wszystkie grupy
Client
ConnectionManager
Serializer
ServerController
DataBaseConnection
alt
[decisionClicked]
Header + ReqPackage
ReqPackage
Json: { MEMBREQ, userId, groupId }
userId, groupId
true/false
Json: { MEMBREQ }
ResPackage: Header + Json
[refreshClicked]
Header + ReqPackage
ReqPackage
Json: { GRPLIST, userId }
userId
<List> group
Json: { GRPLIST, <List> groups }
ResPackage: Header + Json
Client
ConnectionManager
Serializer
ServerController
DataBaseConnection
www.websequencediagrams.com

Klient:


Wydarzenia
EventsWindowController
Serializer
ConnectionManager
Server
alt
[createClicked]
NewEventRequest: MAKEEVT, groupId, eventName
Json: { NewEventRequest }
ReqPackage: Header + Json
Header + ResPackage
ResPackage
Json : { MAKEEVT, id, name }
[refreshClicked]
EventListRequest: GRPEVNT, groupId
Json: { EventListRequest }
ReqPackage: Header + Json
Header + ResPackage
ResPackage
Json: { GRPEVNT, <List> items }
EventsWindowController
Serializer
ConnectionManager
Server
www.websequencediagrams.com

Serwer:


Wydarzenia
Client
ConnectionManager
Serializer
ServerController
DataBaseConnection
alt
[decisionClicked]
Header + ReqPackage
ReqPackage
Json: { MAKEEVT, groupId, eventName }
groupId, eventName
true/false
Json: { MAKEEVT, id, name }
ResPackage: Header + Json
[refreshClicked]
Header + ReqPackage
ReqPackage
Json: { GRPEVNT, groupId }
groupId
<List> items
Json: { GRPEVNT, <List> items }
ResPackage: Header + Json
Client
ConnectionManager
Serializer
ServerController
DataBaseConnection
www.websequencediagrams.com