|  |
| --- |
| МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования  **"МИРЭА - Российский технологический университет"**  **РТУ МИРЭА** |
| Институт информационных технологий (ИТ) |
| Кафедра инструментального и прикладного программного обеспечения |



|  | |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №5** | |
| **по дисциплине** | |
| **«Архитектура клиент-серверных приложений»** | |
| Выполнил студент группы **ИКБО-01-21** | Маров Г.А. |
| Принял ассистент кафедры ИиППО | Волков М.Ю. |

| Практические работы выполнены | « » 2023г. |  |
| --- | --- | --- |
| «Зачтено» | « » 2023г. |  |

Москва 2023

**Теоретическое введение**

Практическая работа №5. Введение в многослойные клиент-серверные архитектуры.

**Цель работы**

**Теоретические сведения**

В предыдущих практических работах мы неявно познакомились с одной из архитектур распределенных систем: клиент-серверная архитектура.

**‎Тонкие клиенты‎**

‎Тонкий клиент спроектирован так, чтобы основная часть обработки данных происходила на сервере. Тонкий клиент как правило без жесткого диска: действуют как простой терминал к серверу и требует постоянной связи с сервером. ‎

**‎‎Толстые клиенты‎**

‎Толстый клиент выполняет основную часть обработки. У толстых клиентов нет необходимости в непрерывной связи с сервером, поскольку они в основном передают информацию на сервер.

**Когда какой клиент использовать**‎

Тонкие клиенты обеспечивают работу рабочего стола в средах, где конечный пользователь имеет четко определенное и регулярное количество задач, для которых используется система. Тонких клиентов можно найти в медицинских офисах, авиабилетах, школах, правительствах, производственных предприятиях и даже колл-центрах. Наряду с простотой установки, тонкие клиенты также предлагают более низкую общую стоимость владения по сравнению с толстыми клиентами.

Если вашим приложениям требуются мультимедийные компоненты или которые интенсивно используют пропускную способность, стоит присмотреться к разработке толстых клиентов. Одно из самых больших преимуществ толстых клиентов – некоторые операционные системы и программное обеспечение не могут работать на тонких клиентах. Толстые клиенты могут справиться с ними, поскольку у них есть свои собственные ресурсы.

**Трехуровневая архитектура распределенных систем**

Трехуровневая архитектура (рисунок 1) организует приложения в три логических и физических вычислительных уровня: уровень представления (пользовательский интерфейс), уровень приложений (обработка данных происходит здесь) и уровень данных (хранение и управление данными). Поскольку каждый уровень работает в своей собственной инфраструктуре, он может разрабатываться одновременно отдельной командой разработчиков и может обновляться или масштабироваться по мере необходимости, не влияя на другие уровни, не говоря уже о реплицировании.

В трехуровневом приложении вся связь проходит через уровень приложения. Уровень представления и уровень данных не могут напрямую взаимодействовать друг с другом.

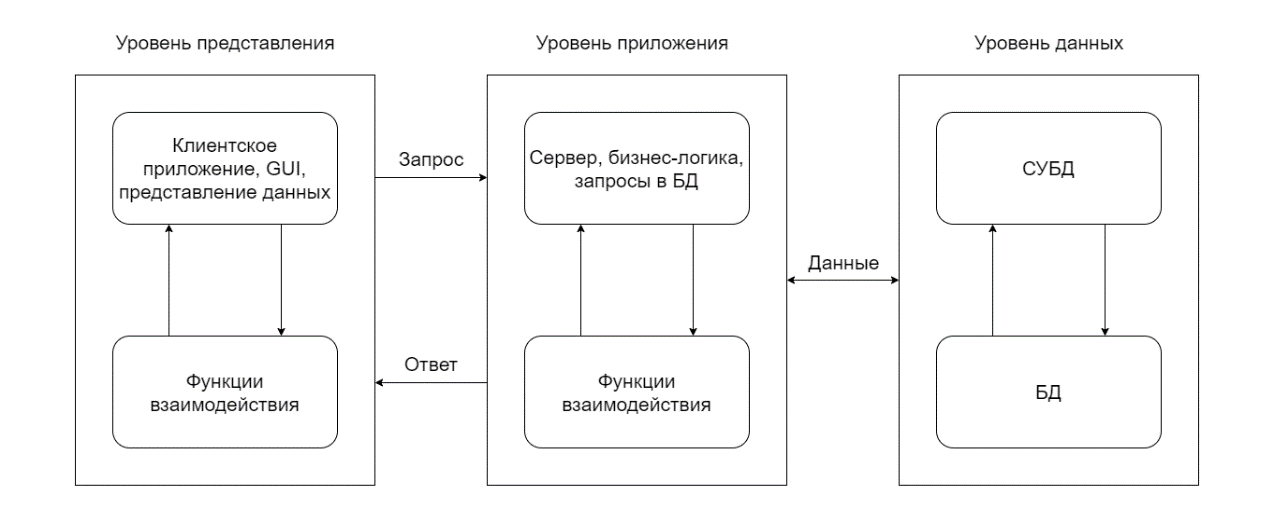


Рисунок 1 – Трехуровневая архитектура

**Уровень презентации**

Это пользовательский интерфейс и слой связи приложения, на котором конечный пользователь взаимодействует с приложением. Его основная цель – отображать информацию для и собирать информацию от пользователя. Уровень презентации может работать в веб-браузере, в качестве настольного приложения или графического пользовательского интерфейса (GUI).

**Уровень приложений**

Уровень приложений, также известный как уровень логики или средний уровень, является сердцем приложения. На этом уровне информация, собранная на уровне представления, обрабатывается с использованием бизнес-логики. Уровень приложений также может добавлять, удалять или изменять данные на уровне данных.

**Уровень данных**

Уровень данных, иногда называемый уровнем базы данных, уровнем доступа к данным или серверной частью, – это место, где хранится и управляется информация, обрабатываемая приложением.

**Зачем нужна трехуровневая архитектура в распределённых системах**

Как уже отмечалось ранее, главное преимущество трехуровневой архитектуры заключается в ее логическом и физическом разделении функций. Каждый уровень может работать на отдельной операционной системе и серверной платформе. Соответственно, не важно где будут расположены платформы: в облаке, в одном здании, на разных континентах. В любом случае части распределенной системы должны быть соединены надежными и защищенными линиями связи. Что касается скорости передачи данных, то она в значительной степени зависит от важности соединения между двумя частями системы с точки зрения обработки и передачи данных и в меньшей степени от их удаленности. Службы каждого уровня могут быть настроены и оптимизированы без влияния на другие уровни.

Другие преимущества:

* более быстрая разработка: отдельный уровень может разрабатываться одновременно разными командами, организация может быстрее выводить приложение на рынок, а программисты могут использовать новейшие и лучшие языки и инструменты в соответствии с каждым уровнем;
* улучшенная масштабируемость: отдельный уровень можно масштабировать независимо от других по мере необходимости;
* повышенная надежность: сбой на одном уровне с меньшей вероятностью повлияет на доступность или производительность других уровней;
* улучшенная безопасность: так как уровень представления и уровень данных не могут взаимодействовать напрямую, хорошо спроектированный уровень приложений может функционировать как своего рода внутренний брандмауэр, предотвращающий инъекции SQL и другие вредоносные эксплойты.

**Пример: трехуровневая архитектура в веб-разработке**

В веб-разработке уровни имеют разные названия, но выполняют схожие функции:

* веб-сервер является уровнем представления и предоставляет пользовательский интерфейс. Обычно это веб-страница или веб-сайт, но никто не запрещает использование мобильных или декстопных приложений. Этот уровень по существу взаимодействует с клиентом, отображая данные различного рода. Лишь немногие из логики приложения выполняются на стороне клиента, имея ограниченные возможности;
* сервер приложений соответствует среднему уровню, в котором размещена бизнес-логика, используемая для обработки пользовательских вводов. По сути, он играет роль объединения. При технической обработке различных входных данных и выборок, полученных клиентами, он играет роль взаимодействия с обширной базой данных, представленной на следующем уровне;
* сервер базы данных – это уровень данных или серверной части веб-приложения.

**Взаимодействия между уровнями в распределенных системах**

Существует два основных способа обмена данных между системами: синхронный способ и асинхронный (блокирующее и неблокирующее). Блокирующее – если участвующие стороны должны дождаться окончания взаимодействия, прежде чем приступить к следующей работе. Параллельная работа фрагментов систем никак не связана с асинхронностью и неблокирующими взаимодействиями. Сервер может работать параллельно со своими другими клиентами, обрабатывая запрос в синхронном режиме. Под синхронностью понимается работа запрашивающего клиента и процесса на сервере, который обрабатывает этот запрос. Синхронное взаимодействие – если работа клиента приостановлена, пока сервер обрабатывает запрос. Асинхронное взаимодействие – если вместо блокировки клиент выполняет другие действия после выдачи запроса.

**Постановка задачи**

Поскольку для трехуровневой архитектуры необходимо физическое разделение подсистем, то предлагается разработать трехслойное приложение: БД, сервер, приложение. В качестве приложения можно использовать: запросы в postman/insomnia/testmace, простой сайт, десктопное приложение, мобильное приложение. В качестве БД можно использовать: SQLite, PostgreSQL.

Нечетные варианты реализуют синхронное взаимодействие через API. Чётные варианты реализуют несохранное асинхронное взаимодействие через WebSockets.

В данной работе представлено несохранное асинхронное взаимодействие через WebSockets, представлением выступают запросы Postman, а база данных – PostgreSQL.

**Программный код**

Приложение представляет чат для распространения широковещательных сообщений. Каждый подписавшийся на рассылку клиент получает все отправленные за последние 10 секунд сообщения. После одноразовой отправки сообщение стирается из БД. Программный код представлен на листингах 1-7.

Листинг 1 – application.yaml

spring:  
 jpa:  
 database: POSTGRESQL  
 show-sql: true  
 hibernate:  
 ddl-auto: create-drop  
 datasource:  
 url: jdbc:postgresql://localhost:5432/architecture-db  
 username: postgres  
 password: password123  
 driver-class-name: org.postgresql.Driver

Листинг 2 – Main.java

import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class Main {  
  
 public static void main(String[] args) {  
 SpringApplication.run(Main.class, args);  
 }  
}

Листинг 3 – ChatMessage.java

import jakarta.persistence.Entity;  
import jakarta.persistence.GeneratedValue;  
import jakarta.persistence.GenerationType;  
import jakarta.persistence.Id;  
import lombok.AllArgsConstructor;  
import lombok.Builder;  
import lombok.Getter;  
import lombok.NoArgsConstructor;  
import lombok.Setter;  
import lombok.ToString;  
  
@ToString  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
@Getter  
@Setter  
@Entity  
public class ChatMessage {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.IDENTITY)  
 private Long id;  
  
 private String senderSessionId;  
 private String creationTimestamp;  
 private String content;  
 private Boolean isSent;  
}

Листинг 4 – ChatMessageRepository.java

import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.stereotype.Repository;  
  
@Repository  
public interface ChatMessageRepository extends JpaRepository<ChatMessage, Long> {  
  
 void deleteAllByIsSent(Boolean isSent);  
}

Листинг 5 – WebSocketConfig.java

import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.web.socket.WebSocketHandler;  
import org.springframework.web.socket.config.annotation.EnableWebSocket;  
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;  
import org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;  
  
@Configuration  
@EnableWebSocket  
public class WebSocketConfig implements WebSocketConfigurer {  
  
 private final WebSocketHandler webSocketHandler;  
 public WebSocketConfig(@Qualifier("serverWebSocketHandler") WebSocketHandler webSocketHandler) {

Продолжение листинга 5

this.webSocketHandler = webSocketHandler;  
 }  
  
 @Override  
 public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {  
 registry.addHandler(webSocketHandler, "/websocket");  
 }  
}

Листинг 6 – ChatMessageService.java

import lombok.RequiredArgsConstructor;  
import org.springframework.stereotype.Service;  
import org.springframework.transaction.annotation.Transactional;  
  
import java.util.List;  
  
@RequiredArgsConstructor  
@Service  
public class ChatMessageService {  
  
 private final ChatMessageRepository chatMessageRepository;  
  
 public void save(ChatMessage chatMessage) {  
 chatMessageRepository.save(chatMessage);  
 }  
  
 public List<ChatMessage> findAll() {  
 return chatMessageRepository.findAll();  
 }  
  
 @Transactional  
 public void clearSentMessages() {  
 chatMessageRepository.deleteAllByIsSent(true);  
 }  
  
 public void saveAllAndFlush(List<ChatMessage> messages) {  
 chatMessageRepository.saveAllAndFlush(messages);  
 }  
}

Листинг 7 – ChatMessageService.java

import lombok.RequiredArgsConstructor;  
import org.springframework.scheduling.annotation.EnableScheduling;  
import org.springframework.scheduling.annotation.Scheduled;  
import org.springframework.stereotype.Component;  
import org.springframework.web.socket.CloseStatus;  
import org.springframework.web.socket.TextMessage;  
import org.springframework.web.socket.WebSocketSession;  
import org.springframework.web.socket.handler.TextWebSocketHandler;  
  
import java.io.IOException;  
import java.text.SimpleDateFormat;  
import java.util.Date;  
import java.util.List;  
import java.util.Queue;  
import java.util.concurrent.ConcurrentLinkedQueue;  
  
@RequiredArgsConstructor  
@EnableScheduling  
@Component  
public class ServerWebSocketHandler extends TextWebSocketHandler {  
  
 private final ChatMessageService chatMessageService;

Продолжение листинга 7

private final Queue<WebSocketSession> sessions = new ConcurrentLinkedQueue<>();  
  
 @Override  
 public void handleTextMessage(WebSocketSession session, TextMessage message) {  
 String now = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Date());  
  
 chatMessageService.save(  
 ChatMessage.builder()  
 .senderSessionId(session.getId())  
 .content(message.getPayload())  
 .creationTimestamp(now)  
 .isSent(false)  
 .build());  
 }  
  
 @Scheduled(fixedDelay = 10000)  
 void sendPeriodicMessages() {  
 chatMessageService.clearSentMessages();  
  
 List<ChatMessage> messages = chatMessageService.findAll();  
 if (messages.isEmpty()) return;  
 String messagesBatch = getConcatenatedMessage(messages);  
  
 sessions.stream()  
 .filter(WebSocketSession::isOpen)  
 .forEach(webSocketSession -> {  
 try {  
 webSocketSession.sendMessage(new TextMessage(messagesBatch));  
 } catch (IOException e) {  
 //TODO: logger.log(e);  
 }  
 });  
  
 messages.forEach(chatMessage -> chatMessage.setIsSent(true));  
 chatMessageService.saveAllAndFlush(messages);  
 }  
  
 private String getConcatenatedMessage(Iterable<ChatMessage> messages) {  
 StringBuilder stringBuilder = new StringBuilder();  
 messages.forEach(m -> stringBuilder  
 .append(m.toString())  
 .append("\n"));  
  
 return stringBuilder.toString();  
 }  
  
 @Override  
 public void afterConnectionEstablished(WebSocketSession session) throws Exception {  
 sessions.add(session);  
 super.afterConnectionEstablished(session);  
 }  
  
 @Override  
 public void afterConnectionClosed(WebSocketSession session, CloseStatus status) throws Exception {  
 sessions.remove(session);  
 super.afterConnectionClosed(session, status);  
 }  
}

**Вывод программы**

На рисунке 2 можно наблюдать запуск сервера и выполнение SQL-запросов в базу данных. На рисунках 3-4 представлена работа приложения на примере двух клиентов. Сначала оба выполняют подключение к сервера, затем свое сообщение отправляет первый клиент. Его получают оба клиента. После чего своё сообщение отравляет второй клиент, и оно так же приходит всем клиентам с установленным соединением.

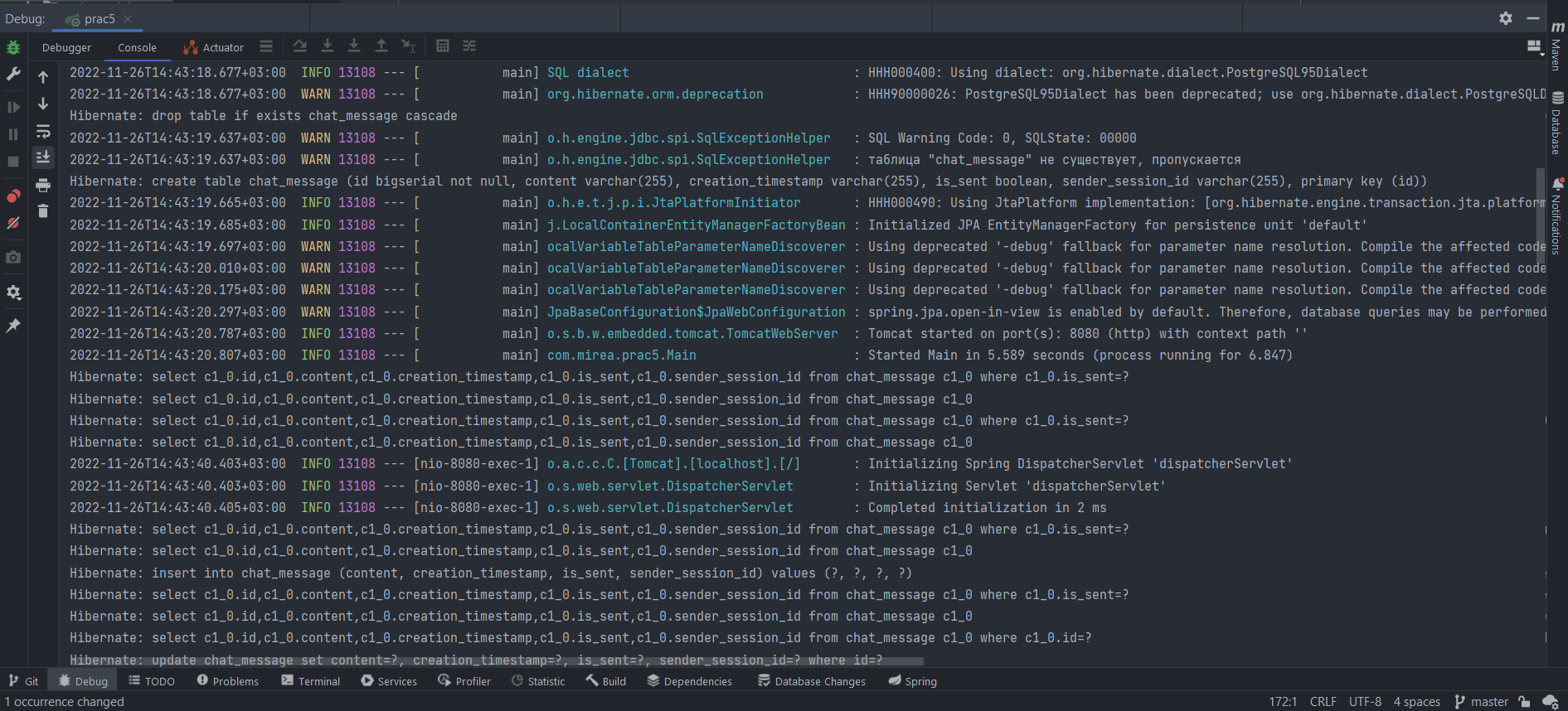


Рисунок 2 - Успешная работа сервера и выполнения запросов к БД

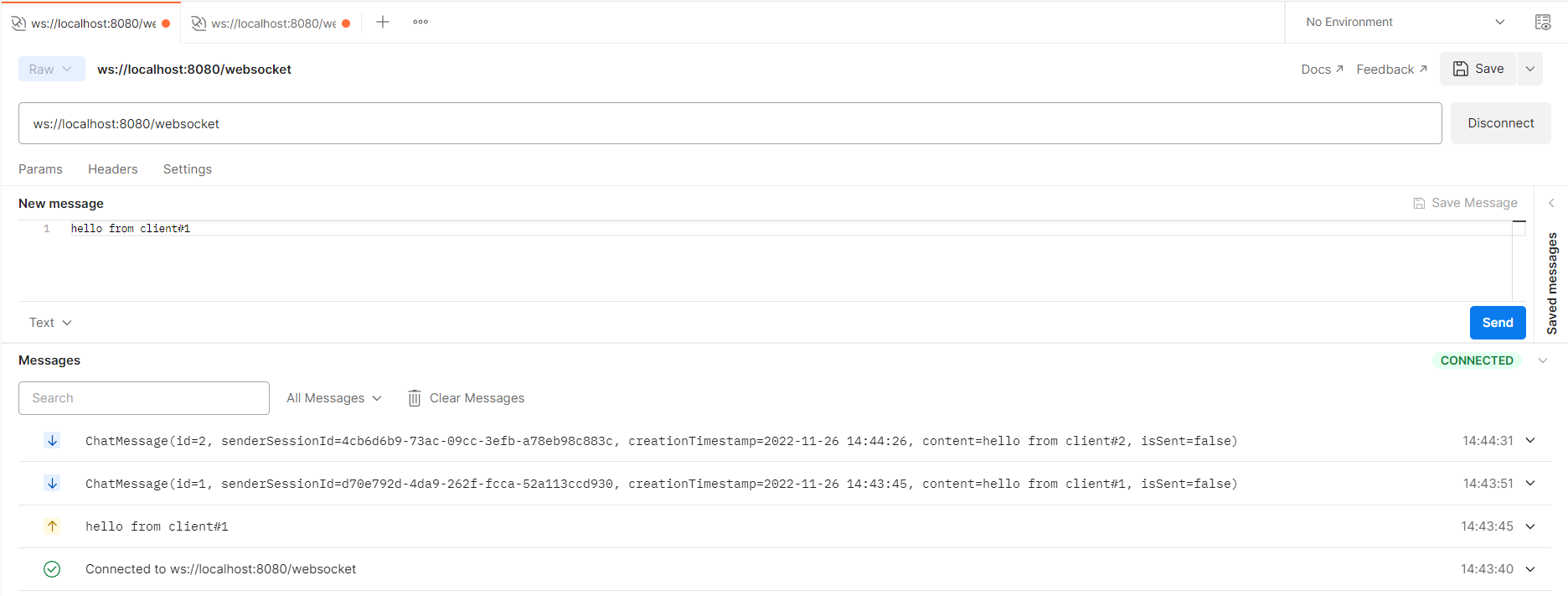


Рисунок 3 – Асинхронное взаимодействие с точки зрения первого клиента

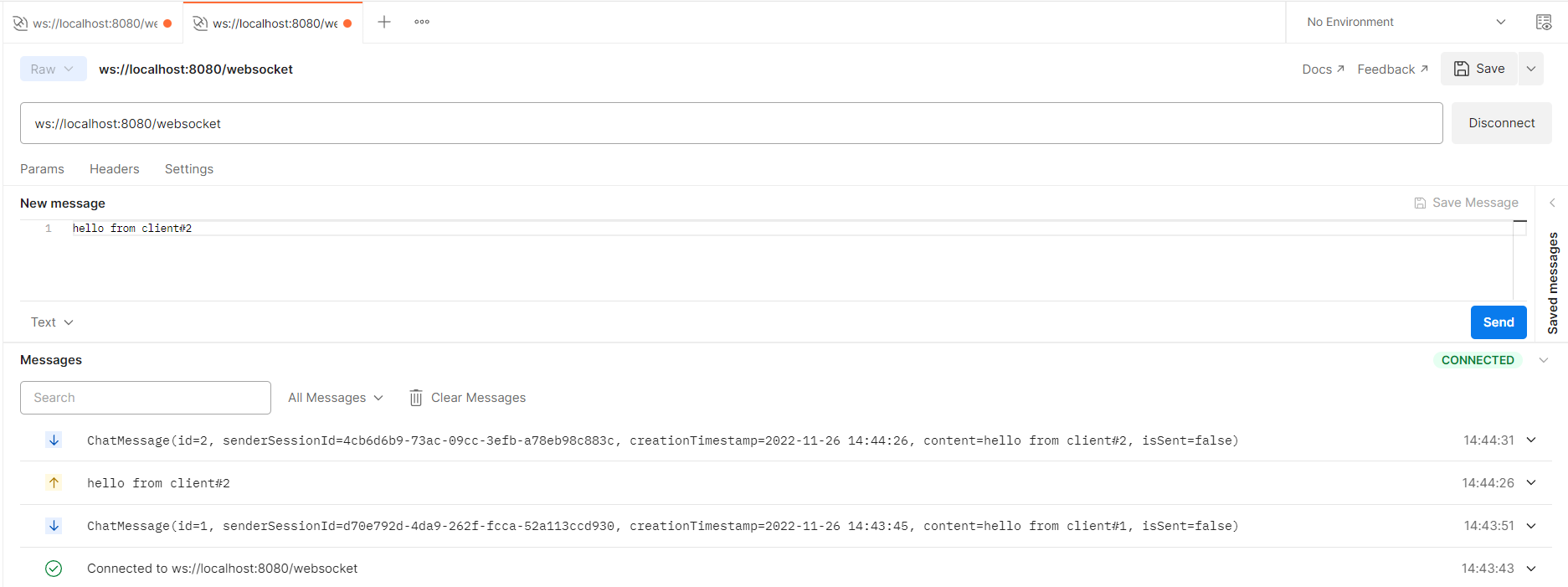


Рисунок 4 – Асинхронное взаимодействие с точки зрения второго клиента

**Вывод**

В результате выполнения данной практической работы было реализовано асинхронное несохранное приложение на WebSockets, спроектированное по многослойной клиент-серверной архитектуре, включающей в себя 3 слоя – БД (PostgreSQL), представление (Postman) и приложение (Java Spring WebSockets).