Санкт-Петербургский Государственный Университет
Математико-механический факультет
Математическое обеспечение и администрирование
информационных систем

Винник Екатерина Петровна

Клиент-серверный чат

Отчет по учебной практике

Научный руководитель:
к.т.н., доцент Брыксин Т.А.

Санкт-Петербург
2020

Содержание

Введение

Знание технологий и умение использовать их в проекте очень важно в настоящее время, так как переиспользование существующих решений ускоряет разработку и обобщает код программного продукта. Использование уже готового инструмента для решения поставленной задачи, как правило, предоставляет большую функциональность и документацию, чем в случае написания своего собственного аналога. Перечисленное в дальнейшем облегчит поддержку кода разработчикам, работающим над этим же продуктом, а также упростит знакомство с ним тем, кто хотел бы использовать этот продукт в своих проектах.

Одной из ключевых технологий, используемых при разработке на Java является фреймворк Spring. Согласно статистике JetBrains [1], для решения задачи написания сервера какого-либо приложения на языке Java, воспользоваться именно этим инструментом предпочитает 61% разработчиков.

Другим важным программным решением, использующимся не только среди Java разработчиков, является Docker. Согласно статистике Google Trends [2], популярность этой технологии возрастала с начала рассматриваемого периода и на момент запроса достигает абсолютного максимума.

Также для того, чтобы использовать более подходящие инструменты при решении конкретной задачи, разработчику важно видеть ее как часть проекта, над которым ведется работа – часто как часть взаимодействия интерфейса приложения и его низкоуровневой реализации. Для этого важно иметь представление об используемых технологиях в разных направлениях разработки.

Клиент-серверный веб-чат, использующий Spring и Docker, имеющий программный интерфейс – страницу в браузере, которая взаимодействует с внутренней системой приложения, помог бы в освоении этих технологий и формировании представления о программе как о совокупности взаимодействующих между собой подпрограмм.

Постановка задачи

Целью данного проекта является освоение технологий Spring и Docker посредством написания клиент-серверного веб-чата.

Для достижения этой цели были поставлены следующие задачи:

- провести обзор технологий создания программных интерфейсов для реализации клиентской части;
- выбрать базу данных для хранения метаданных пользователей;
- реализовать сервер;
- реализовать клиент;
- установить взаимодействие между клиентом и сервером;
- выполнить апробацию.

1. Обзор инструментов, используемых в приложении

1.1. Обзор технологий для создания пользовательского интерфейса клиентской части

На данный момент существует два весьма распространенных инструмента для разработки программных интерфейсов – React и Angular.

Согласно [3], Angular является фреймворком, разработанным Google. Инструмент использует статически типизированный язык TypeScript. Так как это фреймворк, его использование требует соблюдения определенной унифицированной структуры проекта, что упрощает разработку программного интерфейса приложения. Веб-приложение, разработанное на Angular, состоит из взаимодействующих между собой компонентов, каждый из которых отвечает за некоторую функциональность и может быть переиспользован в другой подсистеме приложения. Angular был создан как инструмент разработки программных интерфейсов для разработчиков со знанием объектно-ориентированной парадигмы программирования. Возможным минусом Angular является то, что при отрисовке элементов HTML страницы используется Document Object Model. Document Object Model (DOM) – объектная модель документа, которая создается браузером для представления HTML страницы по ее исходному коду. Любое изменение страницы (т.е., изменение какого-либо элемента ее объектной модели) может привести к обновлению всей структуры DOM, и если объектная модель страницы содержит в себе много элементов, каждый из них потребуется отрисовать заново. Это занимает значительное время, и, следовательно, увеличивает время отклика страницы приложения в браузере.

В отличие от Angular [3], React является JavaScript библиотекой, поддерживаемой Facebook. Благодаря тому, что данное решение использует JavaScript, его легко интегрировать в уже существующие JavaScript проекты. Так как это библиотека, ее использование не требует соблюдения определенной структуры проекта (как если бы это был фреймворк), но подобная гибкость обеспечивается большими сложностями в написа-

нии кода, а также его отладки. Плюсом использования React является наличие Virtual DOM – способа оптимизации обновления элементов страницы, эффективного в случае если DOM содержит много элементов и элементы страницы должны часто изменяться. При изменении какого-то элемента HTML страницы Virtual DOM вычисляет наименее затратный (требующий обновления минимального числа элементов) способ обновления DOM, после чего обновляет элементы DOM в соответствии с найденным способом. Подобная стратегия значительно уменьшает время отклика страницы в браузере.

В таком приложении как чат, пользователь работает с окном диалога. Отправление сообщений в чате не может быть причиной продолжительного обновления дерева элементов страницы, так как сама страница диалога, как правило, не обладает сложной структурой DOM, и, следовательно, изменения в ней не требуют много времени на его обновление. Поэтому проблема обновления элементов HTML страницы в чате не актуальна, т.е., отсутствие технологии Virtual DOM у Angular несущественно. Кроме того, Angular, благодаря четкой, определенной структуре проекта и использованию статически типизированного языка Typescript, более прост в освоении для разработчика, знакомого с объектно-ориентированным программированием, чем React, поэтому инструментом для создания программного интерфейса приложения был выбран он.

1.2. Выбор базы данных

В чате между клиентом и сервером предполагается постоянный обмен данными – например, клиентское приложение может посылать на сервер объект зарегистрированного пользователя с целью сохранения его в базе данных, или какое-либо отправленное пользователем сообщение. Extensible Markup Language (XML) плохо подходит для таких задач, так как XML документ более тяжеловесен, чем JSON (JavaScript Object Notation) объект. Также данные, пришедшие в XML, требуют больше времени для парсинга в случае необходимости дальнейшего использования их в приложении [4]. Поэтому в проекте было решено использовать

JSON. В связи с принятым решением, хранить объекты в базе данных в каком-то ином формате могло бы быть не очень удобно – любое сообщение в таком случае необходимо передавать в формате JSON, а после, при сохранении в базе данных, преобразовывать в другой формат. Преобразование форматов передаваемых объектов увеличивает время отклика приложения.

Кроме того, в случае запаковывания базы данных в Docker контейнер, не нужно отдельно инициализировать NoSQL базу данных [5], в отличие от SQL базы [6], где нужно создавать специальный скрипт, декларирующий структуру таблиц.

Также используемая база данных должна иметь потенциальную возможность хранить текст сообщений чата. Так как текст сообщения может содержать разные объекты, например, эмодзи или картинки, его проще хранить как неструктурированную информацию.

На основании обозначенных проблем были сформированы следующие требования к базе данных:

- хранение объектов в формате JSON;
- простота запаковки;
- возможность хранить неструктурированную информацию.

Всем перечисленным требованиям удовлетворяет MongoDB. Согласно документации [7], MongoDB хранит объекты в формате BSON, что позволяет легко сохранять информацию в формате JSON в базу данных. Также MongoDB позволяет хранить неструктурированную информацию [8] и является NoSQL базой данных, позволяя не инстанцировать схему базы данных при запаковке в контейнер.

2. Архитектура решения

2.1. Основные компоненты архитектуры

В приложении есть два сервера: сервер для работы с базой данных, и сервер, поддерживающий функционирование клиентской части (веб-приложения). Плюсом наличия двух независимых друг от друга серверов в приложении является то, что возможная поломка сервера, взаимодействующего с базой данных, отразится на пользователе опосредованно – он получит соответствующее уведомление о поломке на странице в браузере от веб-приложения, которое продолжит функционировать благодаря клиентскому серверу. Взаимодействие клиентской и серверной части реализовано с учетом Representational State Transfer (REST) [9].

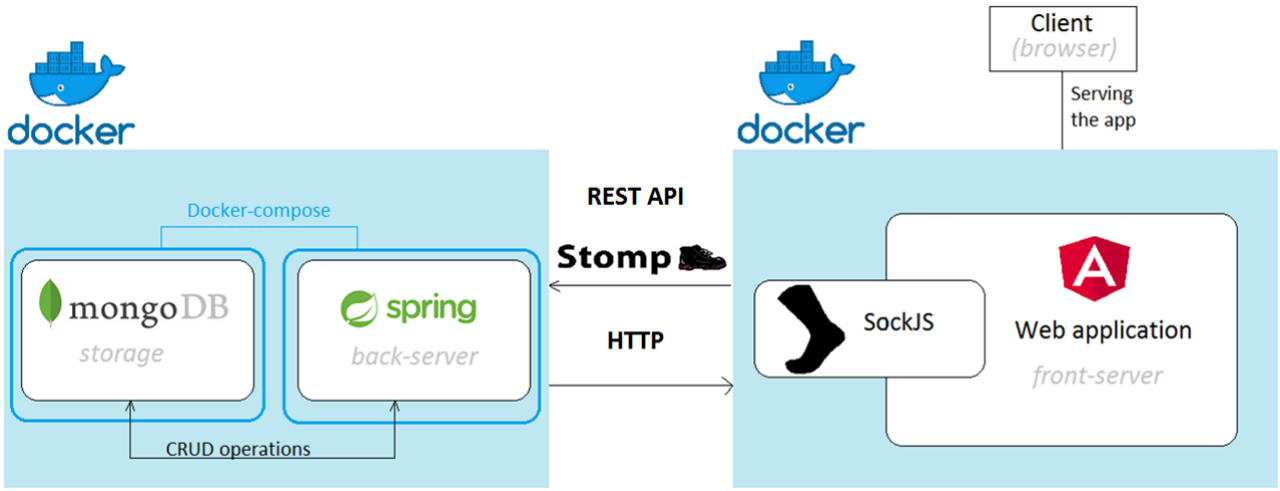

Рисунок 2.1 — Архитектура приложения

На рисунке 2.1 представлена схема взаимодействия сервера, работающего с базой данных (на рисунке: "back-server"), и сервера, обеспечивающего функционирование клиентской части (на рисунке: "front-server"). Опишем более подробно сценарий их совместной работы. Пользователь взаимодействует со страницей в браузере – заполняет поля в HTML странице и нажимает различные кнопки: отправки сообщения, регистрации и т.д. Клиентская часть – веб-приложение обрабатывает действия пользователя и, в случае необходимости, отправляет запросы серверу, взаимодействующему с базой данных, по REST API. Он производит операции с хранящейся в нем информацией и отправляет ответ веб-приложению –

код возврата проведенной операции или JSON-объект, соответствующий бизнес-логике приложения.

Оба компонента приложения – клиент и сервер, запакованы в соответствующие Docker-контейнеры. Для того, чтобы не запускать отдельно MongoDB, был реализован одновременный запуск двух Docker-образов бэк-сервера и MongoDB через docker-compose.

2.2. Сценарий регистрации пользователя

В приложении была реализована регистрация пользователей. Все пользователи чата имеют уникальный никнейм.

Рассмотрим процесс регистрации пользователя на рисунке 2.2. Когда он заполняет данные регистрации – пароль, никнейм и остальные параметры, веб-приложение инициализирует ими объект User и отправляет его через POST запрос в формате JSON на сервер.

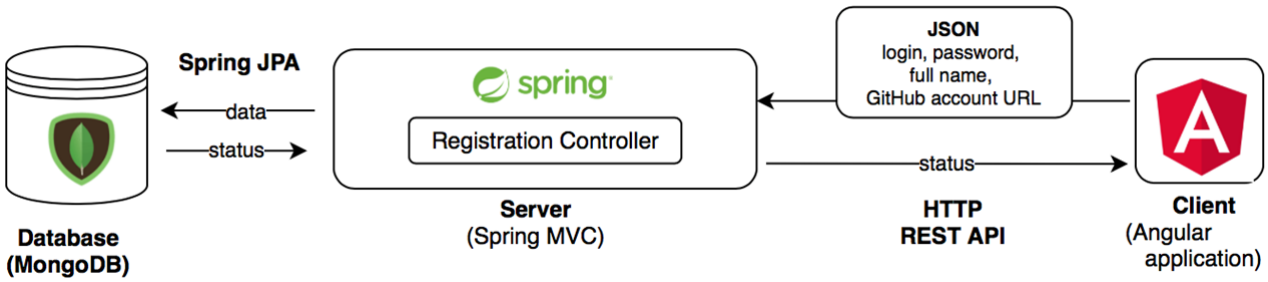

Рисунок 2.2 — Сценарий регистрации пользователя

В приложении присутствует два этапа валидации данных: валидация на клиенте – проверка, что все необходимые поля были заполнены, валидация на сервере – проверка того, что каждый пользователь должен иметь уникальный никнейм. Если валидация на клиенте и сервере прошла успешно, данные пользователя сохраняются в базу, и он считается зарегистрированным в приложении. Сервер возвращает веб-приложению код возврата об успешно проведенной операции добавления пользователя. Веб-приложение, получив этот код, открывает зарегистрированному пользователю окно диалога.

Также стоит отметить, что пользователь может заполнить опциональное поле аватара, выбрав желаемую картинку на своем компьютере.

В таком случае на сервер будет отправлена картинка в качестве JSON-объекта, которая проинициализирует аватар пользователя.

При сохранении пароль пользователя зашифровывается с помощью алгоритма Bcrypt, поддерживаемого библиотекой PasswordEncoder в Spring. Этот алгоритм является алгоритмом медленного хеширования – это значит, что процесс хеширования каждого пароля занимает очень много времени. При использовании медленного хеширования, даже если возможные злоумышленники смогли угадать используемую хеш-функцию, процесс подбора пароля пользователя путем перебора займет огромное количество времени, и, следовательно, не станет подходящим решением для атакующих. Также алгоритм использует соль в качестве защиты от атак с помощью радужных таблиц [10]. На данный момент Bcrypt считается наиболее мощным алгоритмом для шифрования паролей в Spring, остальные варианты являются устаревшими [11].

2.3. Сценарий обмена сообщениями между клиентами

В приложении поддерживается два типа сообщений – общие (сообщения, которые видят все пользователи в чате) и личные (сообщения, которые видит один пользователь). Обмен сообщениями между клиента-

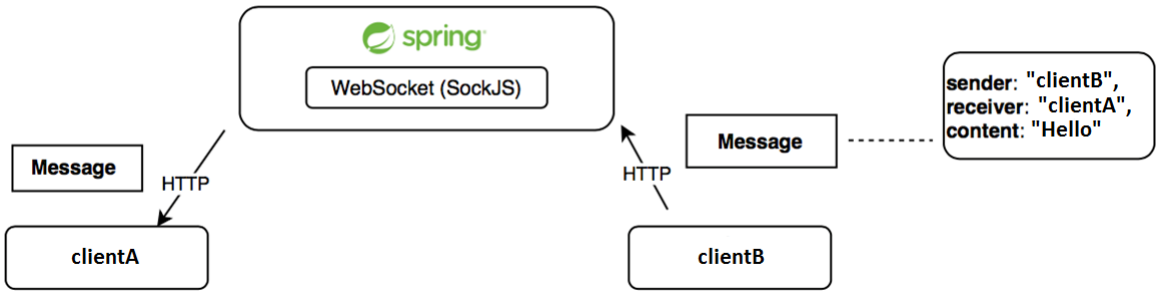

Рисунок 2.3 — Сценарий обмена сообщениями

ми происходит через сервер. При установлении соединения с сервером все клиенты подписываются на конечную точку приема сообщений, видных всем пользователям, и на персональную конечную точку (индивидуальную для каждого пользователя) для получения личных сообщений.

На сервере также есть два вида конечных точек: для получения личных сообщений клиентов и получения общих сообщений клиентов. В

зависимости от вида сообщения, клиент отправляет сообщение серверу на соответствующую конечную точку приема сообщений.

В случае, если клиент хочет послать личное сообщение, он указывает получателя в соответствующем поле страницы браузера. Веб-приложение отправляет сообщение на сервер, который перенаправляет пришедшее сообщение получателю на его личную конечную точку.

Если клиент хочет отправить общее сообщение, он оставляет поле получателя пустым. Сервер, которому пришло сообщение, не имеющее получателя, посылает его на единую для всех клиентов конечную точку приема общих сообщений.

2.4. Архитектура клиентской части приложения

В ходе работы над клиентской частью была разработана архитектура, представленная на рисунке 2.4. Точкой входа в веб-приложение является файл app.component – он декларирует компонент приложения и является обработчиком действий, производимых с HTML страницей, например, нажатие на кнопку Login при регистрации, заполнение текста сообщения в диалоге и т.д. Обработку некоторых действий он перепо-

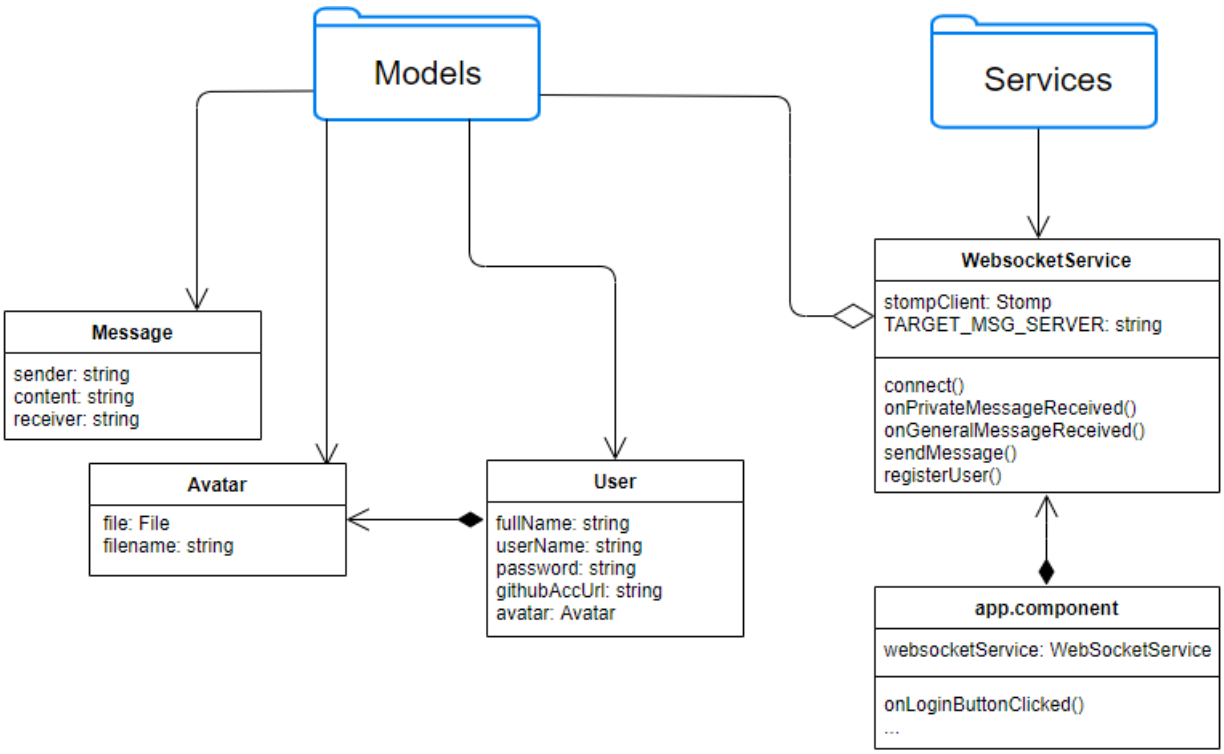

Рисунок 2.4 — Архитектура клиентской части приложения

ручает классу WebsocketService, в котором сосредоточена бизнес-логика

приложения. Он является классом, ответственным за общение с сервером, поддерживающим функционирование базы данных, т.е., устанавливает с ним вебсокетное соединение по имеющемуся адресу (константное значение TARGET_MSG_SERVER) и обменивается данными, которые собрал app.component. Например, WebsocketService отправляет данные зарегистрированного пользователя на сервер для сохранения в базу данных, а также отправляет и получает различные виды клиентских сообщений. Данный класс не работает с данными пользователя напрямую – он использует абстрактные модели данных приложения, такие как User и Message, лежащие в папке Models. Инициализацией упомянутых моделей данными, собранными из HTML страницы, занимается app.component.

2.5. Архитектура серверной части приложения

Обобщенная архитектура серверной части приложения представлена на рисунке 2.5. Точкой входа в приложение является класс MessagingStompWebsocketApplication. Архитектура сервера разбивается на три подмодуля: пакет Persistence, пакет Config и подмодуль, включающий в себя два класса, ответственные за обмен сообщениями. MessageController ответственен за получение сообщений от клиентов и пе-

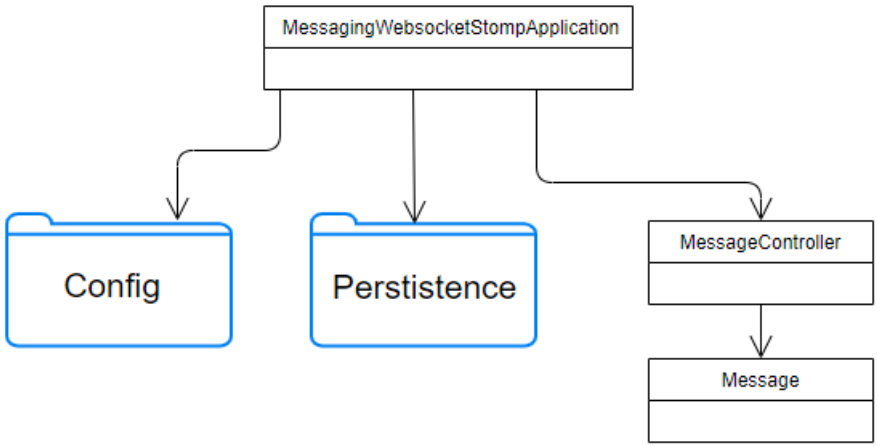

Рисунок 2.5 — Обобщенная архитектура серверной части

ренаправление адресатам в соответствии с указанными конечными точками. Класс Message является классом пересылаемого сообщения, аналогичным классу Message клиента. В подмодуле Config хранятся настройки сервера: например, от каких отправителей ему разрешено обрабатывать запросы. Также там хранятся некоторые настройки шифрования

паролей, а именно, настройки используемого алгоритма Bcrypt. Подмодуль Perstistence рассмотрим более подробно на рисунке 2.6. Для общения с клиентом реализовано три REST-контроллера – UserController, LoginController, RegistrationController. С помощью них сервер обменивается данными с клиентом. Для валидации данных при регистрации, или попытке войти в систему уже зарегистрированного пользователя используется класс UserValidator. UserRepository предоставляет необходимые запросы для работы с базой данных, в которой хранятся объекты User. Например, метод findUserByNickname() в UserRepository нахо-

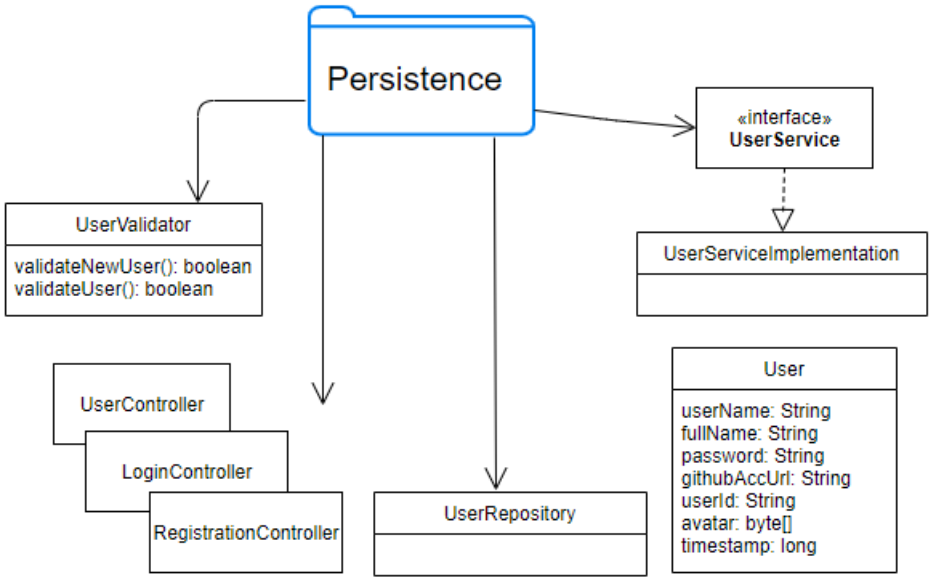

Рисунок 2.6 — Подмодуль хранения данных

дит всех пользователей в базе, у которых никнейм совпадает с данным. UserService – сервис для работы с базой, который взаимодействует с более низким уровнем абстракции – UserRepository.

3. Апробация

В ходе данной работы в приложении была реализована возможность обмена личными, общими сообщениями и хранение пользователей. Рассмотрим разные аспекты работы программы на рисунках.

На рисунке 3.1 изображена регистрационная форма пользователя. При регистрации клиент подписывается на общую и уникальную конечные точки для получения сообщений. На рисунке 3.2 приведена соответ-

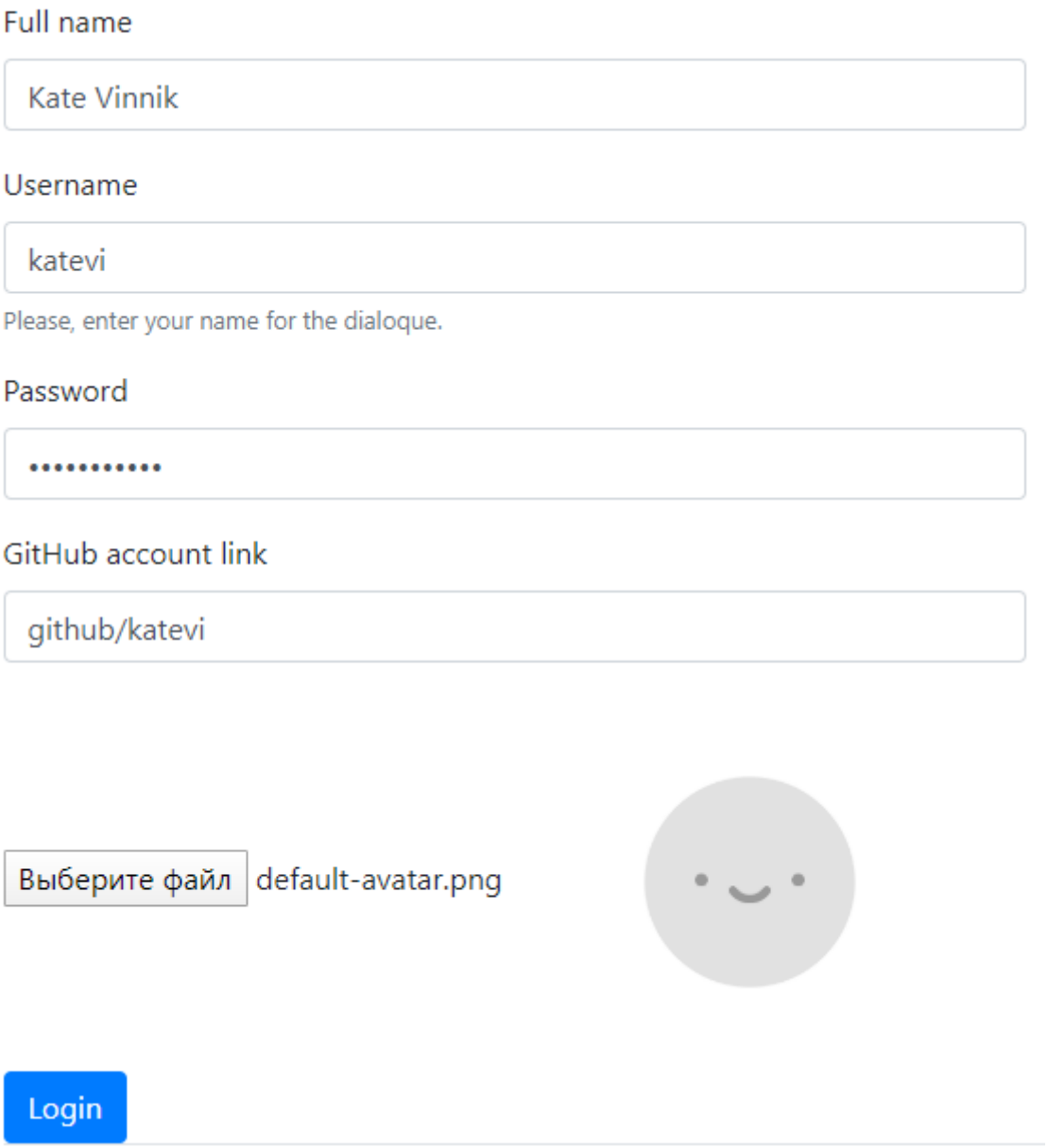

Рисунок 3.1 — Регистрационная форма приложения

ствующая запись в логах, иллюстрирующая подписку пользователя при регистрации на конечную точку, на которую будут приходить сообще-

ния, видные всем пользователям – "/topic/publishedMessages" и конечную точку, использующую уникальный никнейм зарегистрированноого пользователя "/user/katevi/" – на нее будут приходить личные сообще-

```
>>> SUBSCRIBE
id:sub-0
destination:/topic/publishedMessages
```

```
>>> SUBSCRIBE
id:sub-1
destination:/user/katevi/
```

Рисунок 3.2 — Подписка на конечные точки для получения личных и общих сообщений

ния, адресованные данному пользователю.

После прохождения регистрации пользователю доступно окно диалога с формой для отправления сообщения. На рисунке 3.3 отображе-

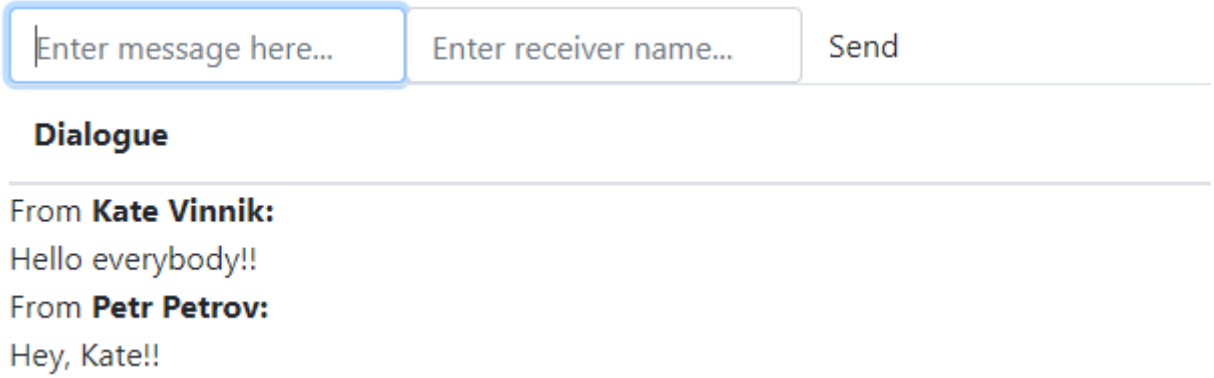

Рисунок 3.3 — Окно диалога

ны два сообщения: первое из них является сообщением, видным всем пользователям. На рисунке 3.4 продемонстрирована публикация полу-

```
<<< MESSAGE
destination:/topic/publishedMessages
content-type:application/json
subscription:sub-0
message-id:0drmk05v-0
content-length:70

{"content":"Hello everybody!!","sender":"Kate Vinnik","receiver":null}
```

Рисунок 3.4 — Получение общего сообщения

ченного сервером сообщения от пользователя "Kate Vinnik"на общую

для всех клиентов конечную точку: "/topic/publishedMessages" и захват полученных данных клиентом "Kate Vinnik". Так как сообщение получают все пользователи, в частности, его получает и клиент-отправитель сообщения, т.е., в рассматриваемом случае, его получает и клиент "Kate Vinnik". Получатель сообщения не указан, так как сообщение не является личным. Сообщение, посланное клиентом "Petr Petrov"пользователю "Kate Vinnik", является личным.

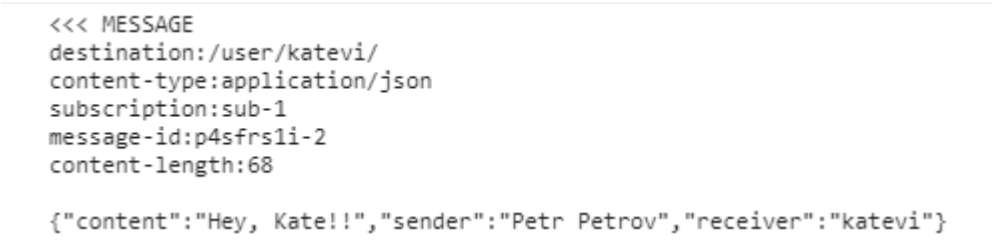

Рисунок 3.5 — Получение личного сообщения

На рисунке 3.5 продемонстрировано пришедшее пользователю "Kate Vinnik" сообщение от пользователя "Petr Petrov". Сообщение получено по уникальной для пользователя конечной точке получения личных сообщений – "user/katevi/".

Заключение

В ходе работы над данным проектом были достигнуты следующие результаты:

- реализована серверная часть сообщения с помощью Spring;
- выбрана база данных для хранения пользователей;
- реализована клиентская часть с помощью Angular;
- настроено взаимодействие клиента и сервера с помощью обмена сообщениями в JSON-формате;
- реализована возможность обмена личными и общими сообщениями;
- произведена апробация приложения.

Код проекта доступен на *GitHub* [1].

[1] `https://github.com/katevi/Gitlogue`

Список использованных источников

[1] JetBrains statistics. Spring framework usage, – URL: https://www.jetbrains.com/lp/devecosystem-2019/java/ (дата обращения 2020-05-15).

[2] Google Trends. Docker usage, – URL: https://trends.google.ru/trends/explore?date=today%205-y&q=%2Fm%2F0wkcjgj (дата обращения 2020-05-15).

[3] Angular vs React, – URL: https://programmingwithmosh.com/react/react-vs-angular/, (дата обращения 2020-05-24).

[4] JSON for the web data interchange, – URL: https://hackr.io/blog/json-vs-xml, (дата обращения 2020-06-01).

[5] Medium. MongoDB on Docker container, – URL: https://medium.com/@nashaad/mongodb-on-docker-a-barebones-tutorial-720d09e169c3 (дата обращения 2020-05-18).

[6] Medium. Customize your MySQL Database in Docker, – URL: https://medium.com/better-programming/customize-your-mysql-database-in-docker-723ffd59d8fb (дата обращения 2020-05-18).

[7] MongoDB documentation. JSON and BSON, – URL: https://www.mongodb.com/json-and-bson, (дата обращения 2020-05-24).

[8] MongoDB. Unstructured data, – URL: https://www.mongodb.com/unstructured-data, (дата обращения 2019-12-13).

[9] REST, – URL: https://ru.wikipedia.org/wiki/REST, (дата обращения 2020-06-01).

[10] Salted Password Hashing. Doing it Right, – URL: https://crackstation.net/hashing-security.htm, (дата обращения 2020-05-28).

[11] Baeldung. Password Encoding with Spring, – URL: https://www.baeldung.com/spring-security-registration-password-encoding-bcrypt, (дата обращения 2020-05-25).