Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное

учреждение высшего образования

Пермский национальный исследовательский политехнический университет

(ПНИПУ)

Факультет: Электротехнический (ЭТФ)

Направление: 09.03.04 – Программная инженерия (ПИ)

Профиль: Разработка программно-информационных систем (РИС)

Кафедра информационных технологий и автоматизированных систем (ИТАС)

Допустить к защите

Зав. кафедрой ИТАС: д-р экон. наук, проф.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Р.А. Файзрахманов

«\_\_\_\_\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

на соискание академической степени бакалавра

на тему

**«Разработка встраиваемого веб-чата поддержки пользователей на базе платформы Rocket.Chat»**

|  |  |  |
| --- | --- | --- |
| Студент: |  | Ширинов Рустам Рамазанович |
|  | (подпись, дата) |  |
| Группа: | РИС-17-1б | |

Состав ВКР:

1. Пояснительная записка на 114 стр.
2. Приложения на 9 стр.

|  |  |  |
| --- | --- | --- |
| Руководитель ВКР: |  | канд. техн. наук, доцент каф. ИТАС  О.А. Полякова |
| (подпись, дата) |
| Консультант по предметной области: |  | руководитель отдела разработки чат-платформы А.С. Казаринов |
| (подпись, дата) |
| Прохождение предзащиты: |  | канд. техн. наук, доцент каф. ИТАС  О.А. Полякова |
| (подпись, дата) |
| Прохождение нормоконтроля: |  | заведующий лабораториями  Д.А. Карлов |
| (подпись, дата) |
| Прохождение проверки на объем заимствований: |  | инженер  Е.В. Ерискина |
|  | (подпись, дата) |  |

Пермь — 2021Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное

учреждение высшего образования

Пермский национальный исследовательский политехнический университет

(ПНИПУ)

Факультет: Электротехнический (ЭТФ)

Направление: 09.03.04 – Программная инженерия (ПИ)

Профиль: Разработка программно-информационных систем (РИС)

Кафедра информационных технологий и автоматизированных систем (ИТАС)

УТВЕРЖДАЮ

Зав. кафедрой ИТАС: д-р экон. наук, проф.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Р.А. Файзрахманов

«\_\_\_\_\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

**ЗАДАНИЕ**

на выполнение выпускной квалификационной работы бакалавра

Фамилия, имя, отчество: Ширинов Рустам Рамазанович

Группа: РИС-17-1б

Начало выполнения работы: 01.02.2021

Контрольные сроки просмотра работы кафедрой:

1) 09.04.21, 2) 23.04.21, 3) 14.05.21

Защита работы на заседании ГЭК: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Наименование темы: «Разработка встраиваемого веб-чата поддержки пользователей на базе платформы Rocket.Chat».

2. Исходные данные к работе: техническое задание на разработку встраиваемого веб-чата поддержки пользователей на базе платформы «Rocket.Chat».

3. Содержание пояснительной записки:

а) исследовательский раздел:

анализ достоинств и недостатков встраиваемых веб-чатов, используемых компанией на текущий момент; исследование устройства и функциональных возможностей Rocket.Chat LiveChat API; анализ разработанного компанией дизайна; исследование подходов к разработке встраиваемых веб-чатов; требования к реализуемому встраиваемому веб-чату.

б) конструкторский раздел:

выбор средств разработки для реализации встраиваемого веб-чата; проектирование архитектуры встраиваемого веб-чата; проектирование и разработка алгоритмов работы веб-чата.

в) технологический раздел:

настройка окружения разработки; реализация интерфейсов взаимодействия с «Rocket.Chat LiveChat API»; реализация хранилища для управления состоянием приложения; реализация пользовательского интерфейса; реализация механизма кроссдоменного обмена данными между фреймами; реализация Iframe-приложения; реализация скрипта загрузки Iframe-приложения.

г) экономический раздел:

технико-экономическое обоснование проекта, расчет затрат на разработку и ввод в эксплуатацию, расчет экономического эффекта.

д) обеспечение информационной безопасности:

описание используемых средств защиты информации.

4. Дополнительные указания:

Дополнительных указаний нет.

5. Основная литература:

a) REST API - Rocket.Chat Developer. [Электронный ресурс] – URL: https://developer.rocket.chat/api/rest-api/ (дата обращения 27.04.2021).

б) Realtime API - Rocket.Chat Developer. [Электронный ресурс] – URL: https://developer.rocket.chat/api/realtime-api (дата обращения 27.04.2021).

в) Тег <iframe> | htmlbook.ru. [Электронный ресурс] – URL: http://htmlbook.ru/html/iframe/ (дата обращения 28.04.2021).

г) Window.postMessage() - Интерфейсы веб API | MDN. [Электронный ресурс] – URL: https://developer.mozilla.org/ru/docs/Web/API/Window/postMessage (дата обращения 29.04.2021).

|  |  |  |
| --- | --- | --- |
| Руководитель ВКР: |  | канд. техн. наук, доцент каф. ИТАС  О.А. Полякова |
| (подпись, дата) |
| Консультант по предметной области: |  | руководитель отдела разработки чат-платформы А.С. Казаринов |
| (подпись, дата) |
| Задание получил: |  | Ширинов Рустам Рамазанович |
| (подпись, дата) |

КАЛЕНДАРНЫЙ ГРАФИК ВЫПОЛНЕНИЯ

ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование этапа | Объем (в%) | Начало | Конец |
| 1. Сбор и анализ исходных данных, постановка задачи | 12 | 01.02.21 | 12.02.21 |
| 1. Изучение теоретического материала по предметной области; анализ и выбор методов и средств решения задачи | 23 | 15.02.21 | 12.03.21 |
| 1. Разработка теоретической части, методики решения; Выбор и разработка средств решения задачи | 24 | 15.03.21 | 09.04.21 |
| 1. Тестирование разработки, постановка экспериментов, формулировка выводов | 21 | 12.04.21 | 07.05.21 |
| 1. Оформление пояснительной записки | 20 | 10.05.21 | 31.05.21 |
| 1. Представление работы на проверку и отзыв руководителя квалификационной работы | - | 12.06.21 | 12.06.21 |
| 1. Предварительная защита работы | - | 16.06.21 | 16.06.21 |
| 1. Прохождение нормоконтроля | - |  |  |
| 1. Представление работы на кафедру | - |  |  |
| 1. Защита на заседании ГЭК | - |  |  |

|  |  |  |
| --- | --- | --- |
| Руководитель ВКР: |  | канд. техн. наук, доцент каф. ИТАС  О.А. Полякова |
| (подпись, дата) |
| Задание получил: |  | Ширинов Рустам Рамазанович |
| (подпись, дата) |

РЕФЕРАТ

Выпускная квалификационная работа бакалавра состоит из 5 разделов, 114 стр., 12 рис., 10 источн., 1 прил.

ВСТРАИВАЕМЫЙ ВЕБ-ЧАТ, ВЕБ-ПРИЛОЖЕНИЕ, ПЛАТФОРМА ROCKET.CHAT, ПОДДЕРЖКА ПОЛЬЗОВАТЕЛЕЙ

Объектом разработки является встраиваемый веб-чат поддержки пользователей на базе платформы «Rocket.Chat».

Целью данной работы является проектирование программной архитектуры и разработка встраиваемого веб-чата поддержки пользователей на базе платформы «Rocket.Chat».

В результате проделанной работы выполнены следующие задачи: выполнен анализ достоинств и недостатков встраиваемых веб-чатов, используемых компанией на текущий момент; исследовано устройство и определены функциональные возможности «Rocket.Chat LiveChat API»; проанализирован разработанный компанией собственный дизайн веб-чата; исследованы подходы к разработке встраиваемых веб-чатов; сформулированы требования к реализуемому встраиваемому веб-чату; выбраны средства разработки для реализации встраиваемого веб-чата; спроектирована и разработана общая архитектура встраиваемого веб-чата и показаны системы, с которыми он взаимодействует; спроектирована и разработана программная архитектура встраиваемого веб-чата; спроектированы и разработаны основные алгоритмы работы программы; разработана программная реализация спроектированного встраиваемого веб-чата; выполнена оценка технико-экономической эффективности внедрения.

Таким образом, результатом выполненной работы является спроектированный и разработанный встраиваемый веб-чат поддержки пользователей на базе платформы «Rocket.Chat».

СОДЕРЖАНИЕ

[ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ 9](#_Toc74792765)

[ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ 10](#_Toc74792766)

[ВВЕДЕНИЕ 11](#_Toc74792767)

[1 Исследовательский раздел 13](#_Toc74792768)

[1.1 Анализ достоинств и недостатков встраиваемых веб-чатов, используемых компанией на текущий момент 13](#_Toc74792769)

[1.1.1 Анализ достоинств и недостатков существующей реализации встраиваемого веб-чата для платформы Rocket.Chat 13](#_Toc74792770)

[1.1.2 Анализ достоинств и недостатков встраиваемого веб-чата от компании Intercom 14](#_Toc74792771)

[1.2 Исследование устройства и функциональных возможностей Rocket.Chat LiveChat API 14](#_Toc74792772)

[1.3 Анализ разработанного компанией дизайна 18](#_Toc74792773)

[1.4 Исследование подходов к разработке встраиваемых веб-чатов 22](#_Toc74792774)

[1.5 Требования к реализуемому встраиваемому веб-чату 24](#_Toc74792775)

[1.6 Выводы по разделу 25](#_Toc74792776)

[2 Конструкторский раздел 27](#_Toc74792777)

[2.1 Выбор средств разработки для реализации встраиваемого веб-чата 27](#_Toc74792778)

[2.2 Проектирование архитектуры встраиваемого веб-чата 28](#_Toc74792779)

[2.2.1 Общая архитектура проектируемой системы 28](#_Toc74792780)

[2.2.2 Проектирование программной архитектуры веб-чата 29](#_Toc74792781)

[2.3 Проектирование и разработка алгоритмов работы веб-чата 33](#_Toc74792782)

[2.3.1 Разработка алгоритма инициализации веб-чата 33](#_Toc74792783)

[2.3.2 Разработка алгоритма отправки сообщения 36](#_Toc74792784)

[2.3.3 Разработка алгоритма дозагрузки сообщений 37](#_Toc74792785)

[2.4 Выводы по разделу 40](#_Toc74792786)

[3 Технологический раздел 41](#_Toc74792787)

[3.1 Настройка окружения разработки 41](#_Toc74792788)

[3.1.1 Установка зависимостей 41](#_Toc74792789)

[3.1.2 Настройка сборщика пакетов Webpack и транслятора Babel 43](#_Toc74792790)

[3.1.3 Настройка форматирования кода с помощью Prettier 50](#_Toc74792791)

[3.1.4 Настройка статического анализа кода с помощью ESLint 51](#_Toc74792792)

[3.2 Реализация интерфейсов взаимодействия с Rocket.Chat LiveChat API 53](#_Toc74792793)

[3.2.1 Реализация интерфейса взаимодействия с Rocket.Chat LiveChat HTTP API 53](#_Toc74792794)

[3.2.2 Реализация интерфейса взаимодействия с Rocket.Chat LiveChat WebSocket API 57](#_Toc74792795)

[3.3 Реализация хранилища для управления состоянием приложения 65](#_Toc74792796)

[3.3.1 Реализация компонента Сhat 65](#_Toc74792797)

[3.3.2 Реализация компонента Agent 70](#_Toc74792798)

[3.3.3 Реализация компонента CustomFields 71](#_Toc74792799)

[3.3.4 Реализация компонента Messages 72](#_Toc74792800)

[3.3.5 Реализация компонента Room 77](#_Toc74792801)

[3.3.6 Реализация компонента Settings 79](#_Toc74792802)

[3.3.7 Реализация компонента Visitor 80](#_Toc74792803)

[3.3.8 Реализация компонента FeedbackForm 83](#_Toc74792804)

[3.4 Реализация пользовательского интерфейса 84](#_Toc74792805)

[3.4.1 Реализация компонента LiveChat 84](#_Toc74792806)

[3.4.2 Реализация компонента Chat 85](#_Toc74792807)

[3.4.3 Реализация компонента Header 87](#_Toc74792808)

[3.4.4 Реализация компонента FinishChat 88](#_Toc74792809)

[3.4.5 Реализация компонента MessageList 90](#_Toc74792810)

[3.4.6 Реализация компонента Message 93](#_Toc74792811)

[3.4.7 Реализация компонента FeedbackForm 96](#_Toc74792812)

[3.4.8 Реализация компонента InputBox 98](#_Toc74792813)

[3.5 Реализация механизма кроссдоменного обмена данными между фреймами 101](#_Toc74792814)

[3.6 Реализация Iframe-приложения 102](#_Toc74792815)

[3.6.1 Реализация хранилища для управления состоянием 102](#_Toc74792816)

[3.6.2 Реализация кнопки открытия и сокрытия веб-чата 105](#_Toc74792817)

[3.6.3 Реализация компонента ChatIframe 106](#_Toc74792818)

[3.7 Реализация скрипта загрузки Iframe-приложения 107](#_Toc74792819)

[3.8 Выводы по разделу 108](#_Toc74792820)

[4 Экономический раздел 109](#_Toc74792821)

[4.1 Расчет затрат на разработку и ввод в эксплуатацию программного продукта 109](#_Toc74792822)

[4.2 Расчет экономического эффекта 110](#_Toc74792823)

[4.3 Сопутствующие технико-экономические результаты 111](#_Toc74792824)

[4.4 Выводы по разделу 111](#_Toc74792825)

[5 Обеспечение информационной безопасности 112](#_Toc74792826)

[ЗАКЛЮЧЕНИЕ 113](#_Toc74792827)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 114](#_Toc74792828)

[ПРИЛОЖЕНИЕ А 115](#_Toc74792829)

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

В настоящей работе применяют следующие термины с соответствующими определениями:

widget (рус. виджет) – это веб-приложение, которое может быть интегрировано в веб-страницу для выполнения какой-то конкретной задачи или решения определенной проблемы.

script (рус. скрипт) – последовательность действий, описанных с помощью скриптового языка программирования для автоматического выполнения определенных задач.

frame (рус. фрейм) – отдельный законченный HTML-документ, который вместе с другими HTML-документами может быть отображён в окне браузера.

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

В настоящем отчете о выпускной квалификационной работе применяют

следующие сокращения и обозначения:

URL – Uniform Resource Locator, унифицированный указатель ресурса

API – Application Programming Interface, программный интерфейс приложения

HTTP – HyperText Transfer Protocol, протокол передачи гипертекста

CSS – Cascading Style Sheets, каскадные таблицы стилей

HTML – HyperText Markup Language, язык гипертекстовой разметки

UML – Unified Modeling Language, унифицированный язык моделирования

ПО – программное обеспечение

XSS – Cross-Site Scripting, межсайтовый скриптинг

ВВЕДЕНИЕ

Неотъемлемой составляющей деятельности современного человека является возможность использования различных сервисов, услуг и развлечений, основанных на инфокоммуникационных технологиях (главным образом, на использовании Интернета), таких, например, как: интернет-магазины, онлайн-игры, социальные сети и других. Одной из проблем, с которой столкнулись компании, владеющие бизнесом в Интернете, оказалась оперативная поддержка пользователей. Наиболее часто используемым инструментом, созданным для решения этой проблемы, является веб-чат – средство обмена сообщениями через Интернет в режиме реального времени. Применение веб-чата обладает существенным преимуществом: его можно встроить на веб-сайт компании и для его работы не нужно никаких дополнительных действий (установок, настроек и других) со стороны пользователя.

Компания, занимающаяся приемом платежей в онлайн-играх, использует встраиваемые веб-чаты для поддержки своих пользователей. При этом, на данный момент компанией используются разные веб-чаты для множества сервисов компании, например, для поддержки партнеров компании используется платный веб-чат от компании Intercom, а для поддержки обычных пользователей используется решение с открытым исходным кодом – «Rocket.Chat.Livechat», которое базируется на платформе «Rocket.Chat». Платформа «Rocket.Chat» – это приложение с открытым исходным кодом, представляющее собой систему обмена различными видами сообщений в реальном времени. Недостатки в интерфейсе и надежности веб-чата «Rocket.Chat.Livechat» не позволяют компании использовать его для всех своих сервисов, поскольку, например, в случае работы с партнерами, это может испортить впечатление о компании и помешать оперативному решению проблемы. Поэтому руководством компании было принято решение о разработке собственного веб-чата на базе платформы «Rocket.Chat», который позволит уменьшить затраты компании на платный веб-чат Intercom и будет лишен недостатков, присущих «Rocket.Chat.Livechat», при этом предоставив возможность поддержки и расширения исходя из предпочтений партнеров и клиентов.

Объектом разработки является встраиваемый веб-чат поддержки пользователей на базе платформы «Rocket.Chat».

Целью данной работы является проектирование архитектуры и разработка встраиваемого веб-чата поддержки пользователей на базе платформы «Rocket.Chat».

Задачи:

* выполнить анализ достоинств и недостатков встраиваемых веб-чатов, используемых компанией на текущий момент;
* исследовать устройство и определить функциональные возможности «Rocket.Chat LiveChat API»;
* проанализировать разработанный компанией собственный дизайн веб-чата;
* исследовать подходы к разработке встраиваемых веб-чатов;
* сформулировать требования к реализуемому встраиваемому веб-чату;
* выбрать средства разработки для реализации встраиваемого веб-чата;
* спроектировать и разработать общую архитектуру встраиваемого веб-чата и показать системы, с которыми он взаимодействует;
* спроектировать и разработать программную архитектуру встраиваемого веб-чата;
* спроектировать и разработать основные алгоритмы работы программы;
* разработать программную реализацию спроектированного встраиваемого веб-чата;
* провести оценку технико-экономической эффективности внедрения.

# Исследовательский раздел

## Анализ достоинств и недостатков встраиваемых веб-чатов, используемых компанией на текущий момент

Анализ достоинств и недостатков существующей реализации встраиваемого веб-чата для платформы Rocket.Chat

Из достоинств существующей реализации встраиваемого веб-чата «Rocket.Chat.Livechat» для платформы «Rocket.Chat» можно отметить:

* открытый исходный код;
* возможность бесплатного использования;
* возможность расширения и добавления функционала.

Ключевым недостатком данного веб-чата является несоответствие пользовательского интерфейса фирменному стилю компании. Во всех продуктах компании пользовательский интерфейс строится с соблюдением фирменного стиля, то есть с использованием конкретных цветов и определенно оформленных элементов. Этот фирменный стиль компании изложен в виде отдельного компонента, называемого UI Kit, который содержит набор готовых элементов пользовательского интерфейса, оформленных в соответствии с брендом. Однако, существующий веб-чат не соответствует фирменному стилю компании и поэтому его встраивание на страницы продуктов компании может нарушить общую стилистику и негативно повлиять на пользовательский опыт. Таким образом, поскольку в клиентских приложениях преобладающую часть исходного кода составляет код, реализующий пользовательский интерфейс, разработчиками, которые отвечают за поддержку существующего веб-чата, было принято решение не перерабатывать исходный код имеющейся реализации, а разработать новую реализацию, соответствующую фирменному стилю компании и стандартам написания исходного кода клиентских приложений компании.

Анализ достоинств и недостатков встраиваемого веб-чата от компании Intercom

Из достоинств встраиваемого веб-чата от компании Intercom можно отметить:

* современный пользовательский интерфейс;
* поддерживается сотрудниками компании Intercom.

Встраиваемый веб-чат от компании Intercom обладает таким же недостатком пользовательского интерфейса, что и реализация веб-чата с открытым исходным кодом для платформы «Rocket.Chat», то есть пользовательский интерфейс не соответствует фирменному стилю компании. Но при этом веб-чат от компании Intercom имеет достаточно современный и удобный пользовательский интерфейс, который отдаленно похож на фирменный стиль компании. Именно поэтому компания позволяет себе использовать его на веб-страницах ключевых продуктов для взаимодействия с партнерами.

Еще одним недостатком веб-чата от компании Intercom является невозможность расширения и добавления нового функционала, необходимого компании, поскольку компания Intercom не имеет цели создавать под каждого своего клиента индивидуальный функционал и дизайн.

Но всё же ключевым недостатком веб-чата от компании Intercom является стоимость его использования. Использование веб-чата от компании Intercom обходится примерно в 40 тыс. долларов в год. Это и является ключевой причиной, по которой руководством компании принято решение заменить веб-чат от компании Intercom на собственное решение.

## Исследование устройства и функциональных возможностей Rocket.Chat LiveChat API

Платформа «Rocket.Chat» – это приложение с открытым исходным кодом, представляющее собой систему обмена различными видами сообщений в реальном времени. Компания использует «Rocket.Chat» для общения с партнерами и оказания технической поддержки пользователям.

В платформе «Rocket.Chat» общение происходит в рамках чатов, которые называются комнатами. Существуют различные виды комнат: личные, публичные групповые, приватные групповые и другие, но для поддержки пользователей используется отдельный тип комнат, который называется «живой чат» (LiveChat). Поскольку в живых чатах общение происходит между сотрудником, использующим платформу «Rocket.Chat» и пользователем, использующим встраиваемый веб-чат, то для того, чтобы встраиваемый веб-чат позволял пользователю отправлять сообщения в платформу и получать их из платформы, был создан специальный серверный LiveChat API, который должен использоваться встраиваемым веб-чатом для того, чтобы взаимодействовать с платформой.

Для реализации встраиваемого веб-чата, необходимо изучить устройство LiveChat API и возможности, которые он предоставляет. Основными сущностями и понятиями, которыми оперирует «Rocket.Chat» для реализации живых чатов, являются:

* комната (Room) – сущность, содержащая информацию об отдельном чате, в котором происходит общение пользователя и сотрудника поддержки;
* департамент (Department) – сущность, содержащая информацию об отделе поддержки пользователей компании. Так, например, пользователь, скорее всего, будет обращаться в отдел поддержки, а партнер компании – в отдел менеджеров;
* посетитель (Visitor) – сущность, содержащая информацию о пользователе, такую, например, как имя и электронная почта;
* агент (Agent) – сущность, содержащая информацию о сотруднике службы поддержки;
* сообщение (Message) – сущность, содержащая информацию о сообщении в определенной комнате. Оно также содержит информацию об отправителе, о дате и времени отправки и других;
* настраиваемое поле (Custom Field) – сущность, представляющая собой поле типа ключ-значение, которое создается в панели администратора для того, чтобы в него записывать дополнительную информацию о посетителе или комнате. Например, можно создать настраиваемое поле «Номер транзакции» для комнаты и в него записывать номер транзакции, с которой у пользователя проблема в данный момент. Или можно создать поле «Номер телефона» для посетителя и в него записывать контактный номер телефона пользователя.

Таким образом, выявлены основные сущности и понятия, которыми оперирует платформа «Rocket.Chat» для реализации живых чатов.

Поскольку речь идет о клиент-серверном взаимодействии, в котором клиентом выступает браузер, то основным протоколом взаимодействия встраиваемого веб-чата и платформы «Rocket.Chat» является протокол HTTP(S) [1]. Но помимо протокола HTTP(S), в LiveChat API для реализации общения в реальном времени используется протокол WebSocket [2].

Исходя из документации, определен список возможностей, которые предоставляет LiveChat HTTP API для реализации встраиваемого веб-чата:

* получение информации об агенте;
* получение информации о конфигурации виджета и дополнительных данных о пользователе, комнате, департаментах и других;
* установка настраиваемых полей;
* отправка сообщения;
* изменение сообщения;
* удаление сообщения;
* получение истории сообщений в комнате;
* получение информации о текущей комнате или регистрация новой, если текущая отсутствует;
* закрытие комнаты;
* переключение комнаты на другой департамент;
* отправка оценки обслуживания;
* получение информации о пользователе;
* регистрация нового пользователя;
* отправка истории навигации пользователя.

Исходя из той же документации [2], определен список возможностей, которые предоставляет LiveChat WebSocket API для реализации встраиваемого веб-чата:

* получение информации о конфигурации виджета и дополнительных данных о пользователе, комнате, департаментах и других;
* регистрация нового пользователя;
* отправка сообщения;
* получение новых или измененных сообщений из определенной комнаты;
* получение информации о том, что сообщение удалено.

Следующим шагом построен алгоритм действий для создания чата и общения между пользователем и сотрудником поддержки. При отправке пользователем первого сообщения в чат, алгоритм действий должен быть следующий:

* если пользователь ранее не был зарегистрирован, посылается запрос на регистрацию, иначе посылается запрос на обновление текущих данных пользователя;
* выполняется запрос на создание комнаты;
* выполняется запрос на подписку событий комнаты;
* выполняется запрос на установку настраиваемых полей, если они есть;
* посылается запрос на отправку сообщения в комнату.

В случае, если комната уже создана, то происходит стандартный обмен сообщениями между пользователем и агентом, новые сообщения пользователь получает по протоколу WebSocket, потому что была выполнена подписка на комнату. По окончанию общения, пользователь или агент могут закрыть чат, в этом случае с сервера по протоколу WebSocket приходит соответствующее событие.

Таким образом, выявлен функционал, предоставляемый «Rocket.Chat LiveChat API» для реализации встраиваемого веб-чата, который позволит пользователю общаться с агентом в реальном времени.

## Анализ разработанного компанией дизайна

С целью определения актуального для поставленной задачи функционала выполнен анализ разработанного специалистами компании дизайна реализуемого веб-чата.

На рисунке 1 представлен главный экран чата при первом открытии его пользователем.

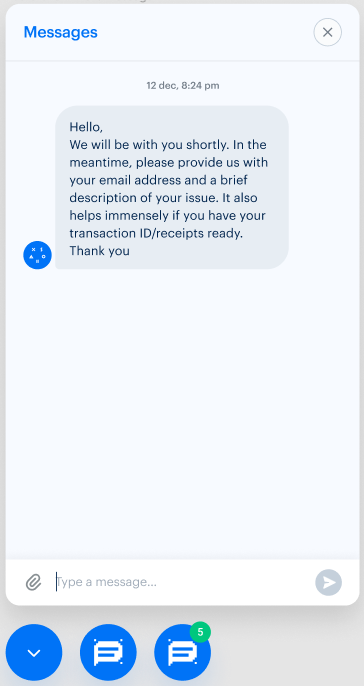


Рисунок – Главный экран чата при первом открытии

Реализуемый встраиваемый веб-чат подразумевает два основных состояния – скрытое и открытое. Когда веб-чат скрыт, в углу веб-страницы отображается только кнопка для его открытия, если веб-чат открыт, то над кнопкой появляется окно чата, с которым взаимодействует пользователь.

На рисунке 1, кроме самого чата, снизу отображены три состояния кнопки открытия и сокрытия чата. Первое состояние, слева, отображается, когда окно веб-чата открыто. Следующие два состояния отображаются, когда окно веб-чата скрыто. Помимо этого, в третьем состоянии кнопки отображен счетчик непрочитанных пользователем сообщений.

В самом чате отображено три ключевых блока: блок вверху – шапка, блок истории сообщений и блок ввода сообщения.

В шапке отображено два элемента: текст слева и кнопка сокрытия чата справа. В блоке истории сообщений находится приветственное сообщение для пользователя при первом открытии чата, у данного сообщения отображается логотип компании и над сообщением находится дата и время отправки. В блоке ввода сообщения находится три элемента: слева – кнопка прикрепления файла, по центру – поле ввода сообщения, а справа – кнопка отправки сообщения.

На рисунке 2 представлены фрагменты переписки между агентом и пользователем.

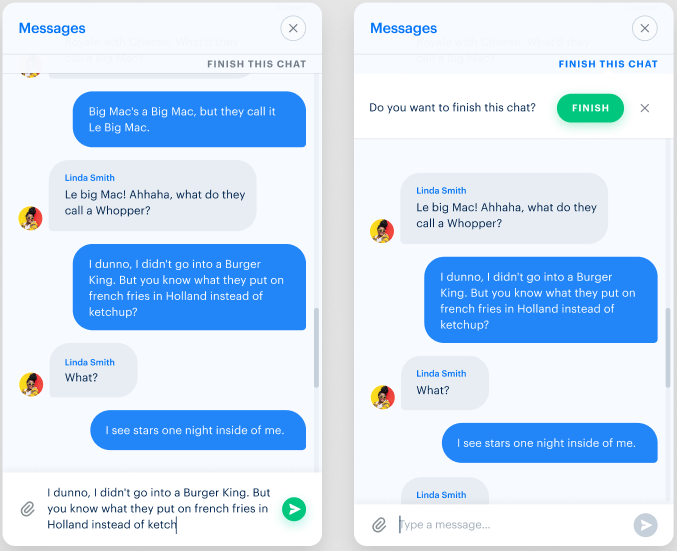


Рисунок – Фрагменты переписки между агентом и пользователем

На рисунке 2 отображено несколько новых элементов. В блоке ввода сообщения видно, что при вводе длинного сообщения, блок увеличивается по высоте. Кроме этого, у кнопки отправки сообщения есть два состояния: когда сообщение нельзя отправить кнопка обозначена серым цветом, а в случае, когда сообщение можно отправить, кнопка обозначена зеленым цветом. В блоке истории сообщений отображены сообщения как от агента и пользователя. При этом, у сообщений агента, слева, вместо логотипа компании, отображается фото профиля агента, а в сообщении сверху его имя. Под шапкой отображен элемент, при нажатии на который появляется новый блок для закрытия комнаты пользователем. В данном блоке отображены: поясняющий текст, кнопка «FINISH» для закрытия чата, и кнопка сокрытия данного блока в виде крестика.

На рисунке 3 отображена форма для оценки пользователем качества обслуживания.



Рисунок – Форма для оценки пользователем качества обслуживания

Форма оценки качества появляется после окончания разговора пользователя и агента. Вверху формы отображен вопрос, который необходим пользователю для выставления оценки обслуживания. В форме отображены две кнопки оценки: положительная и отрицательная. Иконка выбранной оценки увеличивается в размере, относительно своего первоначального состояния, и выделяется в рамку. Помимо этого, после выбора пользователем оценки ему выводится сообщение с поясняющим текстом.

На рисунке 4 отображены возможные состояния сообщения и внешний вид сообщений при отправке файлов.

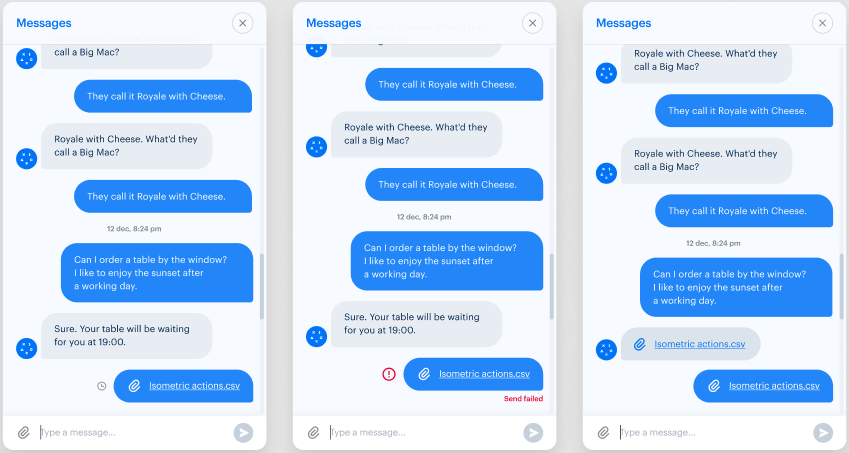


Рисунок – Возможные состояния сообщений и их внешний вид при

отправке файлов

На экране слева отображено сообщение с файлом, которое находится в состоянии «отправка». На центральном экране отображено аналогичное сообщение, но в состоянии «ошибка отправки». И на третьем экране отображено, как должны выглядеть сообщения с файлом от обеих сторон, то есть такое сообщение состоит из иконки «скрепка» и названия файла, при нажатии на которое он должен скачиваться на компьютер пользователя.

Таким образом, выполнен анализ разработанного компанией дизайна, выявлены все функциональные элементы разрабатываемого пользовательского интерфейса встраиваемого веб-чата, определено их поведение и параметры отображения.

## Исследование подходов к разработке встраиваемых веб-чатов

Реализация встраиваемого веб-чата подразумевает разработку браузерного веб-приложения, которое встраивается на веб-страницы продуктов компании. Все браузерные веб-приложения разрабатываются с использованием языков HTML, CSS и JavaScript. В данном случае веб-чат не является исключением и к нему применены те же инструменты, что и к стандартным веб-приложениям, разрабатываемым компанией. Однако, возможность встраивания веб-чата на другие веб-страницы требует исследования подходов, позволяющих реализовать данный функционал.

Анализ подходов, которые используют текущая реализация встраиваемого веб-чата для платформы «Rocket.Chat» и веб-чат от компании Intercom, показал, что они используют один и тот же инструмент для встраивания веб-чата на веб-страницу – HTML элемент iframe, который позволяет встроить в веб-страницу любой другой независимый документ, загруженный по указанному адресу [3]. Таким образом, в веб-страницу продукта компании будет интегрироваться HTML элемент iframe, в котором указан адрес к веб-чату. Однако, есть некоторые тонкости такой реализации.

Первая тонкость заключается в том, что кнопка открытия и сокрытия чата должна быть создана отдельно от него, вне iframe элемента, иначе чат и кнопка будут перекрывать расположенный рядом контент из-за прямоугольной формы iframe элемента. Данная проблема присутствует в существующей реализации веб-чата для платформы «Rocket.Chat» и отображена на рисунке 5.

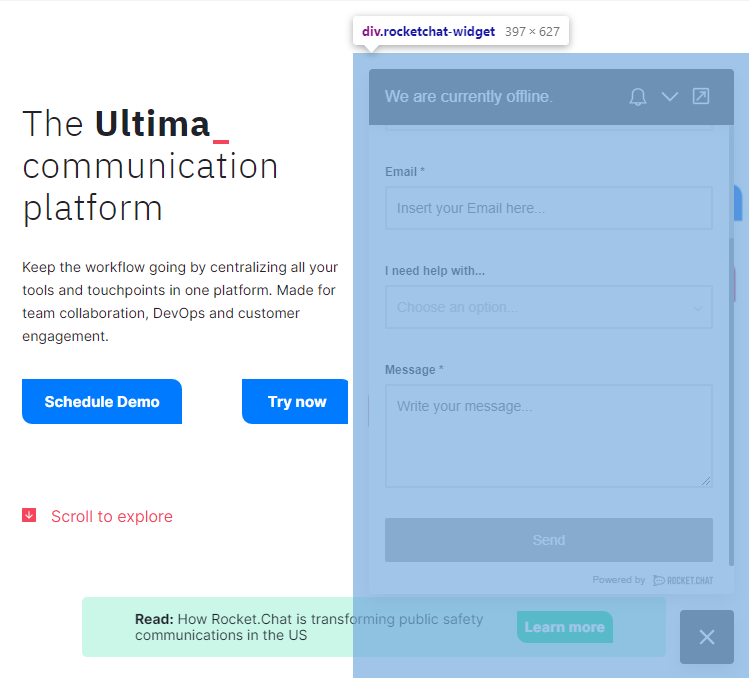


Рисунок – Перекрытие рядом находящихся элементов в существующей

реализации веб-чата для платформы «Rocket.Chat»

На рисунке 5 голубым цветом выделена прямоугольная область, которую перекрывает веб-чат, в итоге – нажатие на кнопку «Learn more» не сработает из-за перекрытия.

Второй тонкостью является передача данных в веб-чат с веб-страницы, на которую он встраивается. Поскольку некоторые продукты компании уже хранят какую-либо информацию о пользователе, который может воспользоваться веб-чатом, то данная информация должна автоматически попадать в веб-чат, чтобы во время общения агенту не пришлось запрашивать у пользователя его данные, которые он уже указал в своем аккаунте. Проблема передачи данных с веб-страницы в веб-чат заключается в том, что в некоторых случаях браузер может блокировать чтение данных веб-страницы из фрейма и в обратном направлении, поскольку это нарушает политику «Одинакового источника» (англ. Same Origin), которую обеспечивает браузер. Согласно данной политике, когда у двух окон различаются протокол, домен или порт, то браузер блокирует для них доступ к переменным и объекту document друг друга [4]. В ином случае окна имеют полный доступ друг к другу. Веб-чат не будет иметь тот же источник, что и продукты компании, на которые он будет встраиваться, поэтому необходимо разработать кроссдоменный способ обмена данными между окном веб-чата и веб-страницей, на которую он будет встраиваться. Браузер позволяет осуществлять кроссдоменный обмен информацией между окнами с помощью метода «window.postMessage()» [5], этот метод позволяет разработать кроссдоменный обмен данными между окном веб-чата и веб-страницей.

Третьей тонкостью является механизм встраивания iframe элемента в веб-страницу продукта. Помимо iframe элемента в веб-страницу должны быть интегрированы кнопка открытия и закрытия чата и механизм кроссдоменного обмена данными между веб-страницей и веб-чатом. Данные требования означают, что необходимо реализовать отдельное приложение, которое будет встраивать веб-чат на веб-страницу и содержать описанный выше функционал. Для загрузки данного приложения в веб-страницу необходимо реализовать скрипт, который будет предоставляться разработчикам продуктов компании для его интеграции в исходный код. В итоге, полный процесс встраивания веб-чата будет унифицирован в одном скрипте.

Таким образом, исследованы подходы к реализации встраиваемого веб-чата. В результате исследования выявлено, что веб-чат должен разрабатываться как стандартное браузерное веб-приложение, а его встраивание будет происходить с помощью HTML iframe элемента. Определены и решены проблемы реализации, присущие данному подходу.

## Требования к реализуемому встраиваемому веб-чату

В результате исследования и анализа выявлены следующие требования к реализуемому встраиваемому веб-чату:

* пользовательский интерфейс веб-чата должен соответствовать разработанному компанией собственному дизайну;
* веб-чат должен осуществлять регистрацию новых пользователей;
* веб-чат должен обновлять информацию о зарегистрированном пользователе, если какие-либо его данные изменились;
* веб-чат должен создавать комнату при первом обращении пользователя;
* веб-чат должен отслеживать онлайн статус пользователя;
* веб-чат должен позволять пользователю осуществлять обмен текстовыми сообщениями с агентом в режиме реального времени;
* веб-чат должен позволять пользователю отправлять и получать файлы;
* веб-чат должен позволять пользователю закрыть комнату самостоятельно;
* при отсутствии в текущей момент созданной комнаты, веб-чат должен отображать приветственное сообщение с текстом, полученным с сервера;
* после закрытия комнаты веб-чат должен выводить форму для оценки пользователем качества обслуживания;
* веб-чат должен иметь возможность встраивания на другие веб-страницы в Интернете;
* веб-чат должен уметь принимать и обрабатывать данные, полученные от веб-страницы, на которую он встроен;
* веб-чат должен отправлять полученные настраиваемые поля от веб-страницы, на которую он встроен, на сервер;
* веб-чат должен иметь возможность сокрытия и открытия по специальной кнопке, отображенной в предоставленном в дизайне;
* веб-чат не должен перекрывать какой-либо рядом расположенный контент на веб-страницах, в которые он встроен.

## Выводы по разделу

Таким образом, в данном разделе решены следующие задачи:

* выполнен анализ достоинств и недостатков встраиваемых веб-чатов, используемых компанией на текущий момент;
* исследовано устройство и определены функциональные возможности «Rocket.Chat LiveChat API»;
* проанализирован разработанный компанией собственный дизайн веб-чата;
* исследованы подходы к разработке встраиваемых веб-чатов;
* сформулированы требования к реализуемому встраиваемому веб-чату.

# Конструкторский раздел

## Выбор средств разработки для реализации встраиваемого веб-чата

Для разработки браузерных веб-приложений используются языки HTML, CSS и JavaScript, поскольку эти языки воспринимаются браузером. Однако, со временем стали появляться различные фреймворки, упрощающие разработку, и надстройки над браузерными языками, которые позволяют упростить и ускорить разработку.

В компании существует внутренний стандарт, который регламентирует выбор средств для разработки различных продуктов. Стандартными инструментами для разработки браузерных веб-приложений в компании являются:

* язык JavaScript для программирования логики работы пользовательского интерфейса;
* библиотека «React.js» для создания пользовательских интерфейсов;
* библиотека MobX для управления состоянием веб-приложения;
* препроцессор SCSS, который предназначен для увеличения уровня абстракции CSS-кода и упрощения его написания;
* язык HTML для создания разметки документа;
* сборщик модулей Webpack для унификации процесса сборки и разработки программы;
* транслятор Babel для транслирования кода на языке JavaScript, написанного в соответствии с последними спецификациями ECMAScript, в совместимую версию, которая соответствует более ранним типам спецификации и поддерживается в используемых раннее браузерах или средах;
* ESLint – инструмент статического анализа кода для выявления проблемных шаблонов в коде на языке JavaScript;
* Prettier – инструмент для форматирования кода в соответствии с заданными правилами оформления.

Таким образом, вышеперечисленные инструменты разработки использованы при реализации встраиваемого веб-чата поддержки пользователей на базе платформы «Rocket.Chat».

## Проектирование архитектуры встраиваемого веб-чата

Общая архитектура проектируемой системы

На рисунке 6 отображена общая архитектура реализуемого веб-чата и системы, с которыми он взаимодействует.

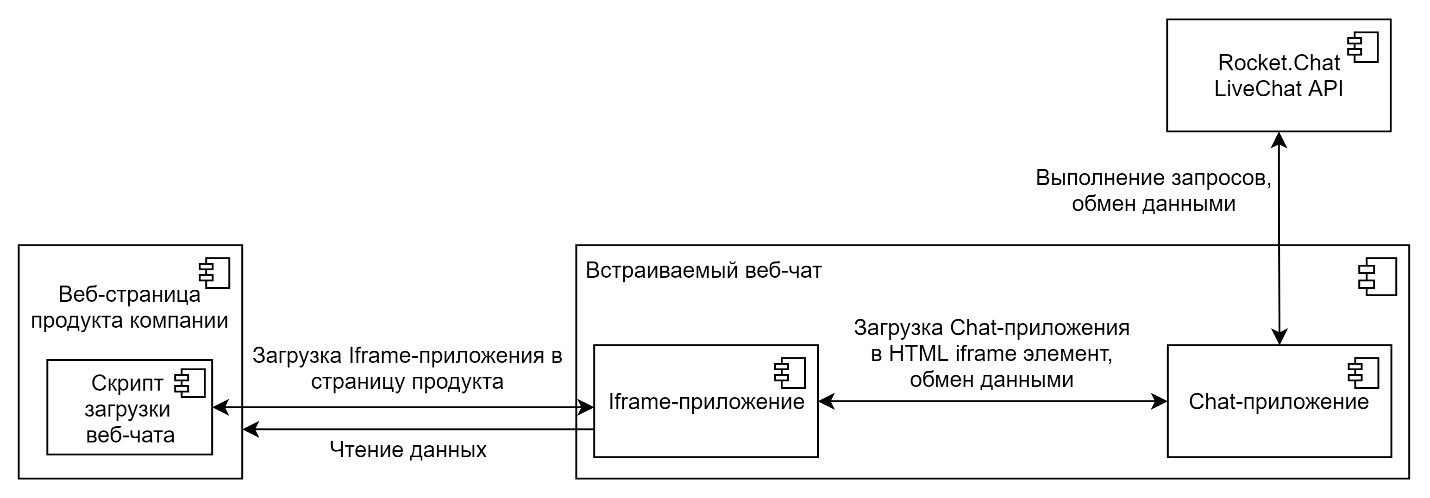


Рисунок – Общая архитектура проектируемой системы

На данном рисунке отображено три основных компонента, взаимодействующих друг с другом:

* веб-страница продукта компании, на которую встраивается веб-чат;
* встраиваемый веб-чат;
* платформа «Rocket.Chat».

На веб-страницу продукта компании интегрируется специальный скрипт загрузки веб-чата, который загружает JavaScript Iframe-приложение, которое встраивается в код веб-страницы продукта и создает кнопку открытия и закрытия веб-чата и HTML iframe элемент, в который будет загружено окно чата. Кроме того, Iframe-приложение должно передавать данные с веб-страницы в Chat-приложение. Окно веб-чата реализуется Chat-приложением, которое содержит логику его работы и предоставляет возможность пользователю общаться с агентом в режиме реального времени. Чтобы позволить пользователю общаться с агентом, который использует платформу «Rocket.Chat», Chat-приложение должно взаимодействовать c «Rocket.Chat LiveChat API» – это и отображено на рисунке 6. Стоит отметить, что несмотря на то, что Iframe-приложение и Chat-приложение являются отдельными друг от друга программными единицами, только их совместное взаимодействие образует встраиваемый веб-чат, поскольку по отдельности ни одно из этих приложений не может быть встраиваемым веб-чатом.

Таким образом, разработана общая архитектура реализуемого веб-чата и показаны системы, с которыми он взаимодействует.

Проектирование программной архитектуры веб-чата

Основываясь на используемых инструментах разработки и сформированных требованиях, спроектирована общая программная архитектура реализуемого веб-чата (рисунок 7).

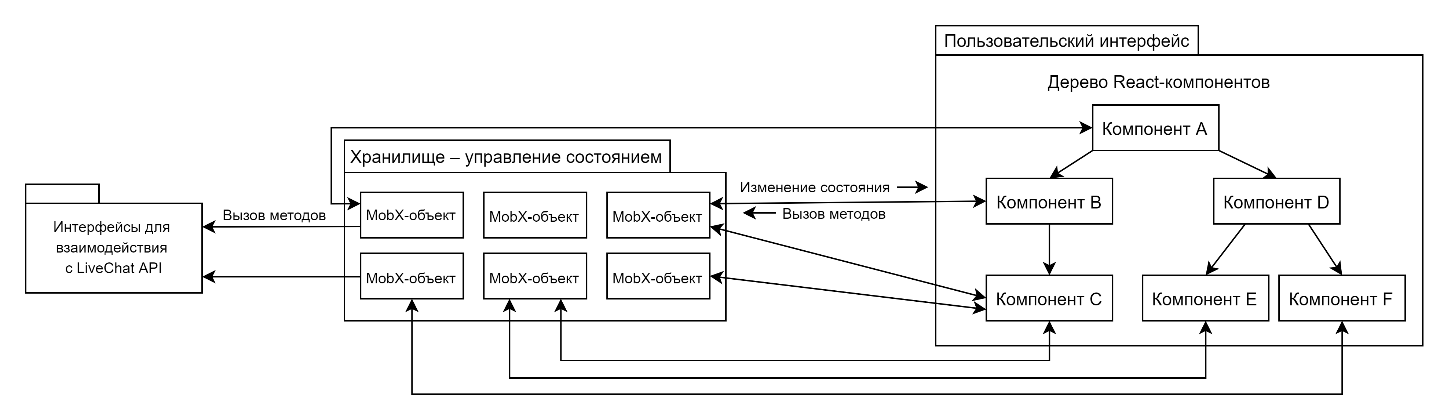


Рисунок – Общая программная архитектура веб-чата

На рисунке 7 отображены три основных компонента приложение веб-чата:

* интерфейсы для взаимодействия с LiveChat API платформы «Rocket.Chat»;
* хранилище, в котором содержится вся логика управления состоянием приложения;
* дерево React-компонентов, которое является реализацией пользовательского интерфейса.

Каждый React-компонент является строительным блоком пользовательского интерфейса, то есть любая кнопка, любые поля ввода, блоки и другие элементы реализуются в React-компонентах. React-компонент отвечает за отображение определенного элемента чата и обработку событий, связанных с этим элементом, таких как нажатие кнопки мыши, выделение и других. Поскольку приложение взаимодействует с пользователем и имеет определенную логику работы, то без хранения данных невозможно обойтись. Хранение данных реализуется в объекте, который на рисунке 7 назван «Хранилище» и является набором MobX-объектов, с определенными методами и свойствами-данными. Библиотека MobX позволяет в React-компоненте отслеживать изменение данных в MobX-объектах, реализуя паттерн «Наблюдатель». Таким образом, логика взаимодействия React-компонентов и MobX-объектов заключается в следующем:

* React-компонент подписывается на изменение определенных данных MobX-объекта и при их изменении срабатывает его перерисовка;
* поскольку состояние приложения хранится в MobX-объектах, то при возникновении событий в компоненте, которые должны затрагивать состояния приложения, React-компонент вызывает методы нужного MobX-объекта.

Для взаимодействия веб-чата с LiveChat API платформы «Rocket.Chat», реализованы специальные интерфейсы, отображенные на рисунке 7 слева. Поскольку данные веб-чата хранятся в MobX-объектах, то логика их обработки находится там же, согласно шаблону «Информационный эксперт». В соответствии с этим именно MobX-объекты будут взаимодействовать с интерфейсами, реализованными для взаимодействия с LiveChat API, поскольку методы данных интерфейсов возвращают данные для хранения и обработки в веб-чате.

Iframe-приложение имеет аналогичную архитектуру, за исключением отсутствия интерфейсов для взаимодействия с LiveChat API, поскольку не требуется чтобы данное приложение обменивалось с сервером «Rocket.Chat» какими-либо данными.

На рисунке 8 отображена подробная UML-диаграмма программных компонентов реализуемого веб-чата.

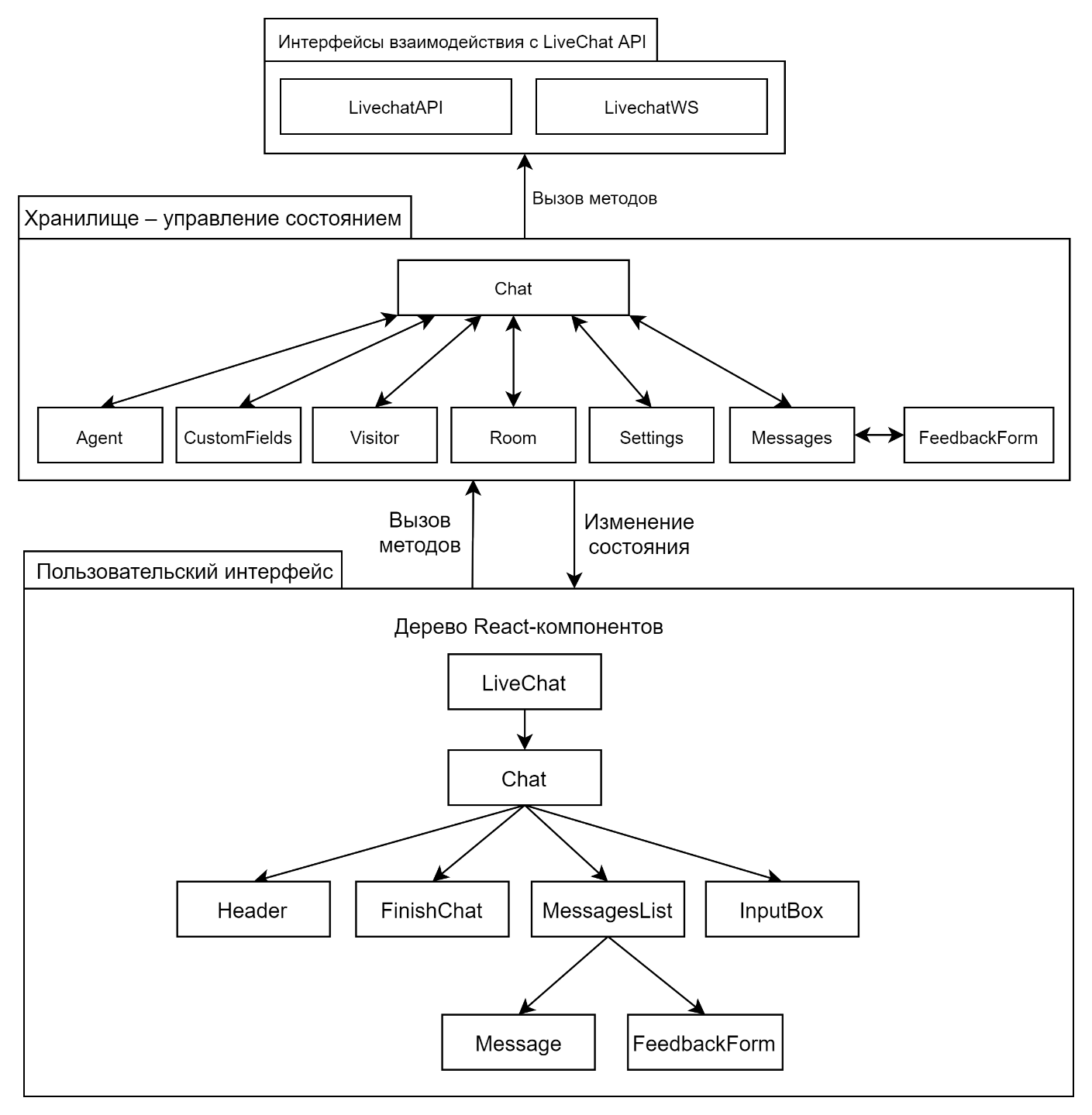


Рисунок – UML-диаграмма программных компонентов

На диаграмме отображены конкретные компоненты реализуемого веб-чата, поделенные на три группы: интерфейсы взаимодействия с LiveChat API, хранилище и пользовательский интерфейс.

Группа компонентов, которые являются интерфейсами взаимодействия с LiveChat API содержит:

* LivechatAPI – интерфейс для взаимодействия с HTTP(S) API платформы «Rocket.Chat»;
* LivechatWS – интерфейс для взаимодействия с WebSocket API платформы «Rocket.Chat».

Хранилище данных состоит из следующих компонентов:

* Chat – корневое хранилище, которое содержит другие хранилища. А также данный объект содержит различные состояния и методы, присущие сущности «Чат» и необходимые для реализации пользовательского интерфейса;
* Agent – компонент, необходимый для хранения данных о сотруднике поддержки, который общается с пользователем;
* CustomFields – компонент, необходимый для хранения и обработки настраиваемых полей;
* Visitor – компонент, необходимый для хранения и обработки имеющихся данных о пользователе;
* Room – компонент, необходимый для управления состоянием комнаты и хранения данных о ней;
* Settings – компонент, необходимый для полученных с сервера настроек, используемых в чате;
* Messages – компонент, необходимый для хранения, загрузки и обработки сообщений, содержащихся в чате, и состояний, связанных загрузкой сообщений;
* FeedbackForm – компонент, который необходим для отображения и обработки формы оценки качества обслуживания в пользовательском интерфейсе.

Пользовательский интерфейс содержит следующие компоненты:

* LiveChat – компонент, являющийся точкой входа отрисовки пользовательского интерфейса, в нем должна вызываться инициализация веб-чата;
* Chat – компонент, который необходим для отображения главного экрана чата, с которым взаимодействует пользователь;
* Header – компонент, отвечающий за отображение шапки веб-чата в пользовательском интерфейсе;
* FinishChat – компонент, который необходим для отображения интерфейса, позволяющего пользователю закрыть текущую комнату;
* MessageList – компонент, который необходим для отображения истории сообщений между пользователем и агентом;
* InputBox – компонент, который необходим для отображения и обработки блока ввода сообщения в пользовательском интерфейсе;
* Message – компонент, который необходим для отображения текстовых и файловых сообщений в пользовательском интерфейсе;
* FeedbackForm – компонент, который необходим для отображения и обработки формы оценки качества обслуживания в пользовательском интерфейсе.

Таким образом, опираясь на общепринятые шаблоны проектирования ПО, разработана программная архитектура встраиваемого веб-чата, на основании которой выполнена его разработка.

## Проектирование и разработка алгоритмов работы веб-чата

Разработка алгоритма инициализации веб-чата

Алгоритм работы программы при инициализации веб-чата представлен в виде диаграммы деятельности на рисунке 9.

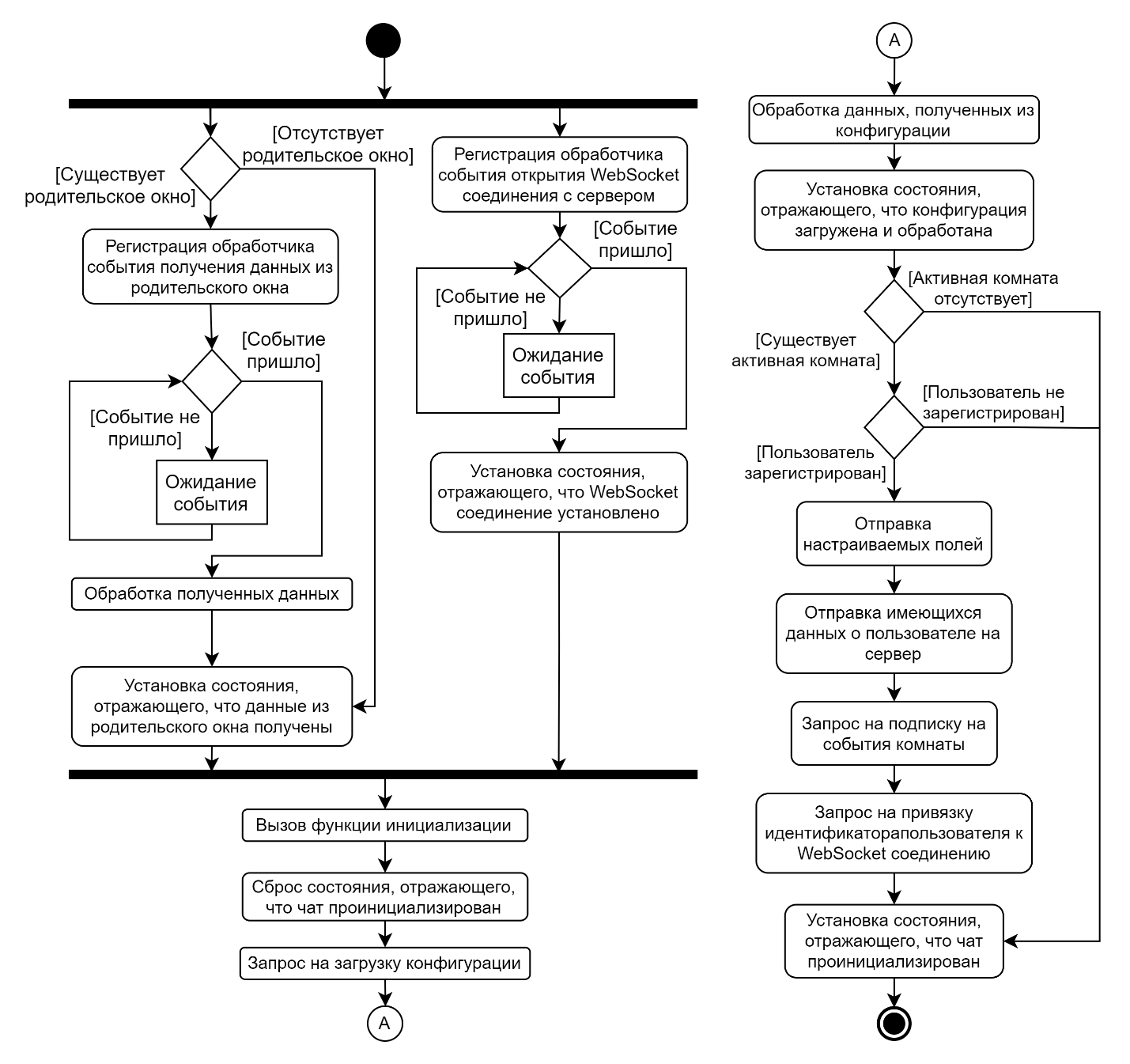


Рисунок – Алгоритм работы программы инициализации веб-чата

Алгоритм не содержит операций, связанных с отображением и изменением пользовательского интерфейса, поскольку данные операции не содержат бизнес-логики и полностью зависят от набора состояний, установка и сброс которых отображены на рисунке 9.

В алгоритме реализованы два параллельных потока действий.

В потоке, который на диаграмме отображен слева, в первую очередь выполняется проверка существования родительского окна (под родительским окном понимается веб-страница, на которую встраивается веб-чат) и в случае, если родительского окна не существует, то следующей операцией в данном потоке будет установка состояния, которое фиксирует, что данные из родительского окна были получены и обработаны. А если родительское окно существует, то следующим действием будет регистрация обработчика события получения данных из родительского окна. Поскольку событие является асинхронной операцией, то программа ожидает, пока это событие будет вызвано и сработает зарегистрированный обработчик. При вызове зарегистрированного обработчика полученные данные обрабатываются и затем в данном потоке будет выполнена уже описанная операция установки состояния, которое фиксирует, что данные из родительского окна получены и обработаны.

В потоке, который на диаграмме отображен справа, первым шагом является регистрация обработчика, который будет реагировать на событие открытия WebSocket соединения с сервером. Затем программа ожидает вызова данного события и при его возникновении выполняется установка состояния, которое фиксирует, что WebSocket соединение с сервером успешно установлено.

После того, как оба потока будут выполнены, программа, реагируя на установленные в потоках состояния, вызывает функцию инициализации.

В функции инициализации выполняется сброс состояния, которое означает, что чат был проинициализирован. Данное состояние, как и все остальные, необходимо для корректного отображения пользовательского интерфейса. Затем выполняется запрос на получение конфигурации веб-чата от сервера. Полученная конфигурация обрабатывается и после её обработки устанавливается соответствующее состояние.

Далее, выполняется проверка регистрации пользователя и существования активной комнаты. Если пользователь не зарегистрирован или активная комната отсутствует, то следующим шагом алгоритма будет установка состояния, фиксирующего, что чат был проинициализирован. Если пользователь зарегистрирован и существует активная комната выполняется следующий набор дополнительных операций:

* отправка настраиваемых полей;
* отправка имеющихся данных о пользователе на сервер, то есть обновление данных;
* выполнение запроса на подписку на события комнаты, такие, например, как «новое сообщение» или «закрытие» комнаты;
* выполнение запроса на привязку идентификатора пользователя к WebSocket соединению. Эта операция необходима для того, чтобы сервер мог отслеживать онлайн статус конкретного пользователя.

После выполнения данного набора операций устанавливается состояние, фиксирующее факт инициализации чата.

Таким образом, разработан алгоритм работы программы при инициализации веб-чата.

Разработка алгоритма отправки сообщения

Алгоритм работы программы при отправке пользователем сообщения представлен в виде диаграммы деятельности на рисунке 10.

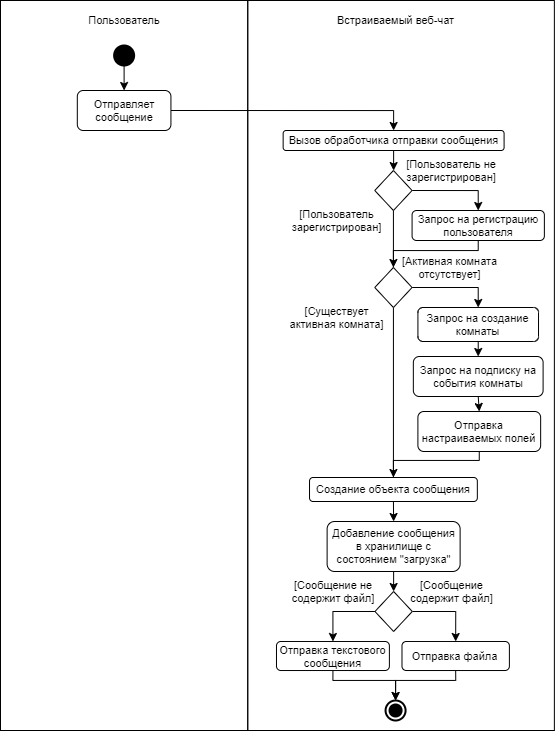


Рисунок – Алгоритм работы программы при отправке пользователем

сообщения

При отправке пользователем сообщения вызывается соответствующий обработчик в хранилище сообщений. Затем выполняется проверка регистрации пользователя. Если пользователь не зарегистрирован, выполняется запрос на его регистрацию. Следующим шагом проверяется существование активной комнаты и если она отсутствует, то выполняется:

* запрос на создание комнаты;
* запрос на подписку на события комнаты, такие как «новое сообщение» или «закрытие» комнаты;
* отправка настраиваемых полей.

Далее, создается объект сообщения и добавляется в хранилище с состоянием «загрузка», которое необходимо для отображения соответствующего состояния в пользовательском интерфейсе.

Предпоследним шагом в алгоритме проверяется содержит ли объект сообщения файл; если содержит, то выполняется отправка файла, если не содержит, то выполняется отправка текстового сообщения.

Таким образом, разработан алгоритм работы программы при отправке пользователем сообщения.

Разработка алгоритма дозагрузки сообщений

С целью оптимизации процесса загрузки веб-чата «Rocket.Chat LiveChat API» предоставляет порционную загрузку истории сообщений, то есть такую, при которой сообщения из истории загружаются не все разом, а порциями по N сообщений. В итоге, при инициализации чата будет загружено только N последних сообщений, а далее, при необходимости, сообщения будут догружаться аналогичными порциями, пока не будут загружены все.

При порционной загрузке сообщений возникает проблема определения наличия ещё не загруженных сообщений. Чтобы определить, что в истории отсутствуют незагруженные сообщения, необходимо загружать N + 1 сообщение и проверять, если было получено меньше, чем N + 1 сообщение, то значит сообщения закончились.

Алгоритм работы программы при дозагрузке сообщений представлен в виде диаграммы деятельности на рисунке 11.

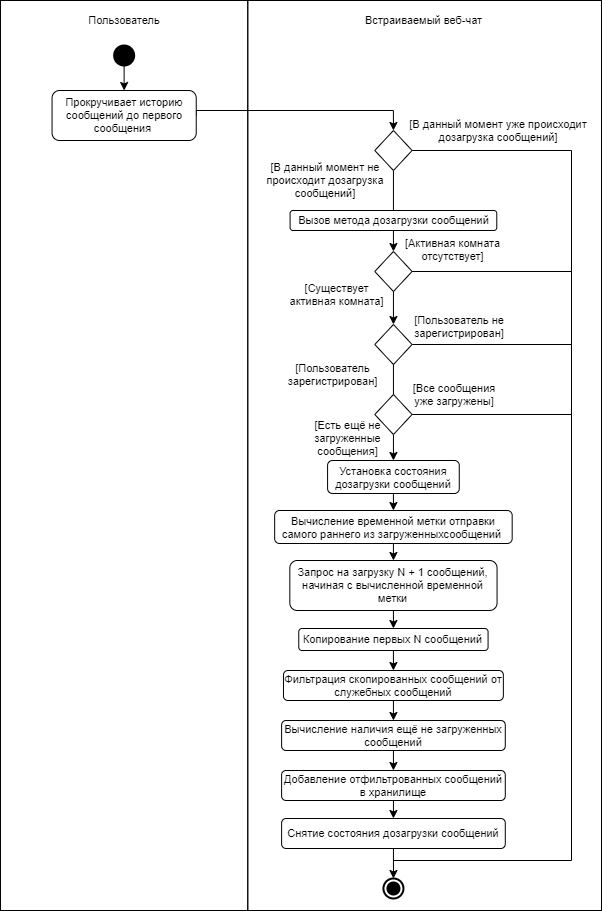


Рисунок – Алгоритм работы программы при дозагрузке сообщений

При рассмотрении пользователем истории сообщений вызывается обработчик события в компоненте интерфейса. В данном обработчике проверяется выполнение дозагрузки сообщений в данный момент и, если дозагрузка не выполняется, то вызывается метод дозагрузки сообщений, определенный в хранилище.

В вызываемом методе выполняется проверка следующих условий:

* существует ли активная комната;
* зарегистрирован ли пользователь;
* существуют ли ещё не загруженные сообщения.

В случае, если хотя бы одно из данных условий не выполняется, то дальнейшее выполнение метода не происходит.

Если все описанные выше условия выполняются, то устанавливается состояние дозагрузки сообщений, которое необходимо для отображения соответствующего состояния в пользовательском интерфейсе.

Затем вычисляется временная метка отправки самого раннего из загруженных сообщений. Эта операция необходима, поскольку «Rocket.Chat LiveChat API» выполняет дозагрузку N сообщений, начиная от конкретной временной метки. В итоге, дозагрузка выполняется с временной метки самого раннего из загруженных сообщений, поскольку пользователю необходимо просмотреть сообщения, которые были отправлены до него.

Далее, выполняется запрос к серверу на загрузку N + 1 сообщения, начиная с вычисленной раннее временной метки. После получения ответа с сервера выполняется копирование первых N сообщений из полученных. Данная операция необходима, поскольку с сервера было запрошено N+1 сообщение, а выводить необходимо только N сообщений. Следующей операцией выполняется фильтрация скопированных сообщений от служебных сообщений, таких как «комната создана», «агент подключился» и других, которые платформа «Rocket.Chat» автоматически добавляет в историю сообщений, но при этом, согласно выявленным требованиям, выводить их в встраиваемом веб-чате не нужно.

Затем вычисляется наличие ещё не загруженных сообщений. В последнем шаге алгоритма отфильтрованные сообщения добавляются в хранилище и снимается состояние, фиксирующее окончание дозагрузки сообщений.

Таким образом, разработан алгоритм работы программы при дозагрузке сообщений.

## Выводы по разделу

Таким образом, в данном разделе решены следующие задачи:

* выбраны средства разработки для реализации встраиваемого веб-чата;
* спроектирована и разработана общая архитектура встраиваемого веб-чата и показаны системы, с которыми он взаимодействует;
* спроектирована и разработана программная архитектура встраиваемого веб-чата;
* спроектированы и разработаны основные алгоритмы работы программы.

# Технологический раздел

## Настройка окружения разработки

Установка зависимостей

Первым шагом установим все зависимости, необходимые для разработки программного продукта и настройки окружения (листинг 1).

Листинг 1 – Установка зависимостей, необходимых для разработки

npm i -D @babel/core @babel/plugin-proposal-class-properties @babel/plugin-proposal-decorators @babel/plugin-syntax-dynamic-import @babel/plugin-transform-async-to-generator @babel/plugin-transform-runtime @babel/preset-env @babel/preset-react @svgr/webpack babel-eslint babel-loader clean-webpack-plugin css-loader dotenv-flow-webpack eslint eslint-config-airbnb eslint-config-prettier eslint-plugin-import eslint-plugin-jsx-a11y eslint-plugin-prettier eslint-plugin-react eslint-plugin-react-hooks file-loader html-webpack-plugin mini-css-extract-plugin prettier sass sass-loader serve style-loader terser-webpack-plugin webpack webpack-cli webpack-dev-server webpack-merge

Далее, установим зависимости, которые будут использоваться непосредственно в коде программы (листинг 2).

Листинг 2 – Установка зависимостей, необходимых для работы программы

npm i -P @xsolla/uikit axios classnames eventemitter3 mobx mobx-react-lite moment prop-types react react-dom react-linkify react-router-dom react-textarea-autosize react-transition-group

В итоге, после установки зависимостей будет сформирован файл с названием «package.json», представленный в листинге 3.

Листинг 3 – Файл «package.json» после установки зависимостей

{

"name": "livechat-widget",

"version": "0.0.1",

"description": "LiveChat Widget for Rocket.chat",

"files": ["/build"],

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"license": "ISC",

"dependencies": {

"@xsolla/uikit": "^2.21.0",

"axios": "^0.21.1",

"classnames": "^2.2.6",

"eventemitter3": "^4.0.7",

"mobx": "^6.3.2",

"mobx-react-lite": "^3.1.6",

"moment": "^2.29.1",

"prop-types": "^15.7.2",

"react": "^17.0.1",

"react-dom": "^17.0.1",

"react-linkify": "^1.0.0-alpha",

"react-router-dom": "^5.2.0",

"react-textarea-autosize": "^8.3.0",

"react-transition-group": "^4.4.1"

},

"devDependencies": {

"@babel/core": "^7.12.3",

"@babel/plugin-proposal-class-properties": "^7.12.1",

"@babel/plugin-proposal-decorators": "^7.14.2",

"@babel/plugin-syntax-dynamic-import": "^7.8.3",

"@babel/plugin-transform-async-to-generator": "^7.12.1",

"@babel/plugin-transform-runtime": "^7.12.1",

"@babel/preset-env": "^7.12.1",

"@babel/preset-react": "^7.12.5",

"@svgr/webpack": "^5.5.0",

"babel-eslint": "^10.1.0",

"babel-loader": "^8.2.1",

"clean-webpack-plugin": "^3.0.0",

"css-loader": "^5.0.1",

"dotenv-flow-webpack": "^1.1.0",

"eslint": "^7.2.0",

"eslint-config-airbnb": "^18.2.1",

"eslint-config-prettier": "^6.15.0",

"eslint-plugin-import": "^2.22.1",

"eslint-plugin-jsx-a11y": "^6.4.1",

"eslint-plugin-prettier": "^3.1.4",

"eslint-plugin-react": "^7.21.5",

"eslint-plugin-react-hooks": "^4.0.0",

"file-loader": "^6.2.0",

"html-webpack-plugin": "^5.3.1",

"mini-css-extract-plugin": "^1.3.1",

"prettier": "^2.1.2",

"sass": "^1.29.0",

"sass-loader": "^10.1.0",

"serve": "^11.3.2",

"style-loader": "^2.0.0",

"terser-webpack-plugin": "^5.0.3",

"webpack": "^5.38.0",

"webpack-cli": "^4.2.0",

"webpack-dev-server": "^3.11.0",

"webpack-merge": "^5.4.0"

}

}

Таким образом, установлены все зависимости, необходимые для разработки программы.

Настройка сборщика пакетов Webpack и транслятора Babel

Webpack – это сборщик модулей, позволяющий унифицировать процесс сборки и разработки программного продукта. Перечислим некоторые ключевые возможности, которые предоставляет Webpack:

* предоставляет возможность сборки проекта, состоящего из множества JavaScript файлов, в один единый JS-файл;
* с помощью Babel позволяет выполнить трансляцию JavaScript кода, написанного в соответствии с последними спецификациями ECMAScript, в обратно совместимую версию, соответствующую более старым спецификациям и поддерживаемую в старых браузерах или средах;
* предоставляет возможность преобразования, объединения или упаковки различных видов файлов;
* следит за изменениями файлов и повторно выполняет различные задачи, такие как форматирование, трансляция и так далее;
* предоставляет возможность запуска встроенного сервера для разработки.

Для настройки Webpack создадим файл конфигурации, с названием «webpack.common.js», в котором будет описана общая настройка Webpack для сборки проекта независимо от окружения development или production.

Файл конфигурации «webpack.common.js» представлен в листинге 4.

Листинг 4 – Файл конфигурации «webpack.common.js»

const { resolve } = require('path');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

const { CleanWebpackPlugin } = require('clean-webpack-plugin');

const { ContextReplacementPlugin } = require('webpack');

module.exports = {

entry: {

chat: resolve(\_\_dirname, './src/chat/index.js'),

iframe: resolve(\_\_dirname, './src/iframe/index.js'),

},

output: {

filename: '[name].js',

chunkFilename: '[name].[chunkhash].js',

path: resolve(\_\_dirname, './build'),

publicPath: '/',

},

module: {

rules: [

{

test: /\.(js|jsx)$/,

exclude: /node\_modules/,

use: [

{

loader: 'babel-loader',

options: {

presets: ['@babel/preset-env', '@babel/preset-react'],

plugins: [

'@babel/plugin-syntax-dynamic-import',

'@babel/plugin-transform-runtime',

'@babel/plugin-transform-async-to-generator',

['@babel/plugin-proposal-decorators', { legacy: true }],

['@babel/plugin-proposal-class-properties', { loose: false }],

],

},

},

],

},

{

test: /\.module.(sa|sc|c)ss$/,

use: ['style-loader', 'css-loader', 'sass-loader'],

},

{

test: /\.(sa|sc|c)ss$/,

exclude: /\.module\.(sa|sc|c)ss$/,

use: [MiniCssExtractPlugin.loader, 'css-loader', 'sass-loader'],

},

{

test: /\.(png|jpg|gif|jpeg)$/,

use: ['file-loader'],

},

{

test: /\.(woff|woff2|eot|ttf|otf)$/,

use: ['file-loader'],

},

{

test: /\.svg$/,

use: ['@svgr/webpack', 'file-loader'],

},

],

},

plugins: [

new CleanWebpackPlugin(),

new MiniCssExtractPlugin({

filename: '[name].[fullhash].css',

}),

new HtmlWebpackPlugin({

template: 'template.html',

chunks: ['chat'],

}),

new ContextReplacementPlugin(/moment[/\\]locale$/, /en/),

],

resolve: {

alias: {

'@': resolve(\_\_dirname, 'src/'),

'@core': resolve(\_\_dirname, 'src/core'),

'@chat': resolve(\_\_dirname, 'src/chat'),

'@iframe': resolve(\_\_dirname, 'src/iframe'),

},

},

};

Конфигурация является JavaScript объектом. В свойстве entry описываются точки входа в собираемые JavaScript приложения [6]. Поскольку при исследовании было выявлено, что встраиваемый веб-чат будет реализован в виде двух взаимодействующих между собой JavaScript приложений, то соответственно в свойстве entry указаны две точки входа: chat и iframe.

Настройка свойства output сообщает Webpack каким образом записывать обработанные файлы на диск [7], (с каким именем, в какую директорию файловой системы и так далее).

Далее, в конфигурации объявлена настройка загрузчиков. Загрузчики – это модули, позволяющие Webpack преобразовывать и обрабатывать различные типы файлов, используемых JavaScript приложением [8]. Для того, чтобы описать для каких конкретно файлов необходимо использовать тот или иной загрузчик в Webpack используются правила (англ. rules). В приведенной выше конфигурации объявлены следующие правила:

* для JavaScript файлов использовать загрузчик, который с помощью Babel транслирует JavaScript код, написанный в соответствии с последними спецификациями ECMAScript, в обратно совместимую версию, соответствующую более старым спецификациям и поддерживаемую в старых браузерах;
* для CSS, SASS и SCSS файлов использовать загрузчики, которые транслируют SASS и SCSS код в CSS и позволяют импортировать данные файлы в код программы на JavaScript;
* для изображений формата PNG, JPG, GIF и JPEG использовать загрузчик, который позволяет импортировать их непосредственно в код программы, преобразуя в строку с URL адресом;
* для шрифтов использовать загрузчик, который позволяет импортировать их непосредственно в код программы, преобразуя в строку с URL адресом;
* для файлов формата SVG использовать загрузчик, который позволяет импортировать данные файлы непосредственно в код программы, преобразуя их в React-компоненты или строку с URL адресом.

Следующая настройка в приведенной конфигурации называется plugins (плагины). Плагины – это дополнительные модули Webpack, которые используется для всех видов задач, которые невозможно реализовать с помощью загрузчиков [9]. В реализованной конфигурации объявлены следующие плагины:

* «clean-webpack-plugin» – данный плагин отвечает за очистку директории, в которую Webpack записывает получившиеся сборки программы. Очистка необходима, чтобы файлы из старых сборок, у которых наименование может изменяться, не скапливались в выходной директории;
* «mini-css-extract-plugin» – данный плагин извлекает CSS в отдельные файлы;
* «html-webpack-plugin» – данный плагин на основе шаблона формирует HTML файл, в который встраивает собранные Webpack ресурсы, такие как JavaScript файлы, CSS файлы и другие. Этот плагин необходим, чтобы автоматически создавать HTML страницу, в которую будет загружаться реализованное JavaScript приложение;
* «ContextReplacementPlugin» – данный плагин необходим, чтобы при использовании пакета «moment.js» загружать только необходимые языковые локали.

Последней настройкой в приведенной конфигурации является «resolve.alias». Данная настройка позволяет определить псевдонимы для различных путей в файловой системе. В итоге, это позволяет сократить строку пути при импорте модулей. Например, вместо «../../../helpers/module.js» можно использовать «@helpers/module.js», где «@helpers» – это псевдоним для нужной директории.

Следующим шагом необходимо создать ещё две конфигурации Webpack, каждая из которых будет использоваться в зависимости от окружения: development или production.

Файл конфигурации для development окружения с названием «webpack.dev.js» представлен в листинге 5.

Листинг 5 – Файл конфигурации «webpack.dev.js»

const DotenvFlow = require('dotenv-flow-webpack');

const { merge } = require('webpack-merge');

const common = require('./webpack.common.js');

module.exports = merge(common, {

mode: 'development',

devtool: 'source-map',

output: {

sourceMapFilename: '[file].map[query]',

},

devServer: {

contentBase: './build',

hot: true,

open: true,

historyApiFallback: true,

port: 4000,

},

plugins: [

new DotenvFlow({

path: \_\_dirname,

node\_env: 'development',

}),

],

});

Данная конфигурация наследуется от описанной ранее конфигурации в файле «webpack.common.js» и расширяет её. В представленной конфигурации добавлены следующие настройки:

* настройка mode, указывающая webpack что сборка выполняется для окружения development;
* настройка генерации карт исходного кода, которые позволят разработчику видеть в браузере исходный код программы, а не его скомпилированную версию;
* настройка для запуска сервера разработки;
* настройка, добавляющая плагин DotenvFlow, который позволяет передать в программу переменные окружения, объявленные в «.env.development» файле.

Файл конфигурации для production окружения с названием «webpack.prod.js» представлен в листинге 6.

Листинг 6 – Файл конфигурации «webpack.prod.js»

const TerserPlugin = require('terser-webpack-plugin');

const DotenvFlow = require('dotenv-flow-webpack');

const { merge } = require('webpack-merge');

const common = require('./webpack.common.js');

module.exports = merge(common, {

mode: 'production',

optimization: {

minimize: true,

minimizer: [new TerserPlugin()],

},

plugins: [

new DotenvFlow({

path: \_\_dirname,

node\_env: 'production',

})

],

});

Данная конфигурация наследуется от описанной ранее конфигурации в файле «webpack.common.js» и расширяет её. В представленной конфигурации добавлены следующие настройки:

* настройка mode, указывающая webpack что сборка выполняется для окружения production;
* настройка оптимизации сборки, то есть минификации выходных файлов;
* настройка, добавляющая плагин DotenvFlow, который позволяет передать в программу переменные окружения, объявленные в «.env.production» файле.

Последним шагом необходимо реализовать скрипты, которые используя конфигурации Webpack будут собирать и запускать приложение (листинг 7).

Листинг 7 – Скрипты для сборки и запуска приложения

{

    "start:dev": "webpack serve -c webpack.dev.js",

    "start:prod": "serve -s ./build",

    "build:dev": "webpack -c webpack.dev.js",

    "build:prod": "webpack -c webpack.prod.js",

}

Рассмотрим каждый из реализованных скриптов:

* скрипт «start:dev» запускает встроенный в webpack веб-сервер для разработки, который будет отдавать собранные файлы, и автоматически пересобирает проект, если какой-либо из файлов изменится;
* скрипт «start:prod» запускает веб-сервер, который будет отдавать файлы, находящиеся в директории «/build»;
* скрипт «build:dev» выполняет сборку программы для окружения development;
* скрипт «build:prod» выполняет сборку программы для окружения production.

Таким образом, выполнена настройка сборщика пакетов Webpack и транслятора Babel, которые необходимы для сборки реализуемой программы. В дополнение к этому, реализованы скрипты, которые, используя конфигурации Webpack, позволяют собирать и запускать приложение.

Настройка форматирования кода с помощью Prettier

Prettier – инструмент для форматирования кода в соответствии с заданными правилами оформления. Использование данного инструмента позволяет обеспечить единое форматирование для исходного кода программы. Prettier изначально имеет стандартное форматирование, которое можно переопределять с помощью файла конфигурации.

Для настройки Prettier создан файл конфигурации «.prettierrc», который представлен в листинге 8.

Листинг 8 – Файл конфигурации «.prettierrc»

{

"tabWidth": 4,

"singleQuote": true,

"printWidth": 150

}

В данном файле описаны следующие настройки форматирования:

* tabWidth – указывает сколько пробелов должно использоваться для отображения вложенности в исходном коде программы;
* singleQuote – указывает, что для обыкновенных строк в коде необходимо использовать одинарные кавычки;
* printWidth – указывает максимальную длину строки исходного кода, до которой нет необходимости выполнять перенос.

Помимо файла настроек необходимо создать файл «.prettierignore», в котором указываются директории или файлы, которые Prettier должен игнорировать. Файл «.prettierignore» представлен в листинге 9.

Листинг 9 – Файл конфигурации «.prettierignore»

build/

В данном файле в единственной строчке указана директория build, которую необходимо игнорировать, поскольку в ней находятся файлы сборки, не требующие форматирования.

Следующим шагом необходимо реализовать скрипт для запуска Prettier, который представлен в листинге 10.

Листинг 10 – Скрипт для запуска Prettier

{

"prettier": "prettier --write \"\*\*/\*.\*(js|jsx)\"",

}

Таким образом, выполнена настройка форматирования кода с помощью Prettier.

Настройка статического анализа кода с помощью ESLint

ESLint – инструмент статического анализа кода для выявления проблемных шаблонов, обнаруженных в коде на языке JavaScript [10].

Для работы ESLint требуется создать файл конфигурации «.eslintrc.js», представленный в листинге 11.

Листинг 11 – Файл конфигурации «.eslintrc.js»

module.exports = {

parser: 'babel-eslint',

env: {

browser: true,

es2020: true,

commonjs: true,

node: true,

},

extends: ['eslint:recommended', 'plugin:react/recommended', 'airbnb', 'airbnb/hooks', 'prettier', 'prettier/react'],

parserOptions: {

ecmaFeatures: {

jsx: true,

},

ecmaVersion: 12,

sourceType: 'module',

},

plugins: ['react', 'prettier'],

rules: {

indent: [2, 4],

'react/jsx-filename-extension': [1, { extensions: ['.js', '.jsx'] }],

'import/no-extraneous-dependencies': [2, { devDependencies: true }],

'import/extensions': 1,

'import/no-unresolved': 0,

'import/prefer-default-export': 0,

'lines-between-class-members': ['error', 'always', { exceptAfterSingleLine: true }],

'no-underscore-dangle': 0,

'jsx-a11y/no-static-element-interactions': 0,

'no-param-reassign': ['error', { props: false }],

'import/no-cycle': 1,

'no-unused-expressions': 0,

'jsx-a11y/click-events-have-key-events': 0,

'no-console': 0,

'no-prototype-builtins': 1,

},

};

В свойстве parserOptions описываются настройки языка, анализ которого необходимо выполнять.

В свойстве extends указывается набор готовых конфигураций, из которых необходимо наследовать правила.

В свойстве rules указываются различные правила и то, каким образом ESLint должен на них реагировать.

Помимо файла конфигурации, необходимо ещё создать файл «.eslintignore», в котором указываются директории или файлы, которые ESLint должен игнорировать. Файл «.eslintignore» представлен в листинге 12.

Листинг 12 – Файл конфигурации «.eslintignore»

build/

В данном файле в единственной строчке указана директория build, которую необходимо игнорировать, поскольку в ней находятся файлы сборки, не требующие статического анализа.

Следующим шагом необходимо реализовать скрипты для запуска ESLint, который представлен в листинге 13.

Листинг 13 – Скрипты для запуска ESLint

{

"prettier": "prettier --write \"\*\*/\*.\*(js|jsx)\"",

"lint": "eslint .",

"fix": "eslint . --fix",

"format": "npm run prettier && npm run fix"

}

Рассмотрим реализованные скрипты:

* скрипт lint выполняет статический анализ кода и выводит результат проверки;
* скрипт fix выполняет статический анализ кода, исправляет проблемные участки кода и выводит результат проверки;
* скрипт format запускает сначала Prettier для форматирования кода, а после этого выполняет скрипт fix.

Таким образом, выполнена настройка статического анализа кода с помощью ESLint.

## Реализация интерфейсов взаимодействия с Rocket.Chat LiveChat API

Реализация интерфейса взаимодействия с Rocket.Chat LiveChat HTTP API

Для взаимодействия с HTTP(S) API платформы «Rocket.Chat» реализуем класс LivechatAPI, представленный в листинге 14.

Листинг 14 – Класс LivechatAPI

class LivechatAPI {

instance;

token;

constructor({ baseURL }) {

this.instance = axios.create({ baseURL });

this.instance.interceptors.response.use((response) => response.data);

}

async getConfig(params = {}) {

params.token ??= this.token;

if (!params.token) {

throw new Error('The token was not specified!');

}

return this.instance.get('livechat/config', { params });

}

async sendMessage(message = {}) {

if (!message.rid) {

throw new Error('The room id (rid) was not specified!');

}

if (!message.msg) {

throw new Error('The message (msg) was not specified!');

}

message.token ??= this.token;

if (!message.token) {

throw new Error('The token was not specified!');

}

return this.instance.post('livechat/message', message);

}

async registerOrUpdateVisitor(visitor = {}) {

visitor.token ??= this.token;

if (!visitor.token) {

throw new Error('The token was not specified!');

}

return this.instance.post('livechat/visitor', { visitor });

}

async room(params = {}) {

params.token ??= this.token;

if (!params.token) {

throw new Error('The token was not specified!');

}

return this.instance.get('livechat/room', { params });

}

async getMessageHistory(rid, params = {}) {

params.token ??= this.token;

if (!params.token) {

throw new Error('The token was not specified!');

}

return this.instance.get(`livechat/messages.history/${rid}`, { params });

}

async setCustomFields(body = {}) {

body.token ??= this.token;

if (!body.token) {

throw new Error('The token was not specified!');

}

if (!body.customFields || typeof body.customFields !== 'object') {

throw new Error('The customFields object was not specified!');

}

body.customFields = Object.entries(body.customFields).map(([key, value]) => ({ key, value, overwrite: true }));

return this.instance.post('livechat/custom.fields', body);

}

async sendFile({ file, rid, token, \_id } = {}) {

if (!file) {

throw new Error('The file was not specified!');

}

if (!rid) {

throw new Error('The rid (room id) was not specified!');

}

const formData = new FormData();

formData.append('file', file);

if (\_id) {

formData.append('\_id', \_id);

}

const headers = { 'x-visitor-token': token ?? this.token };

return this.instance.post(`livechat/upload/${rid}`, formData, { headers });

}

async setVisitorStatus(body = {}) {

body.token ??= this.token;

if (!body.token) {

throw new Error('The token was not specified!');

}

if (!body.status) {

throw new Error('The status was not specified!');

}

return this.instance.post(`livechat/visitor.status`, body);

}

async sendFeedback(body = {}) {

body.token ??= this.token;

if (!body.token) {

throw new Error('The token was not specified!');

}

if (!body.rid) {

throw new Error('The rid was not specified!');

}

if (!body.data || !body.data.length) {

throw new Error('The rating was not specified!');

}

return this.instance.post(`livechat/room.survey`, body);

}

async closeRoom(body = {}) {

body.token ??= this.token;

if (!body.token) {

throw new Error('The token was not specified!');

}

if (!body.rid) {

throw new Error('The rid was not specified!');

}

return this.instance.post(`livechat/room.close`, body);

}

}

Реализованный класс использует HTTP клиент axios для отправки запросов на сервер. В классе объявлено два поля:

* instance – поле, содержащее экземпляр axios;
* token – поле, содержащее токен (уникальный идентификатор) пользователя.

Помимо полей, в классе объявлены следующие методы для взаимодействия с «Rocket.Chat LiveChat HTTP API»:

* getConfig – метод для получения конфигурации чата;
* sendMessage – метод для отправки сообщений;
* sendFile – метод для отправки файлов;
* registerOrUpdateVisitor – метод для обновления данных пользователя. Если пользователя с указанным идентификатором не существует, то создается новый пользователь;
* room – метод для получения текущей активной комнаты пользователя, либо, если таковой нет, то создается новая активная комната;
* getMessageHistory – метод для получения истории сообщений из комнаты;
* setCustomFields – метод для отправки настраиваемых полей;
* setVisitorStatus – метод для отправки онлайн статуса пользователя;
* sendFeedback – метод для отправки оценки пользователем качества обслуживания;
* closeRoom – метод для закрытия комнаты пользователем.

Таким образом, реализован класс LivechatAPI, являющийся интерфейсом для взаимодействия с «Rocket.Chat LiveChat HTTP API»

Реализация интерфейса взаимодействия с Rocket.Chat LiveChat WebSocket API

Для взаимодействия с WebSocket API платформы «Rocket.Chat» реализуем класс WSClient, который является WebSocket клиентом с автоматическим восстановлением соединения (листинг 15).

Листинг 15 – Класс WSClient

import EventEmitter from 'eventemitter3';

export const READY\_STATES = {

CONNECTING: 0,

OPEN: 1,

CLOSING: 2,

CLOSED: 3,

};

export class WSClient extends EventEmitter {

socket;

host;

sendQueue = [];

constructor({ host }) {

super();

this.host = host;

this.connect();

this.on('error', () => ({})); // stub so that the error is not printed to the console

}

send(message) {

try {

if (!this.socket || this.socket.readyState !== READY\_STATES.OPEN) {

this.sendQueue.push(message);

return;

}

const data = typeof message === 'string' ? message : JSON.stringify(message);

this.socket.send(data);

} catch (error) {

console.error('[WebSocket] send error:', error);

throw error;

}

}

connect(reconnect = false) {

const states = [READY\_STATES.CONNECTING, READY\_STATES.OPEN];

if (this.socket && states.includes(this.socket.readyState)) {

if (reconnect) {

this.socket.close();

}

return;

}

console.debug('[WebSocket] Trying to connect!');

this.socket = new WebSocket(this.host);

this.socket.onopen = this.\_onOpen;

this.socket.onmessage = this.\_onMessage;

this.socket.onerror = this.\_onError;

this.socket.onclose = this.\_onClose;

}

\_onOpen = (...args) => {

console.debug('[WebSocket] Connection opened!');

this.emit('open', ...args);

this.sendQueue.forEach((message) => this.send(message));

this.sendQueue = [];

};

\_onMessage = (event) => {

try {

const data = event.data ? JSON.parse(event.data) : {};

this.emit('message', data);

} catch (error) {

console.error('[WebSocket] [ERROR]', error);

}

};

\_onError = (error) => {

console.debug('[WebSocket] [ERROR]', error);

this.emit('error', error);

this.socket.close();

};

\_onClose = (...args) => {

console.debug('[WebSocket] Connection closed!');

this.emit('close', ...args);

this.connect();

};

}

Класс WSClient для создания и управления WebSocket соединением использует интерфейс Web API, предоставляемый браузером. Помимо этого, класс наследуется от класса EventEmitter, который предоставляет функционал для генерации событий, на которые можно подписаться извне.

В реализованном классе объявлены следующие поля:

* socket – поле, используемое для хранения объекта WebSocket соединения;
* host – поле, в котором хранится URL адрес, к которому необходимо подключиться по протоколу WebSocket;
* sendQueue – очередь для отправки сообщений, которая необходима в случае, когда в момент отправления сообщения соединение не было открыто. При возникновении такой ситуации сообщение добавляется в очередь и отправляется при открытии соединения.

Помимо полей, в классе объявлены следующие методы:

* send – метод для отправки сообщений по WebSocket соединению;
* connect – метод для создания WebSocket соединения и установки обработчиков событий, таких как «соединение открыто», «ошибка соединения», «соединение закрыто» и «получено сообщение»;
* «\_onOpen» – обработчик события «соединение открыто», генерирует событие «open», отправляет все сообщение из очереди и очищает её;
* «\_onMessage» – обработчик события «получено сообщение», достает данные из события и передает их и само событие в сгенерированное событие «message»;
* «\_onError» – обработчик события «ошибка соединения», генерирует событие «error» и закрывает текущее WebSocket соединение;
* «\_onClose» – обработчик события «соединение закрыто», генерирует событие «close» и вызывает метод «connect».

Данный класс создает автоматически восстанавливаемое WebSocket соединение, по которому можно отправлять сообщения. При этом, генерируются события, на которые можно подписаться извне, получая информацию о соединении и о пришедших данных.

Следующим шагом реализуем класс LivechatWS, представленный в листинге 16.

Листинг 16 – Класс LivechatWS

import { WSClient } from '../core/WSClient';

import { createToken } from '../helpers/createToken';

class LivechatWS extends WSClient {

token;

subscriptions = {};

internalMessages = {

connect: {

msg: 'connect',

version: '1',

support: ['1', 'pre2', 'pre1'],

},

};

constructor({ host }) {

super({ host });

this.on('open', this.onOpen);

this.on('message', this.onMessage);

this.on('stream-room-messages', this.onStreamRoomMessages);

}

pong() {

this.send({ msg: 'pong' });

}

unsubscribe(name, eventName) {

try {

const key = `${name}.${eventName}`;

if (!this.subscriptions[key]) {

throw new Error('Subscription not found!');

}

const { id } = this.subscriptions[key];

delete this.subscriptions[key];

this.send({ msg: 'unsub', id });

} catch (error) {

console.error('Unsubscribe', error);

throw error;

}

}

subscribe(name, eventName, ...additionalArgs) {

try {

if (!this.token) {

throw new Error('Token was not specified!');

}

const key = `${name}.${eventName}`;

if (this.subscriptions[key] !== undefined) {

throw new Error('Subscription already exists!');

}

const id = createToken();

const args = [{ token: this.token, visitorToken: this.token }, ...additionalArgs];

this.subscriptions[key] = { name, eventName, id };

this.send({

id,

name,

msg: 'sub',

params: [eventName, { useCollection: false, args }],

});

} catch (error) {

console.error('Subscribe', error);

throw error;

}

}

unsubscribeAll() {

try {

Object.values(this.subscriptions).forEach(({ name, eventName }) => {

this.unsubscribe(name, eventName);

});

} catch (error) {

console.error('Unsubscribe all', error);

throw error;

}

}

unsubscribeRoom(rid) {

try {

this.unsubscribe('stream-room-messages', rid);

} catch (error) {

console.error('Unsubscribe room', error);

throw error;

}

}

subscribeRoom(rid, ...args) {

try {

this.subscribe('stream-room-messages', rid, ...args);

} catch (error) {

console.error('Subscribe room', error);

throw error;

}

}

setUpConnection() {

try {

if (!this.token) {

throw new Error('Token was not specified!');

}

this.send({

id: createToken(),

msg: 'method',

method: 'livechat:setUpConnection',

params: [{ token: this.token }],

});

} catch (error) {

console.error('Set up connection', error);

throw error;

}

}

onMessage = (data) => {

if (data.msg === 'ping') {

this.pong();

return;

}

if (data.collection) {

this.emit(data.collection, data);

}

if (data.msg) {

this.emit(data.msg, data);

}

};

onOpen = () => {

this.send(this.internalMessages.connect);

};

onStreamRoomMessages = (data) => {

const messages = data?.fields?.args ?? [];

messages.forEach((message) => {

if (message.editedAt) {

this.emit('livechat-message-edited', message);

return;

}

if (message.t !== undefined) {

this.emit(message.t, message);

}

this.emit('livechat-message-new', message);

});

};

}

Данный класс наследуется от ранее реализованного класса WSClient и расширяет его функционал добавляя поля и методы, необходимые для взаимодействия непосредственно с «Rocket.Chat LiveChat WebSocket API». В реализованном классе объявлены следующие поля:

* token – поле, содержащее токен (уникальный идентификатор) пользователя;
* subscriptions – объект, в котором будут храниться текущие подписки на различные события, такие как «новое сообщение», «сообщение удалено» и т. п., генерируемые «Rocket.Chat LiveChat WebSocket API»;
* internalMessages – объект, в котором хранятся «внутренние» сообщения, то есть такие сообщения, которые необходимы для реализации взаимодействия с «Rocket.Chat LiveChat WebSocket API».

Помимо полей, в классе объявлены следующие методы:

* pong – метод, необходимый для отправки ответа на специальное сообщение, посылаемое сервером «Rocket.Chat», чтобы проверить активность соединения;
* subscribe – метод для создания подписки на определенные события генерируемые сервером «Rocket.Chat», такие, например, как «новое сообщение» и «сообщение удалено»;
* unsubscribe – метод для удаления подписки;
* unsubscribeAll – метод для удаления всех созданных подписок;
* subscribeRoom – метод для создания подписки на события комнаты, в текущей реализации создается только одна подписка на событие «stream-room-messages», которое сообщает о новых или измененных сообщениях;
* unsubscribeRoom – метод для удаления подписок на события комнаты;
* setUpConnection – метод, отправляющий специальное сообщение, тем самым привязывая идентификатор пользователя к WebSocket соединению. Этот метод необходим для того, чтобы сервер мог отслеживать онлайн статус пользователя;
* onMessage – обработчик полученных по WebSocket соединению сообщений. Данный обработчик достает из данных название события, которое хотел сообщить сервер «Rocket.Chat», и генерирует событие с данным названием, чтобы его можно было обработать как снаружи класса, так и внутри;
* onOpen – обработчик события открытия WebSocket соединения, который отправляет специальное сообщение «connect», необходимое для подтверждения установки соединения с сервером «Rocket.Chat»;
* onStreamRoomMessages – обработчик события «stream-room-messages», которое сообщает о новых или измененных сообщениях. Обрабатывая сообщения, данный обработчик генерирует следующие события, передавая в них объект сообщения:
* для измененных сообщений генерируется событие «livechat-message-edited»;
* для сообщений-команд генерируется событие, названием которой является наименование команды;
* для новых сообщений и сообщений-команд генерируется событие «livechat-message-new».

Таким образом, реализован класс WSClient, являющийся WebSocket клиентом с автоматическим восстановлением соединения, на базе которого разработан класс LivechatWS, который является интерфейсом для взаимодействия с «Rocket.Chat LiveChat WS API»

## Реализация хранилища для управления состоянием приложения

Реализация компонента Сhat

Основным компонентом хранилища является класс Chat, представленный в листинге 17.

Листинг 17 – Реализация класса Chat

import { makeObservable, observable, action } from 'mobx';

import { PostEventor } from '@chat/core/PostEventor';

import { LiveChatAPI } from '../api/LiveChatAPI';

import { LiveChatWS } from '../api/LiveChatWS';

import { RETRY\_TIMEOUT } from '../config';

import { Visitor } from './Visitor';

import { Room } from './Room';

import { Agent } from './Agent';

import { CustomFields } from './CustomFields';

import { Settings } from './Settings';

import { Messages } from './Messages';

export class Chat {

@observable propsIsLoaded = false;

@observable isReady = false;

@observable configIsLoaded = false;

@observable wsIsOpen = false;

@observable isOpen = false;

department;

hasAlreadyBeenInit = false;

api;

ws;

constructor(ws, api) {

makeObservable(this);

this.ws = ws;

this.api = api;

this.visitor = new Visitor(this);

this.room = new Room(this);

this.agent = new Agent(this);

this.customFields = new CustomFields(this);

this.settings = new Settings(this);

this.messages = new Messages(this);

if (window.self === window.parent) {

this.propsIsLoaded = true;

} else {

PostEventor.on('chat.props', this.onSetProps);

PostEventor.on('chat.is-open', this.onIsOpenSync);

}

this.ws.on('close', this.onWSClose);

this.ws.on('open', this.onWSOpen);

}

@action.bound

onWSClose() {

this.isReady = false;

}

@action.bound

onWSOpen() {

this.wsIsOpen = true;

if (this.hasAlreadyBeenInit) {

this.init();

}

}

@action.bound

onSetProps(data) {

window.LiveChatWidget = data;

const { visitor, customFields } = window.LiveChatWidget ?? {};

this.visitor.setValues(visitor);

this.customFields.setValues({ values: customFields });

this.propsIsLoaded = true;

}

@action.bound

async init() {

try {

this.isReady = false;

await this.loadConfig();

if (this.visitor.isRegistered && this.room.\_id !== undefined) {

await this.customFields.send();

await this.visitor.save();

await this.room.subscribe();

LiveChatWS.setUpConnection();

}

this.isReady = true;

} catch (error) {

console.error(error);

setTimeout(this.init, RETRY\_TIMEOUT);

} finally {

this.hasAlreadyBeenInit = true;

}

}

@action.bound

async loadConfig() {

const {

config: { guest, room, agent, departments, settings },

} = await this.api.getConfig({ token: this.visitor.token });

this.visitor.setValues(guest, false); // without overwriting existing values

this.visitor.setValues({ isRegistered: guest !== undefined });

this.room.setValues(room);

this.agent.setValues(agent);

this.settings.setValues(settings);

this.department = departments.find(({ \_id, name }) => this.visitor?.department === \_id || this.visitor?.department === name);

this.ws.token = this.visitor.token;

this.api.token = this.visitor.token;

this.configIsLoaded = true;

}

@action.bound

onIsOpenSync({ value }) {

this.isOpen = value;

}

toggle = () => {

PostEventor.send('toggle-chat');

}

}

export default new Chat(LiveChatWS, LiveChatAPI);

В представленном классе объявлены следующие поля:

* «@observable propsIsLoaded» – поле, обозначающее были ли получены и обработаны данные из родительского окна;
* «@observable isReady» – поле, обозначающее был ли проинициализирован чат;
* «@observable configIsLoaded» – поле, обозначающее была ли загружена конфигурация чата;
* «@observable wsIsOpen» – поле, обозначающее было ли открыто WebSocket соединение с сервером;
* «@observable isOpen» – поле, обозначающее открыт или скрыт чат в данный момент. Значение «true» обозначает, что чат открыт. Значение «false» – закрыт;
* department – поле, необходимое для хранения данных об установленном для данного чата департаменте. Под департаментом подразумевается отдел службы поддержки, в который будет направлено обращение пользователя. Это необходимо, поскольку существует множество различных отделов службы поддержки, таких как «отдел поддержки пользователей», «отдел поддержки партнеров» и другие;
* hasAlreadyBeenInit – поле, обозначающее что чат ранее уже когда-либо был проинициализирован;
* api – поле, в котором хранится экземпляр класса LivechatAPI;
* ws – поле, в котором хранится экземпляр класса LivechatWS.

Некоторые из объявленных полей помечены декоратором observable, который предоставляется библиотекой MobX и предписывает ей отслеживать изменение значения данного поля. Это необходимо, чтобы в React-компонентах приложения можно было отслеживать изменения данных полей и на основе их значений отображать различный пользовательский интерфейс. Множество полей, отмеченных как «отслеживаемые» (англ. observable) называют состоянием приложения.

Также в классе объявлены следующие методы:

* constructor – специальный метод, вызываемый при создании экземпляра класса. В данном методе выполняется инициализация различных полей класса, создание хранилищ и регистрация обработчиков различных событий;
* onWSClose – обработчик закрытия WebSocket соединения;
* onWSOpen – обработчик открытия WebSocket соединения;
* onSetProps – обработчик полученных от родительского окна данных;
* init – метод, выполняющий инициализацию чата;
* loadConfig – метод, выполняющий загрузку конфигурации чата;
* onIsOpenSync – обработчик события открытия и сокрытия окна чата;
* toggle – метод, вызывающий событие, необходимое для изменения состояния открытия или сокрытия чата на противоположное.

Некоторые методы данного класса помечены с помощью декоратора action, который сообщает MobX о том, что данный метод изменяет значения отслеживаемых полей.

Таким образом, реализован объект Chat, выступающий в роли корневого хранилища, включающего в себя все остальные хранилища. Помимо этого, данный объект содержит различные состояния и методы, присущие сущности «Чат» и необходимые для реализации пользовательского интерфейса.

Реализация компонента Agent

В листинге 18 представлена реализация класса Agent, необходимого для хранения данных о сотруднике поддержки, который общается с пользователем.

Листинг 18 – Реализация класса Agent

import { makeObservable, observable } from 'mobx';

import { Base, notResettable } from './base';

export class Agent extends Base {

@observable \_id;

@observable email;

@observable name;

@observable username;

constructor() {

super();

makeObservable(this);

chat.ws.on('livechat-close', this.reset);

}

}

Реализованный класс содержит следующие поля:

* «@observable \_id» – поле, содержащее уникальный идентификатор сотрудника в платформе «Rocket.Chat»;
* «@observable email» – поле, содержащее адрес электронной почты сотрудника;
* «@observable name» – поле, содержащее имя сотрудника;
* «@observable username» – поле, содержащее псевдоним сотрудника в платформе «Rocket.Chat».

Таким образом, реализован класс хранилища Agent, хранящий данные о сотруднике поддержки, который общается с пользователем.

Реализация компонента CustomFields

В листинге 19 представлена реализация класса CustomFields, необходимого для хранения и обработки настраиваемых полей.

Листинг 19 – Реализация класса CustomFields

import { makeObservable, action, override } from 'mobx';

import { notResettable, Storable, stored } from './base';

export class CustomFields extends Storable {

@stored values = {};

@notResettable chat;

constructor(chat) {

super('livechat-custom-fields');

makeObservable(this);

this.chat = chat;

this.read();

}

get(name) {

return this.values[name];

}

@action

set(name, value) {

this.values[name] = value;

this.store();

}

@override

setValues({ values } = {}) {

this.values = { ...this.values, ...values };

this.store();

}

async send() {

if (!Object.keys(this.values).length) {

return;

}

try {

await this.chat.api.setCustomFields({ customFields: this.values });

} catch (error) {

console.error('Bad custom fields', error);

}

}

}

В данном классе объявлены следующие поля:

* «@stored values» – объект, необходимый для хранения настраиваемых полей. Декоратор «@stored» обозначает, что значение данного поля необходимо сохранять в локальном хранилище браузера;
* «@notResettable chat» – поле, содержащее ссылку на хранилище Chat. Декоратор «@notResettable» обозначает, что данное поле не должно сбрасываться методами, предназначенными для восстановления исходных значений полей объекта.

Помимо полей, в классе объявлены следующие методы:

* get – метод получения значения какого-либо одного настраиваемого поля;
* set – метод установки значения для настраиваемого поля;
* setValues – метод установки значений для множества настраиваемых полей;
* send – метод для отправки настраиваемых полей на сервер.

Таким образом, реализован класс хранилища CustomFields, необходимый для хранения и обработки настраиваемых полей.

Реализация компонента Messages

В листинге 20 представлена реализация класса Messages, который необходим для хранения сообщений, содержащихся в чате, и состояний, связанных с их загрузкой.

Листинг 20 – Реализация класса Messages

import { makeObservable, action, observable } from 'mobx';

import { PostEventor } from '@chat/core/PostEventor';

import { createToken } from '@chat/helpers/createToken';

import { RETRY\_TIMEOUT } from '@chat/config';

import { FeedbackForm } from './FeedbackForm';

export class Messages {

@observable messages = [];

@observable isLoaded = true;

@observable isLoadingMore = false;

@observable allMessagesLoaded = true;

constructor(chat) {

makeObservable(this);

this.chat = chat;

chat.ws.on('livechat-message-new', this.onMessage);

chat.ws.on('livechat-close', this.onRoomClose);

}

@action.bound

push(...args) {

this.messages.push(...args);

}

@action.bound

async load() {

try {

const { room, visitor, api } = this.chat;

this.isLoaded = false;

const { messages } = await api.getMessageHistory(room.\_id, { token: visitor.token, limit: LIMIT\_STEP + 1 });

this.allMessagesLoaded = messages.length < LIMIT\_STEP + 1;

this.messages = messages?.filter((message) => message.t === undefined).reverse() ?? [];

this.isLoaded = true;

} catch (error) {

console.error('Load messages', error);

setTimeout(this.load, RETRY\_TIMEOUT);

}

}

@action.bound

async loadMore() {

try {

const { room, visitor, api } = this.chat;

if (!room.\_id || !visitor.isRegistered || this.allMessagesLoaded) {

return;

}

this.isLoadingMore = true;

const { ts: end } = this.messages.find((message) => message?.rid !== undefined) ?? {};

const { messages } = await api.getMessageHistory(room.\_id, { limit: LIMIT\_STEP + 1, end });

const textMessages = messages?.slice(0, LIMIT\_STEP)?.filter((message) => message.t === undefined) ?? [];

this.allMessagesLoaded = messages.length < LIMIT\_STEP + 1;

this.messages.unshift(...textMessages.reverse());

this.isLoadingMore = false;

} catch (error) {

console.error('Load more messages', error);

setTimeout(this.loadMore, RETRY\_TIMEOUT);

}

}

@action.bound

async sendMessage(msg, file) {

const { customFields, visitor, room, api, ws } = this.chat;

const \_id = createToken();

const token = visitor?.token;

try {

if (!visitor.isRegistered) {

await visitor.register();

} else if (!room.\_id) {

await visitor.save();

}

if (!room.\_id) {

await room.create();

await room.subscribe();

await customFields.send();

ws.setUpConnection();

}

const rid = room?.\_id;

const message = { \_id, rid, msg, token, file };

this.messages.push({ ...message, isLoading: true });

if (file) {

await api.sendFile(message);

} else {

await api.sendMessage(message);

}

} catch (error) {

console.error('Send message', error);

const message = this.messages.find((item) => \_id === item.\_id);

if (message) {

message.error = { title: 'Send failed' };

}

}

}

@action.bound

displayWelcomeMessage() {

const { department } = this.chat;

if (!department || !department?.welcomeMessage?.text) {

return;

}

this.messages.push({

\_id: createToken(),

ts: Date.now(),

msg: department.welcomeMessage.text,

});

}

@action.bound

onRoomClose() {

this.allMessagesLoaded = true;

const form = new FeedbackForm(this.chat);

this.messages.push(form);

}

@action.bound

onMessageConfirm(message) {

const index = this.messages.findIndex(({ \_id }) => \_id === message.\_id);

if (index !== -1) {

this.messages[index] = message;

} else {

this.messages.push(message);

}

}

@action.bound

onMessage(message) {

if (message.t !== undefined) {

return;

}

message.ts = message.ts?.$date ?? message.ts;

if (message.ts instanceof Date) {

message.ts = message.ts.toISOString();

}

if (this.chat.visitor.token === message.token) {

this.onMessageConfirm(message);

return;

}

const alreadyHaveThisMessage = this.messages.find(({ \_id }) => \_id === message.\_id);

if (alreadyHaveThisMessage) {

return;

}

PostEventor.send('messages.received-agent-message');

this.messages.push(message);

}

}

В представленном классе объявлены следующие поля:

* «@observable messages» – поле, хранящее массив сообщений, содержащихся в чате;
* «@observable isLoaded» – поле, обозначающее была ли загружена история сообщений;
* «@observable isLoadingMore» – поле, обозначающее, что в данный момент выполняется дозагрузка сообщений;
* «@observable allMessagesLoaded» – поле, обозначающее что все сообщения были загружены.

Помимо полей, в классе следующие методы:

* push – метод для добавления сообщений в хранилище;
* load – метод для загрузки истории сообщений;
* loadMore – метод для дозагрузки сообщений;
* sendMessage – метод для отправки текстового сообщения или файла;
* displayWelcomeMessage – метод, добавляющий в хранилище специальное приветственное сообщение;
* onRoomClose – обработчик события закрытия комнаты, в котором в хранилище добавляется специальное сообщение, являющееся формой для оценки качества обслуживания;
* onMessageConfirm – метод обработки новых полученных сообщений и подтверждений сообщений;
* onMessage – обработчик события «livechat-message-new», который получает информацию о новых и измененных сообщениях и сообщениях-командах.

Таким образом, реализован класс хранилища Messages, необходимый для хранения сообщений, содержащихся в чате, и состояний, связанных с загрузкой сообщений.

Реализация компонента Room

В листинге 21 представлена реализация класса Room, который необходим для управления состоянием комнаты и хранения данных о ней.

Листинг 21 – Реализация класса Room

import { makeObservable, observable } from 'mobx';

import { LiveChatWS } from '../api/LiveChatWS';

import { setCookie } from '../helpers/cookies';

import { Base, notResettable } from './base';

export class Room extends Base {

@observable \_id;

@observable departmentId;

@notResettable chat;

constructor(chat) {

super();

makeObservable(this);

this.chat = chat;

chat.ws.on('livechat-close', () => setTimeout(this.reset, 0));

}

async close() {

if (!this.\_id) {

return;

}

try {

await this.chat.api.closeRoom({ rid: this.\_id });

} catch (error) {

console.error(error);

}

}

async subscribe() {

if (!this.\_id) {

return;

}

LiveChatWS.unsubscribeAll();

LiveChatWS.subscribeRoom(this.\_id);

}

async create() {

if (this.\_id) {

return;

}

const params = {};

if (window.parentWindow?.location?.hostname) {

params.host = encodeURIComponent(window.parentWindow.location.hostname);

}

try {

const { room } = await this.chat.api.room(params);

this.setValues(room);

setCookie('rc\_rid', this.\_id, { SameSite: 'None', secure: true });

setCookie('rc\_token', this.chat.visitor.token, { SameSite: 'None', secure: true });

setCookie('rc\_room\_type', 'l', { SameSite: 'None', secure: true });

} catch (error) {

console.error(error);

}

}

}

Реализованный класс содержит следующие поля:

* «@observable \_id» – поле, содержащее уникальный идентификатор комнаты;
* «@observable departmentId» – поле, содержащее уникальный идентификатор департамента, в котором находится данная комната.

Помимо полей, класс содержит следующие методы:

* close – метод, используемый для закрытия комнаты;
* subscribe – метод, используемый для создания подписки на события комнаты;
* create – метод, используемый для создания новой комнаты.

Таким образом, реализован класс хранилища Room, необходимый для управления состоянием комнаты и хранения данных о ней.

Реализация компонента Settings

В листинге 22 представлена реализация класса Settings, необходимого для хранения данных о полученных с сервера настройках, которые используются в чате.

Листинг 22 – Реализация класса Settings

import { makeObservable, observable } from 'mobx';

import { Base } from './base';

export class Settings extends Base {

@observable maxMessageLength;

constructor(chat) {

super();

makeObservable(this);

this.chat = chat;

}

}

В реализованном классе объявлено только одно поле «@observable maxMessageLength», которое хранит данные о максимально возможной длине сообщения. Сообщения, имеющие длину, превосходящую указанную в настройках, будут отклонены сервером.

Таким образом, реализован класс хранилища Settings, необходимый для хранения данных о полученных с сервера настройках, используемых в чате.

Реализация компонента Visitor

В листинге 23 представлена реализация класса Visitor, необходимого для хранения и обработки имеющихся данных о пользователе.

Листинг 23 – Реализация класса Visitor

import { makeObservable, observable, action } from 'mobx';

import { PostEventor } from '@chat/core/PostEventor';

import { createToken } from '@chat/helpers/createToken';

import { onlineEvents } from '@/config';

import { stored, Storable, notResettable, defaultValue } from './base';

export class Visitor extends Storable {

@observable @stored \_id;

@observable @stored username;

@observable @stored token;

@observable @stored department;

@observable @stored name;

@observable @stored email;

@defaultValue(false) isRegistered;

@notResettable isOnline = false;

@notResettable chat;

constructor(chat) {

super('livechat-visitor');

makeObservable(this);

this.chat = chat;

this.read();

if (!this.token) {

this.setValues({ token: createToken() });

}

PostEventor.on('visitor.online-status', this.onOnlineStatusChange);

onlineEvents.forEach((eventName) => document.addEventListener(eventName, this.onOnlineStatusChange.bind(this, { isOnline: true })));

}

@action.bound

async onOnlineStatusChange({ isOnline }) {

if (this.isOnline === isOnline || !this.chat.api.token || !this.isRegistered || !this.chat.room.\_id) {

return;

}

try {

this.isOnline = isOnline;

await this.chat.api.setVisitorStatus({ status: isOnline ? 'online' : 'offline' });

} catch (error) {

this.isOnline = !isOnline;

console.error('Set visitor status', error);

}

}

async save() {

const { visitor } = await this.chat.api.registerOrUpdateVisitor(this);

this.setValues(visitor);

}

async register() {

if (this.isRegistered) {

return;

}

const { visitor } = await this.chat.api.registerOrUpdateVisitor(this);

this.setValues({ ...visitor, isRegistered: true });

}

}

В представленном классе объявлены следующие поля:

* «@observable @stored \_id» – поле, содержащее уникальный идентификатор пользователя в платформе «Rocket.Chat»;
* «@observable @stored username» – поле, содержащее псевдоним пользователя;
* «@observable @stored token» – поле, содержащее уникальную строку, которая необходима для идентификации пользователя и предоставления ему доступа к API;
* «@observable @stored department» – поле, содержащее название департамента, в который был направлен пользователь;
* «@observable @stored name» – поле, содержащее имя пользователя;
* «@observable @stored email» – поле, содержащее адрес электронной почты пользователя;
* «@defaultValue(false) isRegistered» – поле, обозначающее зарегистрирован пользователь в платформе «Rocket.Chat» или нет. Декоратор «@defaultValue» устанавливает переданное в декоратор значение для данного поля и при вызове методов, которые восстанавливают исходные значения полей объекта, в данное поле будет установлено значение, переданное в декаратор;
* «@notResettable isOnline» – поле, отражающее онлайн статус пользователя.

Помимо полей, в классе объявлены следующие методы:

* onOnlineStatusChange – обработчик событий, сообщающих о изменении онлайн статус пользователя;
* save – метод, используемый для отправки обновленных данных о пользователе на сервер;
* register – метод, используемый для регистрации пользователя в платформе «Rocket.Chat».

Таким образом, реализован класс хранилища Visitor, необходимый для хранения и обработки имеющихся данных о пользователе.

Реализация компонента FeedbackForm

В листинге 24 представлена реализация класса FeedbackForm, необходимого для хранения и обработки формы оценки качества обслуживания.

Листинг 24 – Реализация класса FeedbackForm

import EventEmitter from 'eventemitter3';

import { makeObservable, observable, action } from 'mobx';

import { createToken } from '../helpers/createToken';

export class FeedbackForm extends EventEmitter {

@observable \_id;

@observable rid;

@observable rating;

type = 'FeedbackForm';

constructor(chat) {

super();

makeObservable(this);

this.chat = chat;

this.\_id = createToken();

this.rid = this.chat.room.\_id;

}

@action.bound

setRating(value) {

if (this.rating !== undefined) {

return;

}

this.rating = value;

this.chat.messages.push({

\_id: createToken(),

ts: Date.now(),

msg: FeedbackReplyMessages[value] ?? defaultFeedbackMessage,

});

this.send();

}

@action.bound

async send() {

try {

const { rid, rating, chat } = this;

const data = [{ name: 'rating', value: rating }];

await chat.api.sendFeedback({ rid, data });

} catch (error) {

console.error('Feedback send', error);

}

}

}

В представленном классе объявлены следующие поля:

* «@observable \_id» – поле, содержащее уникальный идентификатор формы;
* «@observable rid» – поле, содержащее уникальный идентификатор комнаты;
* «@observable rating» – поле, содержащее оценку пользователя. Возможные значения 0 (не доволен поддержкой) и 1 (доволен поддержкой);
* type – поле, указывающее на тип формы, необходимо для реализации пользовательского интерфейса.

Помимо полей, в реализованном классе объявлены следующие методы:

* setRating – обработчик выбранной пользователем оценки. Данный обработчик устанавливает выбранное пользователем значение, отправляет его на сервер и добавляет в чат новое сообщение с благодарностью;
* send – метод, используемый для отправки выбранной пользователем оценки на сервер.

Таким образом, реализован класс хранилища FeedbackForm, необходимый для хранения и обработки формы оценки качества обслуживания.

## Реализация пользовательского интерфейса

Реализация компонента LiveChat

Основным React-компонентом, с которого начинается отрисовка пользовательского интерфейса, является компонент LiveChat, представленный в листинге 25.

Листинг 25 – Реализация компонента LiveChat

import React, { useEffect } from 'react';

import { observer } from 'mobx-react-lite';

import Loader from '@xsolla/uikit/lib/loader';

import { useStores } from './hooks/useStores';

import { Chat as ChatComponent } from './pages/Chat';

import './LiveChat.scss';

function LiveChat() {

const { Chat } = useStores();

const { configIsLoaded, propsIsLoaded, wsIsOpen, init } = Chat;

useEffect(() => {

if (propsIsLoaded && wsIsOpen) {

init();

}

}, [init, propsIsLoaded, wsIsOpen]);

return <div className="livechat">{configIsLoaded ? <ChatComponent /> : <Loader color="blue" centered />}</div>;

}

export default observer(LiveChat);

Реализованный React-компонент отслеживает изменение трех состояний: propsIsLoaded, wsIsOpen и configIsLoaded. При истинности первых двух состояний компонент вызывает инициализацию чата. Третье же состояние используется компонентом для отображения интерфейса загрузки чата, если конфигурация ещё не была загружена, то отображается состояние загрузки чата, в ином случае отображается интерфейс чата.

Таким образом, реализован React-компонент LiveChat, являющийся точкой входа отрисовки пользовательского интерфейса.

Реализация компонента Chat

Следующим основным React-компонентом является компонент Chat, который необходим для отображения главного экрана чата, с которым взаимодействует пользователь. Реализация компонента Chat представлена в листинге 26.

Листинг 26 – Реализация компонента Chat

import React from 'react';

import { Header } from '@chat/components/Header';

import { InputBox } from '@chat/components/InputBox';

import { MessagesList } from '@chat/components/MessagesList';

import { FinishChat } from '@chat/components/FinishChat';

export default function Chat() {

return (

<>

<Header />

<FinishChat />

<MessagesList />

<InputBox />

</>

);

}

Реализованный React-компонент не имеет никакой логики работы с состояниями и служит как контейнер для других React-компонентов, реализующих более мелкие блоки пользовательского интерфейса. Компонент Chat включает в себя следующие React-компоненты:

* Header – React-компонент, который реализует шапку веб-чата;
* FinishChat – React-компонент, реализующий блок, который содержит кнопку и модальное окно, необходимые для закрытия чата пользователем;
* MessageList – React-компонент, реализующий блок, в котором отображается история сообщений;
* InputBox – React-компонент, реализующий блок ввода сообщения.

На рисунке 12 отображено соответствие React-элементов и блоков пользовательского интерфейса веб-чата.

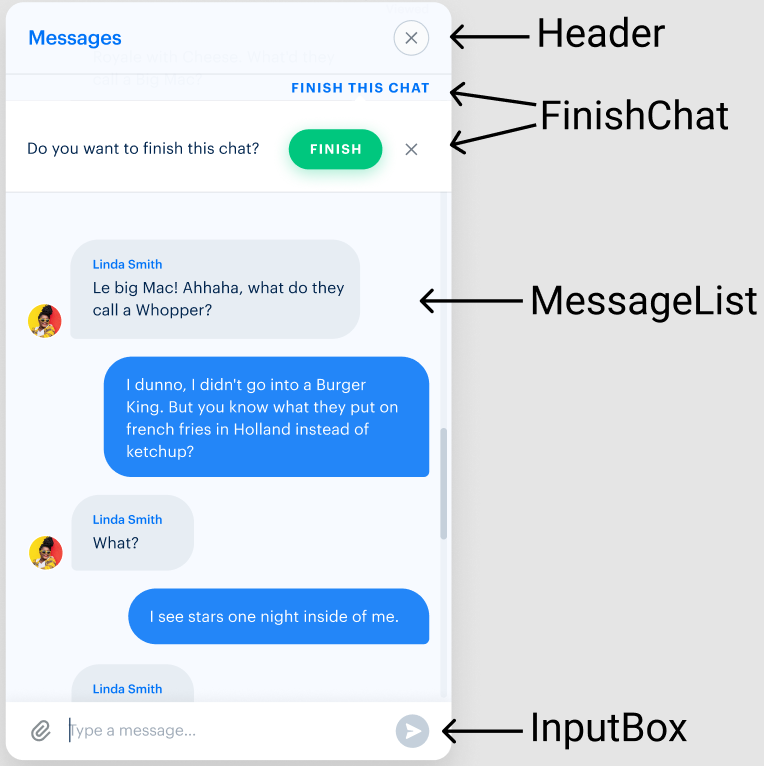


Рисунок – Соответствие React-элементов и блоков пользовательского

интерфейса веб-чата

Таким образом, реализован React-компонент Chat, который необходим для отображения главного экрана чата, с которым взаимодействует пользователь.

Реализация компонента Header

Реализация компонента Header представлена в листинге 27.

Листинг 27 – Реализация компонента Header

import React from 'react';

import { observer } from 'mobx-react-lite';

import { useStores } from '@chat/hooks/useStores';

import { ReactComponent as CrossIcon } from '@chat/assets/cross.svg';

import './Header.scss';

function Header() {

const { Chat } = useStores();

return (

<div className="livechat\_\_header">

<h3>Messages</h3>

{window.self !== window.parent && (

<button className="livechat-close" onClick={Chat.toggle} type="button" title="Close chat">

<CrossIcon />

</button>

)}

</div>

);

}

export default observer(Header);

Реализованный React-компонент содержит два элемента: текст слева и кнопку закрытия чата справа. При этом кнопка отображается только в случае, если веб-чат имеет родительское окно, а при нажатии на кнопку вызывается метод переключения состояния открытия и закрытия окна чата.

Таким образом, реализован React-компонент Header, который необходим для отображения шапки веб-чата в пользовательском интерфейсе.

Реализация компонента FinishChat

Реализация компонента FinishChat представлена в листинге 28.

Листинг 28 – Реализация компонента FinishChat

import React, { useState, useCallback, useRef } from 'react';

import { observer } from 'mobx-react-lite';

import cn from 'classnames';

import Button from '@xsolla/uikit/lib/button';

import { useStores } from '@chat/hooks/useStores';

import { ZoomIn } from '@chat/components/Animations/ZoomIn';

import { ReactComponent as CrossIcon } from '@chat/assets/cross.svg';

import './FinishChat.scss';

function FinishChat() {

const { Chat } = useStores();

const nodeRef = useRef(null);

const [showCloseModal, setShowCloseModal] = useState(false);

const toggleChatFinish = useCallback(() => {

setShowCloseModal(current => !current);

}, []);

const onChatFinish = useCallback(() => {

setShowCloseModal(false);

Chat.room.close();

}, [Chat.room]);

return (

<>

{Chat.room.\_id !== undefined && (

<div className={cn('livechat\_\_finish', { livechat\_\_finish\_active: showCloseModal })}>

<button onClick={toggleChatFinish} type="button">

Finish this chat

</button>

</div>

)}

<ZoomIn nodeRef={nodeRef} showIn={showCloseModal}>

<div ref={nodeRef} className="relative-wrapper">

<div className="livechat\_\_close-room">

<p>Do you want to finish this chat?</p>

<Button onClick={onChatFinish} appearance="primary" type="submit">

Finish

</Button>

<button className="close-room-cancel" type="button" title="Finish chat">

<CrossIcon onClick={toggleChatFinish} />

</button>

</div>

</div>

</ZoomIn>

</>

);

}

export default observer(FinishChat);

Реализованный React-компонент состоит из модального окна и кнопки, необходимой для его открытия. Кнопка для открытия модального окна отображается только в случае, если существует активная комната. Само же модальное окно состоит из трех элементов: поясняющий текст, кнопка закрытия чата и кнопка закрытия модального окна. При нажатии на кнопку закрытия чата вызывается соответствующий метод в компоненте хранилища Room. При нажатии на кнопку закрытия модального окна оно скрывается.

Таким образом, реализован React-компонент FinishChat, который необходим для отображения интерфейса, позволяющего пользователю закрыть текущую комнату.

Реализация компонента MessageList

Реализация компонента MessageList представлена в листинге 29.

Листинг 29 – Реализация компонента MessageList

/\* eslint-disable react-hooks/exhaustive-deps \*/

import React, { useCallback, useEffect, useRef } from 'react';

import { observer } from 'mobx-react-lite';

import Loader from '@xsolla/uikit/lib/loader';

import { useStores } from '../../hooks/useStores';

import { Message } from '../Message';

import { FeedbackForm } from '../FeedbackForm';

import './MessagesList.scss';

const MessageTypes = {

FeedbackForm,

};

function MessagesList() {

const { Chat, Messages } = useStores();

const messageListRef = useRef();

const oldestMessageId = useRef();

const lastMessageListHeight = useRef();

const { isReady, isOpen, visitor, room } = Chat;

const { messages, isLoaded, isLoadingMore, allMessagesLoaded } = Messages;

useEffect(() => {

if (!room.\_id) {

Messages.displayWelcomeMessage();

}

}, []);

useEffect(() => {

if (isReady && visitor.isRegistered && room.\_id) {

Messages.load();

}

}, [Messages, isReady]);

useEffect(() => {

if (isLoadingMore) {

lastMessageListHeight.current = messageListRef.current.scrollHeight;

}

}, [isLoadingMore]);

useEffect(() => {

if (!isReady || !isLoaded || !messages.length || !isOpen) {

return;

}

const [firstMessage] = messages;

if (oldestMessageId.current && firstMessage.\_id !== oldestMessageId.current) {

messageListRef.current.scrollTop = messageListRef.current.scrollHeight - lastMessageListHeight.current;

} else {

messageListRef.current.scrollTop = messageListRef.current.scrollHeight;

}

oldestMessageId.current = messages[0].\_id;

}, [messages.length, isLoaded, isOpen]);

const onScroll = useCallback(() => {

if (messageListRef.current.scrollTop === 0 && !isLoadingMore) {

Messages.loadMore();

}

}, [Messages, isLoadingMore]);

const messageList = messages.map((message, i) => {

if (MessageTypes[message.type] !== undefined) {

const CustomMessage = MessageTypes[message.type];

return <CustomMessage key={message.\_id} message={message} />;

}

return <Message key={message.\_id} message={message} prevMessage={messages[i - 1]} />;

});

return (

<div className="message-list" ref={messageListRef} onScroll={onScroll}>

{isLoadingMore && !allMessagesLoaded && <Loader appearance="dots" color="blue" />}

{isLoaded ? messageList : <Loader color="blue" centered />}

</div>

);

}

export default observer(MessagesList);

Реализованный React-компонент содержит следующую логику:

* при отсутствии активной комнаты, вызывается метод для добавления сообщения приветствия в хранилище;
* при условии, что чат проинициализирован, пользователь зарегистрирован и существует активная комната, вызывается метод загрузки истории сообщений;
* отслеживается положение полосы прокрутки сообщений и как только пользователь достиг самого первого сообщения вызывается метод, реализующий дозагрузку сообщений;
* после дозагрузки выполняется расчет позиции полосы прокрутки на основе положения, которое было до загрузки новой порции сообщений. Это необходимо, поскольку иначе, после дозагрузки сообщений, положение полосы прокрутки сохранится относительно блока и пользователь будет видеть самое первое сообщение, но уже с дозагруженными сообщениями;
* каждому сообщению из хранилища ставится в соответствие определенный React-компонент. Происходит это по следующей логике: если объект сообщения имеет поле type и существует React-компонент, соответствующий этому полю, то выбирается данный компонент, иначе же сообщению ставится в соответствие компонент Message.

Таким образом, реализован React-компонент MessageList, который необходим для отображения истории сообщений между пользователем и агентом.

Реализация компонента Message

Далее, реализуем React-компонент Message, который необходим для отображения текстовых и файловых сообщений в пользовательском интерфейсе. Реализация компонента Message представлена в листинге 30.

Листинг 30 – Реализация компонента Message

import React, { useCallback } from 'react';

import PropTypes from 'prop-types';

import cn from 'classnames';

import moment from 'moment';

import Linkify from 'react-linkify';

import { observer } from 'mobx-react-lite';

import { getAvatar } from '@chat/helpers/getAvatar';

import { getAbsoluteUrl } from '@chat/helpers/getAbsoluteUrl';

import { useStores } from '@chat/hooks/useStores';

import defaultAvatarUrl from '@chat/assets/default-avatar.svg';

import { ReactComponent as MessageErrored } from '@chat/assets/message-error.svg';

import { ReactComponent as MessageLoading } from '@chat/assets/message-loading.svg';

import { ReactComponent as FileIcon } from '@chat/assets/file-icon.svg';

import './Message.scss';

const linkDecorator = (href, text, key) => (

<a href={href} key={key} target="\_blank" rel="noreferrer">

{text}

</a>

);

function Message({ message, prevMessage }) {

const { Chat } = useStores();

const { error, token, file, isLoading, u: user = {}, attachments = [] } = message;

const [attachment] = attachments;

const fileUrl = attachment?.title\_link && getAbsoluteUrl(attachment?.title\_link);

const isMessageByVisitor = token && Chat.visitor?.token === token;

const displayError = isMessageByVisitor && error;

const displayUsername = !isMessageByVisitor && user.username;

const messageDate = new Date(message.ts ?? Date.now());

const prevMessageDate = new Date(prevMessage?.ts ?? Date.now());

const displayDateAndTime = !prevMessage || messageDate.getDate() !== prevMessageDate.getDate();

const onClickOnFile = useCallback(

(event) => {

if (!fileUrl) {

event.preventDefault();

}

},

[fileUrl]

);

return (

<div className="message-list\_\_item">

{displayDateAndTime && <p className="message-time">{moment(messageDate).format('D MMM, h:mm a')}</p>}

<div className={cn('message', { message\_self: isMessageByVisitor })}>

<div className="message\_\_body">

{!isMessageByVisitor && (

<div className="message\_\_avatar">

<img src={user?.username ? getAvatar(user.username) : defaultAvatarUrl} alt="user avatar" />

</div>

)}

{displayError && <MessageErrored className="message\_errored" />}

{!displayError && isLoading && <MessageLoading className="message\_loading" />}

<div className="message\_\_text">

{displayUsername && <p className="message\_\_username">{user.username}</p>}

{file && (

<p className={cn('message\_\_file', { 'message\_\_file\_not-loaded': !fileUrl })}>

<FileIcon />

<> </>

<a href={fileUrl ?? '#'} onClick={onClickOnFile} target="\_blank" rel="noreferrer">

{file.name}

</a>

</p>

)}

<Linkify componentDecorator={linkDecorator}>{message.msg}</Linkify>

</div>

</div>

<div className="message\_\_footer">{displayError && <div className="message\_\_error">{error.title}</div>}</div>

</div>

</div>

);

}

Message.defaultProps = {

prevMessage: undefined,

};

const MessageType = PropTypes.shape({

\_id: PropTypes.string.isRequired,

msg: PropTypes.string.isRequired,

ts: PropTypes.oneOfType([PropTypes.string, PropTypes.number]),

token: PropTypes.string,

isLoading: PropTypes.bool,

u: PropTypes.shape({

username: PropTypes.string,

}),

error: PropTypes.shape({

title: PropTypes.string.isRequired,

text: PropTypes.string,

}),

file: PropTypes.shape({

name: PropTypes.string.isRequired,

}),

attachments: PropTypes.arrayOf(

PropTypes.shape({

title\_link: PropTypes.string.isRequired,

})

),

});

Message.propTypes = {

message: MessageType.isRequired,

prevMessage: PropTypes.shape({

ts: PropTypes.oneOfType([PropTypes.string, PropTypes.number]),

}),

};

export default observer(Message);

Реализованный React-компонент содержит логику для отображения текстовых и файловых сообщений как от агента, так и от пользователя. Компонент разбирает объект сообщения и на основе его данных отображает различные состояния сообщения, такие как загрузка или ошибка отправки. Помимо этого, компонент получает объект предыдущего сообщения и если день его отправки не совпадает с днем отправки текущего сообщения, то над текущим сообщением выводится дата и время его отправки, как это отображено в предоставленном дизайне.

Таким образом, реализован React-компонент Message, который необходим для отображения текстовых и файловых сообщений в пользовательском интерфейсе.

Реализация компонента FeedbackForm

Следующим реализуем React-компонент FeedbackForm, который необходим для отображения и обработки формы оценки качества обслуживания в пользовательском интерфейсе. Реализация компонента FeedbackForm представлена в листинге 31.

Листинг 31 – Реализация компонента FeedbackForm

import React, { useCallback } from 'react';

import cn from 'classnames';

import PropTypes from 'prop-types';

import { observer } from 'mobx-react-lite';

import { FeedbackForm as FeedbackFormClass } from '@chat/store/FeedbackForm';

import thumbUp from '@chat/assets/thumb-up.png';

import thumbDown from '@chat/assets/thumb-down.png';

import './FeedbackForm.scss';

function FeedbackForm({ message: { rating, setRating } }) {

const onThumbUp = useCallback(() => setRating('1'), [setRating]);

const onThumbDown = useCallback(() => setRating('0'), [setRating]);

return (

<div className="message-list\_\_item">

<p className="message-time">Did this help you?</p>

<div className={cn('message feedback-form', { 'feedback-form\_disabled': rating !== undefined })}>

<div className="message\_\_body">

<div className="message\_\_text">

<div>

<div onClick={onThumbUp} className={cn('feedback-item', { selected: rating === '1' })}>

<img src={thumbUp} alt="Thumb up" />

</div>

</div>

<div>

<div onClick={onThumbDown} className={cn('feedback-item', { selected: rating === '0' })}>

<img src={thumbDown} alt="Thumb down" />

</div>

</div>

</div>

</div>

</div>

</div>

);

}

FeedbackForm.propTypes = {

message: PropTypes.instanceOf(FeedbackFormClass).isRequired,

};

export default observer(FeedbackForm);

Реализованный React-компонент соответствует объекту хранилища FeedbackForm, который представляет с себя специализированное сообщение и имеет поле type, для того чтобы в React-компоненте MessageList его можно было сопоставить с реализованным React-компонентом FeedbackForm.

В данном React-компоненте содержатся два основных элемента для выбора оценки качества обслуживания, при нажатии на которые происходит вызов соответствующего обработчика в компоненте хранилища FeedbackForm.

Таким образом, реализован React-компонент FeedbackForm, необходимый для отображения и обработки формы оценки качества обслуживания в пользовательском интерфейсе.

Реализация компонента InputBox

Реализуем React-компонент InputBox, который необходим для отображения и обработки блока ввода сообщения в пользовательском интерфейсе. Реализация компонента InputBox представлена в листинге 32.

Листинг 32 – Реализация компонента InputBox

import React, { useCallback, useState, useRef } from 'react';

import { observer } from 'mobx-react-lite';

import TextareaAutosize from 'react-textarea-autosize';

import cn from 'classnames';

import { useStores } from '@chat/hooks/useStores';

import { ReactComponent as FileIcon } from '@chat/assets/file-icon.svg';

import { ReactComponent as SendMessageIcon } from '@chat/assets/send-message.svg';

import './InputBox.scss';

function InputBox() {

const { Chat, Messages } = useStores();

const [text, setText] = useState('');

const fileInput = useRef();

const { settings } = Chat;

const textLength = text.trim().length;

const maxLengthNotExceeded = !settings.maxMessageLength || settings.maxMessageLength >= textLength;

const canBeSent = maxLengthNotExceeded && textLength > 0;

const onTextChange = useCallback((event) => {

setText(event.target.value);

}, []);

const onSubmit = useCallback(() => {

if (canBeSent) {

Messages.sendMessage(text.trim());

setText('');

}

}, [canBeSent, text, Messages]);

const onKeyDown = useCallback(

(event) => {

if (event.key === 'Enter') {

event.preventDefault();

onSubmit();

}

},

[onSubmit]

);

const onFileSelect = useCallback(

(event) => {

const [file] = event.target.files;

if (file !== undefined) {

Messages.sendMessage('', file);

}

},

[Messages]

);

const openFileDialog = useCallback(() => {

fileInput.current.click();

}, []);

return (

<div className="input-box">

<ul className="input-box\_\_actions">

<li>

<button onClick={openFileDialog} type="button" className="input-box\_\_attach-file" title="Attach file">

<FileIcon />

</button>

<input type="file" ref={fileInput} onChange={onFileSelect} hidden />

</li>

</ul>

<TextareaAutosize

className="input-box\_\_input"

placeholder="Type a message..."

value={text}

onChange={onTextChange}

onKeyDown={onKeyDown}

minRows={1}

maxRows={4}

/>

<button className={cn('input-box\_\_send-message', { active: canBeSent })} onClick={onSubmit} type="button" title="Send message">

<SendMessageIcon />

</button>

</div>

);

}

export default observer(InputBox);

Реализованный React-компонент содержит следующую логику:

* отображает кнопку для отправки файла;
* отображает многострочное поле ввода, реализуемое библиотекой TextareaAutosize;
* отображает кнопку отправки сообщения;
* обрабатывает нажатие на кнопку «прикрепить файл» и вызывает диалоговое окно для выбора файла. После выбора файла вызывает соответствующий метод отправки файла в хранилище Messages, передавая его туда;
* если пользователь нажал клавишу Enter и поле ввода сообщения не пустое, то выполняется отправка сообщения;
* если поле ввода сообщения пустое или длина введенного сообщения превосходит максимально разрешенную, то кнопка отправки сообщения не активна и сообщение невозможно отправить;
* обрабатывает нажатие на кнопку «отправить сообщение» и, если поле сообщение не пустое и не превосходит максимально разрешенную длину, то вызывается соответствующий метод в хранилище Messages, в который передается текст, содержащийся в поле ввода;
* после отправки сообщения поле ввода очищается.

Таким образом, реализован React-компонент InputBox, необходимый для отображения и обработки блока ввода сообщения в пользовательском интерфейсе.

## Реализация механизма кроссдоменного обмена данными между фреймами

В листинге 33 представлена реализация класса PostEvents, который предоставляет возможность кроссдоменного обмена данными между родительским окном и встраиваемым веб-чатом.

Листинг 33 – Реализация класса PostEvents

import EventEmitter from 'eventemitter3';

export class PostEvents extends EventEmitter {

targetWindow;

constructor(targetWindow) {

super();

this.targetWindow = targetWindow;

window.addEventListener('message', this.onMessage);

}

onMessage = (event) => {

if (event.source !== this.targetWindow || !event.data?.action) {

return;

}

const { action, data = {} } = event.data;

this.emit(action, data);

};

send(action, data = {}) {

if (!this.targetWindow) {

throw new Error('You should specify targetWindow!');

}

this.targetWindow.postMessage({ action, data }, '\*');

}

}

В реализованном классе объявлено одно поле targetWindow, в котором будет содержаться ссылка на объект целевого окна, с которым будет происходить обмен данными.

В классе объявлены следующие методы:

* constructor – специальный метод, вызываемый при создании экземпляра класса. Данный метод принимает один аргумент, в котором должна находиться ссылка на объект окна, с которыми будет происходить обмен данными. В самом конструкторе выполняется присваивание переданного объекта в targetWindow и регистрация обработчика события message для текущего окна. Событие message вызывается при получении текущим окном сообщения от какого-либо другого окна;
* onMessage – обработчик полученных от целевого окна сообщений. Все полученные сообщения должны иметь два поля: название события (action) и данные (data). Данный метод извлекает оба эти поля и генерирует событие с названием, указанным в action, передавая в него data;
* send – метод для отправки сообщений в целевое окно. Для отправки сообщений используется предоставляемый браузером метод «window.postMessage()».

Таким образом, реализован класс PostEvents, предоставляющий возможность кроссдоменного обмена данными между родительским окном и встраиваемым веб-чатом.

## Реализация Iframe-приложения

Реализация хранилища для управления состоянием

Основным и единственным компонентом хранилища для данного приложения является класс IframeStore, реализация которого представлена в листинге 34.

Листинг 34 – Реализация класса IframeStore

import { makeObservable, action, observable } from 'mobx';

import { PostEvents } from '@core/PostEvents';

import { onlineEvents, offlineEvents } from '@/config';

class IframeStore {

@observable isOpen = false;

@observable unreadMessages = 0;

PostEventor;

constructor() {

makeObservable(this);

}

onLoad = (iframe) => {

this.PostEventor = new PostEvents(iframe.contentWindow);

this.PostEventor.on('toggle-chat', this.toggleChat);

this.PostEventor.on('messages.received-agent-message', this.onNewAgentMessage);

this.PostEventor.send('chat.props', window.LiveChatWidget);

this.PostEventor.send('global.parent-window-props', {

location: {

origin: window.location.origin,

href: window.location.href,

hostname: window.location.hostname,

host: window.location.host,

},

});

onlineEvents.forEach((event) => document.addEventListener(event, this.onTriggerOnlineEvent));

offlineEvents.forEach((event) => window.addEventListener(event, this.onTriggerOfflineEvent));

};

onTriggerOnlineEvent = () => {

this.PostEventor.send('visitor.online-status', { isOnline: true });

}

onTriggerOfflineEvent = () => {

this.PostEventor.send('visitor.online-status', { isOnline: false });

}

@action.bound

toggleChat({ value } = {}) {

this.isOpen = value ?? !this.isOpen;

this.PostEventor.send('chat.is-open', { value: this.isOpen });

if (this.isOpen) {

this.unreadMessages = 0;

}

}

@action.bound

onNewAgentMessage() {

if (!this.isOpen) {

this.unreadMessages += 1;

}

}

}

export default new IframeStore();

В классе IframeStore объявлены следующие поля:

* «@observable isOpen» – поле, обозначающее открыт или скрыт чат в данный момент. Значение «true» обозначает, что чат открыт. Значение «false» – закрыт;
* «@observable unreadMessages» – поле, содержащее количество непрочитанных пользователем сообщений. Сообщение считается непрочитанным, если в момент его получения пользователем веб-чат был в скрытом состоянии;
* PostEventor – поле, необходимое для хранения объекта класса PostEvents.

Помимо полей, в классе IframeStore объявлены следующие методы:

* onLoad – обработчик события загрузки iframe HTML элемента, содержащего приложение веб-чата. В данном обработчике создается объект класса PostEvents, регистрируются обработчики событий и передаются данные с веб-страницы продукта в веб-чат. Данные, которые необходимо передать с веб-страницы продукта в веб-чат, должны находиться в объекте «window.LiveChatWidget»;
* onTriggerOnlineEvent – обработчик события, необходимого для отслеживания онлайн статуса пользователя;
* onTriggerOfflineEvent – обработчик события, необходимого для отслеживания онлайн статуса пользователя;
* toggleChat – метод, используемый для изменения состояния чата с «открыт» или «скрыт» на противоположное;
* onNewAgentMessage – обработчик события, сообщающего о получении нового сообщения от агента.

Таким образом, реализован основной и единственный компонент хранилища класс IframeStore, необходимый для хранения и обработки состояния Iframe-приложения и обмена данными с приложением веб-чата.

Реализация кнопки открытия и сокрытия веб-чата

Реализуем React-компонент ChatToggler, необходимый для отображения в пользовательском интерфейсе кнопки открытия и сокрытия веб-чата. Реализация компонента ChatToggler представлена в листинге 35.

Листинг 35 – Реализация React-компонента ChatToggler

import React from 'react';

import { observer } from 'mobx-react-lite';

import { useStores } from '@iframe/hooks/useStores';

import { useDelay } from '@core/hooks/useDelay';

import { CSSAnimation } from '@core/components/CSSAnimation';

import { ReactComponent as HideChatIcon } from '@iframe/assets/hide-chat.svg';

import { ReactComponent as OpenChatIcon } from '@iframe/assets/open-chat.svg';

import './ChatToggler.module.scss';

const iconAnimationTime = 150;

function ChatToggler() {

const { IframeStore } = useStores();

const { isOpen, toggleChat, unreadMessages } = IframeStore;

const isOpenDelayed = useDelay(isOpen, iconAnimationTime);

const unreadMessagesDelayed = useDelay(unreadMessages, 300);

const unreadMessagesValue = unreadMessages > 0 ? unreadMessages : unreadMessagesDelayed;

return (

<CSSAnimation in timeout={300} classNames="togglerIn" appear unmountOnExit>

<div className="livechat-toggler">

<button onClick={toggleChat} type="button">

<CSSAnimation in={unreadMessages > 0} timeout={300} classNames="animate" hideOnExit>

<div className="has-unread-messages">{unreadMessagesValue > 9 ? '9+' : unreadMessagesValue}</div>

</CSSAnimation>

<CSSAnimation in={!isOpen ? !isOpenDelayed : false} timeout={iconAnimationTime} classNames="zoomIn" hideOnExit>

<OpenChatIcon className="open-chat" />

</CSSAnimation>

<CSSAnimation in={isOpen ? isOpenDelayed : false} timeout={iconAnimationTime} classNames="zoomIn" hideOnExit>

<HideChatIcon className="hide-chat" />

</CSSAnimation>

</button>

</div>

</CSSAnimation>

);

}

export default observer(ChatToggler);

Реализованный компонент содержит кнопку, которая внутри содержит две иконки, каждая из которых отображается в зависимости от состояния чата «открыт» или «скрыт». При нажатии на кнопку вызывается метод toggleChat, реализованный в хранилище IframeStore. Помимо этого, кнопка содержит элемент, в котором отображается количество непрочитанных пользователем сообщений.

Таким образом, реализован React-компонент ChatToggler, который необходим для отображения в пользовательском интерфейсе кнопки открытия и сокрытия веб-чата.

Реализация компонента ChatIframe

Реализуем основной React-компонент ChatIframe, который будет встраивать iframe HTML-элемент с веб-чатом в веб-страницу продукта. Реализация компонента ChatIframe представлена в листинге 36.

Листинг 36 – Реализация React-компонента ChatIframe

import React, { useCallback, useRef } from 'react';

import { observer } from 'mobx-react-lite';

import cn from 'classnames';

import { useStores } from '@iframe/hooks/useStores';

import { widgetUrl, defaultClassNames } from '@iframe/config';

import { CSSAnimation } from '@core/components/CSSAnimation';

import { ChatToggler } from '@iframe/components/ChatToggler';

import './ChatIframe.module.scss';

function ChatIframe() {

const iframeRef = useRef();

const { IframeStore } = useStores();

const { onLoad, isOpen } = IframeStore;

const onIframeLoad = useCallback(() => onLoad(iframeRef.current), [onLoad]);

const classNames = cn('livechat-widget', defaultClassNames);

return (

<div className={classNames}>

<CSSAnimation in={isOpen} timeout={300} classNames="livechatIn" hideOnExit>

<div className="livechat-iframe">

<iframe src={widgetUrl} ref={iframeRef} onLoad={onIframeLoad} title="LiveChat Widget" />

</div>

</CSSAnimation>

<ChatToggler />

</div>

);

}

export default observer(ChatIframe);

В реализованном компоненте содержаться в элементе: iframe с веб-чатом и кнопка открытия и сокрытия чата. На событие загрузки iframe элемента установлен обработчик onIframeLoad, который вызывает метод onLoad, реализованный в IframeStore, передавая ему ссылку на iframe элемент, чтобы из него можно было получить ссылку на окно веб-чата для реализации кроссдоменного обмена данными.

Таким образом, реализован React-компонент ChatIframe, который необходим для встраивания iframe HTML-элемента с веб-чатом в веб-страницу продукта.

## Реализация скрипта загрузки Iframe-приложения

Реализация скрипта, необходимого для загрузки Iframe-приложения в веб-страницу продукта представлена в листинге 37.

Листинг 37 – Реализация скрипта загрузки Iframe-приложения

(function(url) {

window.LiveChatWidget = window.LiveChatWidget || {};

window.LiveChatWidget.iframe = window.LiveChatWidget.iframe || {};

window.LiveChatWidget.iframe.url = url;

const js = document.createElement('script');

js.async = true;

js.src = new URL('/iframe.js', url).toString();

document.body.append(js);

})('http://localhost:4000');

Реализованный скрипт является немедленно вызываемой функцией, принимающей параметр url, в котором должен быть указан URL адрес, с которого необходимо загрузить приложение встраивания. В теле функции создается HTML-элемент script, в атрибут src которого устанавливается переданный URL адрес с дописанным к нему ресурсом «iframe.js», и добавляется в элемент body документа веб-страницы. После добавления элемента script в body, запустится загрузка Iframe-приложения, после которой оно начнет выполняться и внедрит в тело веб-страницы iframe элемент с веб-чатом и кнопку для его открытия и сокрытия.

Таким образом, реализован скрипт, необходимый для загрузки Iframe-приложения в веб-страницу продукта.

## Выводы по разделу

В данном разделе описан процесс разработки встраиваемого веб-чата поддержки пользователей на базе платформы «Rocket.Chat», который предоставляет возможность партнерам и клиентам компании общаться с сотрудником поддержки в режиме реального времени на веб-страницах продуктов компании.

# Экономический раздел

Основной задачей данного раздела является обоснование целесообразности разработки продукта, с точки зрения его экономической и технической эффективности.

## Расчет затрат на разработку и ввод в эксплуатацию программного продукта

Для оценки эффективности разработанного продукта, приняты во внимание затраты на его создание и ввод в эксплуатацию.

Основной частью затрат в области разработки программ является выплата заработной платы сотрудникам. Для измерения этих затрат используется принятая универсальная единица трудоемкости, которая измеряется в человеко-днях, человеко-месяцах или человеко-годах.

Для реализации проекта компанией выделен один специалист на три месяца работы. Месячная заработная плата выделенного сотрудника составляет 90 тыс. рублей. Однако, с заработной платы сотрудника государство удерживает различные налоги и отчисления, перечислим их:

* налог на доход физических лиц 13%;
* взносы в Пенсионный фонд России 22%;
* взносы в Фонд социального страхования 2,9%;
* взносы в Федеральный фонд обязательного медицинского страхования 5,1%;
* взносы на страхование от несчастных случаев и профессиональных заболеваний 0,2%.

В итоге, месячные затраты работодателя на выплату заработной платы разработчика составляют:

90 000 × 1,432 = 128880 руб.

Таким образом, затраты на выплату заработной платы сотруднику за три месяца составляют:

3 × 128880 = 386640 руб.

Однако, помимо затрат на выплату заработной платы сотруднику, существуют ещё расходы на ввод в эксплуатацию реализованного программного продукта. Поскольку в данном случае программным продуктом является веб-приложение, то для его ввода в эксплуатацию требуется выделенный сервер. Но у компании имеется свой дата-центр, поэтому для ввода в эксплуатацию реализованного программного продукта дополнительных затрат на аренду или покупку выделенного сервера не требуется.

Таким образом, суммарные затраты на разработку и ввод в эксплуатацию программного продукта составят 386640 рублей.

## Расчет экономического эффекта

Основным экономическим эффектом от реализации данного программного продукта можно считать сокращение затрат на текущее используемое решение от компании Intercom. Реализованный программный продукт не требует обслуживания, однако, учитывая, что в процессе использования могут быть обнаружены дефекты и недоработки, компания дополнительно предусмотрела выделение 180 тыс. рублей на сопровождение продукта в первый год его эксплуатации.

Исходя из того, что использование веб-чата от Intercom обходится компании в примерно в 40 тыс. долларов в год, рассчитано сокращение затрат в первый год после внедрения реализованного программного продукта (вместо решения от компании Intercom), учитывая затраты на разработку, ввод в эксплуатацию и сопровождение программного продукта:

40000 × 73 – 386640 – 180000 = 2 353 360 рублей в первый год,

где 73 – округленный курс доллара по отношению к рублю в настоящий момент.

Таким образом, срок окупаемости реализованного программного продукта в первый год составит:

(386640+180000)/(40000/12\*73) = 2,3 месяца

В последующие года программный продукт не требует сопровождения, соответственно, затраты на его содержание будут равны нулю, а расходы компании сократятся на:

40000 \* 73 = 2 920 000 рублей в год,

где 73 – округленный курс доллара по отношению к рублю в настоящий момент.

## Сопутствующие технико-экономические результаты

Компанией установлено, что в результате реализации и внедрения встраиваемого веб-чата поддержки пользователей на базе платформы «Rocket.Chat» появились сопутствующие технико-экономические результаты:

* увеличился индекс удовлетворенности партнеров и пользователей на 15%;
* уменьшился объем загружаемых ресурсов на 35%;
* уменьшилось время загрузки чата на 30%;
* увеличилась скорость разработки нового функционала, поскольку реализованный программный продукт является собственной разработкой компании;
* затраты на использование продуктов сторонних компаний снижены до нуля.

## Выводы по разделу

Таким образом, обоснована целесообразность реализации программного продукта, разработка и внедрение которого привела к появлению внушительного экономического эффекта в размере сокращения расходов компании в первый год эксплуатации на 2 353 360 рублей, а в последующие года – на 2 920 000 рублей.

# Обеспечение информационной безопасности

Защита информации от несанкционированного доступа является одним из главных требований в вопросе обеспечения информационной безопасности. Поскольку реализованный встраиваемый веб-чат является лишь пользовательским интерфейсом для взаимодействия пользователя с веб-сервером платформы «Rocket.Chat», то он не отвечает за хранение и защиту информации, поскольку все данные, которые он собирает, либо хранятся на компьютере пользователя, либо отправляются на сервер «Rocket.Chat». Таким образом, все мероприятия по защите информации пользователя от несанкционированного доступа выполняются на стороне платформы «Rocket.Chat».

Однако, все же существуют атаки, которые направлены на уязвимости пользовательского интерфейса. Одной из таких атак является XSS-атака – это тип атаки на веб-системы, заключающийся во внедрении в выдаваемую веб-системой страницу вредоносного кода (который будет выполнен на компьютере пользователя при открытии им этой веб-страницы) и взаимодействии этого кода с веб-сервером злоумышленника. Защиту от данного типа атак в реализованном веб-чате обеспечивает библиотека «React.js», которая осуществляет автоматическое экранирование всех выводимых данных, тем самым предотвращая выполнение любого выводимого вредоносного кода.

Стоит отметить, что все сотрудники компании работают на технических устройствах компании Apple, которая самостоятельно обеспечивает защиту устройств и информации от несанкционированного доступа.

ЗАКЛЮЧЕНИЕ

В ходе выполнения выпускной квалификационной работы спроектирован и разработан встраиваемый веб-чат поддержки пользователей на базе платформы «Rocket.Chat».

Для достижения цели выполнены следующие задачи:

* выполнен анализ достоинств и недостатков встраиваемых веб-чатов, используемых компанией на текущий момент;
* исследовано устройство и определены функциональные возможности «Rocket.Chat LiveChat API»;
* проанализирован разработанный компанией собственный дизайн веб-чата;
* исследованы подходы к разработке встраиваемых веб-чатов;
* сформулированы требования к реализуемому встраиваемому веб-чату;
* выбраны средства разработки для реализации встраиваемого веб-чата;
* спроектирована и разработана общая архитектура встраиваемого веб-чата и показаны системы, с которыми он взаимодействует;
* спроектирована и разработана программная архитектура встраиваемого веб-чата;
* спроектированы и разработаны основные алгоритмы работы программы;
* разработана программная реализация спроектированного встраиваемого веб-чата;
* выполнена оценка технико-экономической эффективности внедрения.

Таким образом, цель и задачи выпускной квалификационной работы выполнены в полном объеме.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. REST API - Rocket.Chat Developer. [Электронный ресурс] – URL: https://developer.rocket.chat/api/rest-api/ (дата обращения 27.04.2021).
2. Realtime API - Rocket.Chat Developer. [Электронный ресурс] – URL: https://developer.rocket.chat/api/realtime-api (дата обращения 27.04.2021).
3. Тег <iframe> | htmlbook.ru. [Электронный ресурс] – URL: http://htmlbook.ru/html/iframe/ (дата обращения 28.04.2021).
4. Same-origin policy - Веб-безопасность | MDN. [Электронный ресурс] – URL: https://developer.mozilla.org/ru/docs/Web/Security/Same-origin\_policy (дата обращения 29.04.2021).
5. Window.postMessage() - Интерфейсы веб API | MDN. [Электронный ресурс] – URL: https://developer.mozilla.org/ru/docs/Web/API/Window/postMessage (дата обращения 29.04.2021).
6. Entry Points | webpack. [Электронный ресурс] – URL: https://webpack.js.org/concepts/entry-points/ (дата обращения 04.05.2021).
7. Output | webpack. [Электронный ресурс] – URL: https://webpack.js.org/concepts/output/ (дата обращения 04.05.2021)
8. Loaders | webpack. [Электронный ресурс] – URL: https://webpack.js.org/concepts/loaders/ (дата обращения 04.05.2021)
9. Plugins | webpack. [Электронный ресурс] – URL: https://webpack.js.org/concepts/plugins/ (дата 04.05.2021)
10. ESLint - Pluggable JavaScript linter. [Электронный ресурс] – URL: https://eslint.org/ (05.05.2021)

ПРИЛОЖЕНИЕ А

Техническое задание

Утверждаю

Зав. кафедрой: д-р экон. наук, проф.

Файзрахманов Р.А.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_\_\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

Разработка встраиваемого веб-чата поддержки пользователей на базе платформы Rocket.Chat

Техническое задание на 9 листах

Разработал студент гр. РИС-17-1б

Ширинов Р. Р. \_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

Действует с 1 февраля 2021 г.

Содержание

[1 Общие положения 117](#_Toc74597682)

[1.1 Полное наименование программы и ее условное обозначение 117](#_Toc74597683)

[1.2 Шифр темы 117](#_Toc74597684)

[1.3 Наименование предприятия разработчика и заказчика 117](#_Toc74597685)

[1.4 Основание для разработки 117](#_Toc74597686)

[1.5 Источники финансирования 117](#_Toc74597687)

[1.6 Плановые сроки начала и окончания работ 117](#_Toc74597688)

[1.7 Порядок оформления и предъявления результатов работ 118](#_Toc74597689)

[2 Назначение и цели реализации программы 118](#_Toc74597690)

[2.1 Назначение программы 118](#_Toc74597691)

[2.2 Цель и задачи реализации программы 118](#_Toc74597692)

[3 Требования к программе 119](#_Toc74597693)

[3.1 Требования к программе в целом 119](#_Toc74597694)

[3.2 Требования к функциям 120](#_Toc74597695)

[3.3 Требования к организации входных данных 121](#_Toc74597696)

[3.4 Требования к организации выходных данных 121](#_Toc74597697)

[3.5 Требования к видам обеспечения 121](#_Toc74597698)

[4 Состав и содержание работ по созданию программы 121](#_Toc74597699)

[4.1 Этапы создания программы 121](#_Toc74597700)

[5 Порядок контроля и приемки 122](#_Toc74597701)

[5.1 Виды испытаний 122](#_Toc74597702)

[5.2 Общие требования к приемке работы 122](#_Toc74597703)

[6 Требования к документированию 122](#_Toc74597704)

[7 Источники разработки 123](#_Toc74597705)

1. Общие положения
   1. Полное наименование программы и ее условное обозначение

Настоящее Техническое задание определяет требования и порядок разработки встраиваемого веб-чата поддержки пользователей на базе платформы «Rocket.Chat».

* 1. Шифр темы

Настоящее Техническое задание разработано в рамках выполнения выпускной квалификационной работы «Разработка встраиваемого веб-чата поддержки пользователей на базе платформы Rocket.Chat». Кафедра ИТАС, специальность 09.03.04 – Программная инженерия (ПИ).

* 1. Наименование предприятия разработчика и заказчика

Заказчиком является ООО «Систем Проджектс».

Исполнителем является студент группы РИС-17-1б Ширинов Рустам Рамазанович.

* 1. Основание для разработки

Проект выполняется на основании темы выпускной квалификационной работы, согласованной с заведующим кафедрой ИТАС ПНИПУ, Файзрахмановым Р.А.

* 1. Источники финансирования

Финансирование работ осуществляется из средств ООО «Систем Проджектс».

* 1. Плановые сроки начала и окончания работ

Срок реализации программного обеспечения регламентирован действующим учебным планом и предполагает сдачу проекта подсистемы до 23 июня 2021 года.

* 1. Порядок оформления и предъявления результатов работ

Работы по созданию программы производятся и принимаются поэтапно. По окончании каждого из этапов работ, установленных Учебным планом, студенты представляют заведующему кафедрой соответствующие результаты. Заведующий кафедрой оценивает работу.

1. Назначение и цели реализации программы
   1. Назначение программы

Программа предназначена для оказания онлайн поддержки пользователям и партнерам на веб-страницах продуктов компании.

* 1. Цель и задачи реализации программы

Целью данной работы является проектирование архитектуры и разработка встраиваемого веб-чата поддержки пользователей на базе платформы «Rocket.Chat».

Задачи:

* выполнить анализ достоинств и недостатков встраиваемых веб-чатов, используемых компанией на текущий момент;
* исследовать устройство и определить функциональные возможности «Rocket.Chat LiveChat API»;
* проанализировать разработанный компанией собственный дизайн веб-чата;
* исследовать подходы к разработке встраиваемых веб-чатов;
* сформулировать требования к реализуемому встраиваемому веб-чату;
* выбрать средства разработки для реализации встраиваемого веб-чата;
* спроектировать и разработать общую архитектуру встраиваемого веб-чата и показать системы, с которыми он взаимодействует;
* спроектировать и разработать программную архитектуру встраиваемого веб-чата;
* спроектировать и разработать основные алгоритмы работы программы;
* разработать программную реализацию спроектированного встраиваемого веб-чата;
* провести оценку технико-экономической эффективности внедрения.

1. Требования к программе
   1. Требования к программе в целом

Общие требования к программе:

* выполнение всех функций, предусмотренных настоящим документом в пункте 3.2;
* выполнение требований (численности, квалификации персонала; надежности; безопасности, эргономике и дополнительных требований при наличии) к программе, предусмотренных настоящим документом в пунктах 3.1.1., 3.1.2., 3.1.3., 3.1.4., 3.1.5. соответственно.
  + 1. Требования к численности и квалификации персонала

Не предъявляются.

* + 1. Требования к надежности программы

Программа должна безошибочно выполнять все требуемые функции, и быть пригодной для эксплуатации. Программа должна быть устойчивой, то есть способной правильно выполнять запланированные действия, не смотря на случайные отклонения.

* + 1. Требования к безопасности

Не предъявляются.

* + 1. Требования по эргономике

Для использования системы необходим браузер, поддерживающий ECMAScript 6 и выше.

* + 1. Дополнительные требования

Дополнительных требований к работе программы не предъявляется.

* 1. Требования к функциям

Перечень функциональных требований к встраиваемому веб-чату:

* веб-чат должен осуществлять регистрацию новых пользователей;
* веб-чат должен обновлять информацию о зарегистрированном пользователе, если какие-либо его данные изменились;
* веб-чат должен создавать комнату при первом обращении пользователя;
* веб-чат должен отслеживать онлайн статус пользователя;
* веб-чат должен позволять пользователю осуществлять обмен текстовыми сообщениями с агентом в режиме реального времени;
* веб-чат должен позволять пользователю отправлять и получать файлы;
* веб-чат должен позволять пользователю закрыть комнату самостоятельно;
* при отсутствии в текущей момент созданной комнаты, веб-чат должен отображать приветственное сообщение с текстом, полученным с сервера;
* после закрытия комнаты веб-чат должен выводить форму для оценки пользователем качества обслуживания;
* веб-чат должен иметь возможность встраивания на другие веб-страницы в Интернете;
* веб-чат должен уметь принимать и обрабатывать данные, полученные от веб-страницы, на которую он встроен;
* веб-чат должен отправлять полученные настраиваемые поля от веб-страницы, на которую он встроен, на сервер;
* веб-чат должен иметь возможность сокрытия и открытия по специальной кнопке, отображенной в предоставленном в дизайне;
* веб-чат не должен перекрывать какой-либо рядом расположенный контент на веб-страницах, в которые он встроен.
  1. Требования к организации входных данных

Не предъявляются.

* 1. Требования к организации выходных данных

Любые выходные данные должны соответствовать дизайну, который был разработан специалистами компании и предоставлен исполнителю.

* 1. Требования к видам обеспечения
     1. Требования к техническому обеспечению

Для работы программы необходим выделенный сервер со следующими минимальными характеристиками:

* минимум 1 Гб свободного дискового пространства;
* минимум 256 Мб оперативной памяти;
* процессор, имеющий 2 ядра с частотой по 2.2 ГГц;
* операционная система Linux.
  + 1. Требования к ПО

Для работы программы необходимо использовать веб-сервер Nginx для отдачи статических файлов.

1. Состав и содержание работ по созданию программы
   1. Этапы создания программы

Этапы создания системы приведены в таблице А.1.

Таблица А.1 – Этапы создания программы

|  |  |  |
| --- | --- | --- |
| Наименование этапа | Начало | Конец |
| Исследование предметной области | 01.02.2021 | 26.02.2021 |
| Проектирование программного продукта | 01.03.2021 | 02.04.2021 |
| Разработка программного продукта | 05.04.2021 | 11.06.2021 |
| Представление работы на проверку | 14.06.2021 | 23.06.2021 |

1. Порядок контроля и приемки
   1. Виды испытаний

Программа подвергается испытаниям следующих видов:

* предварительные испытания;
* опытная эксплуатация;
* приемочные испытания.

Состав, объем и методы испытаний определяются документом «Программа и методика испытаний».

* 1. Общие требования к приемке работы

Сдача-приемка работ производится поэтапно, в соответствии с Календарным графиком, и осуществляется разработчиками приложения.

Испытания приложения должны осуществляться в соответствии с документом "Программа и методика испытаний", который должен устанавливать необходимый и достаточный объем испытаний, обеспечивающий требуемый уровень достоверности получаемых результатов.

Ход проведения приемо-сдаточных испытаний документируется в Протоколе проведения испытаний.

1. Требования к документированию

Требований к программной документации не предъявляется.

1. Источники разработки

В качестве источников по разработке данного технического задания были использованы:

* ГОСТ 34.602-89 «Техническое задание на создание АС»;
* ГОСТ 34.003-90 «Автоматизированные системы. Термины и определения»;
* документация и материалы на аналогичные системы;
* документация из курсовых и дипломных работ студентов ПНИПУ на аналогичную тему.