МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

РАЗРАБОТКА КЛИЕНТСКОЙ ЧАСТИ ВЕБ-ПЛАТФОРМЫ ДЛЯ СОЗДАНИЯ ВИДЕО КОНТЕНТА

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 411 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Марголина Ефима Григорьевича

Научный руководитель
зав. каф., к. ф.-м. н., доцент _____ С. В. Миронов

Заведующий кафедрой
к. ф.-м. н., доцент _____ С. В. Миронов

Саратов 2026

Актуальность темы. В последние годы видеоконтент стал доминирующей формой информации в интернете, однако процесс его производства остаётся сложным и трудоёмким, включающим множество этапов и участие разных специалистов. На практике команды сталкиваются с проблемой разрозненности инструментов: управление задачами ведётся в одних системах, обмен файлами — через облачные диски, коммуникация — в мессенджерах, а аналитика — в отдельных сервисах. Существующие на рынке решения (Trello, Asana, Jira) не предлагают специализированных функций для медиа-производства, что вынуждает команды использовать несколько разрозненных инструментов. Таким образом, создание веб-платформы, объединяющей управление проектами, рабочими процессами и аналитикой в едином пространстве, является актуальной задачей, направленной на повышение эффективности медиа-команд.

Цель бакалаврской работы — разработка и внедрение веб-платформы для автоматизации командной работы над видеоконтентом, включающей управление этапами производства, файловый менеджер, командный чат и аналитику каналов.

Поставленная цель определила **следующие задачи**:

1. Разработать клиентскую часть веб-приложения: спроектировать информационную архитектуру интерфейса, реализовать адаптивный пользовательский интерфейс, интегрировать систему аутентификации с JWT-токенами.
2. Реализовать вспомогательный сервис на Python для получения статистики с YouTube Data API и VK API, обеспечивающий унифицированный формат ответа для фронтенда.

Методологические основы исследования представлены в работах Д. Флэнагана (JavaScript), Р. Мартина (чистая архитектура), П. Макфедриса (вебдизайн), а также в документации по технологиям JavaScript, gRPC, Protocol Buffers, CSS Grid и Flexbox.

Практическая значимость работы заключается в создании готового к использованию веб-сервиса «ROLLIKI», который развёрнут в интернете и может применяться реальными командами для производства видеоконтента. Разработанная архитектура клиентской части может служить основой для построения аналогичных систем управления медиа-проектами.

Структура и объём работы. Бакалаврская работа состоит из введения, двух разделов, заключения, списка использованных источников и пяти прило-

жений. Общий объём работы — 91 страница, из них 66 страниц — основное содержание, включающее 23 рисунка. Список использованных источников содержит 24 наименования.

Первый раздел «Анализ предметной области и обоснование технических решений» посвящён постановке задачи и анализу требований к сервису. Рассмотрены функциональные требования к веб-платформе «ROLLIKI», которая ориентирована на небольшие видеопроизводящие студии, блогерские команды и корпоративные медиа-отделы. Система должна предоставлять настраиваемый визуальный конвейер производства видео (workflow) с канбан-доской, отображающей движение задач по этапам, и возможностью назначения зависимостей между задачами, чтобы, например, этап монтажа автоматически блокировался до утверждения сценария. Предусмотрена система ролей и команд с четырьмя категориями: владелец проекта, оператор, дизайнер, редактор. Каждый участник может быть назначен ответственным за конкретные этапы. Интеграция с YouTube Analytics и VK API позволяет получать статистику по просмотрам, лайкам и динамике подписчиков как по каналу в целом, так и в разрезе каждого видео. Файловый менеджер организует цепочку передачи файлов между этапами с системой утверждения результатов: владелец или ответственный может подтвердить завершение этапа с возможностью загрузки файлов. Встроенная командная коммуникация в виде личных сообщений между участниками позволяет не покидать сервис для согласования правок. Все обмены данными между клиентом и сервером защищены с использованием JWT-токенов. На основе анализа существующих решений (Trello, Asana, Jira) сделан вывод о наличии пробела между мощными универсальными системами и узкоспециализированными инструментами, что подтверждает актуальность разработки.

Далее проведён анализ средств реализации клиентской части. Рассмотрены языки JavaScript и TypeScript — выбран чистый JavaScript благодаря опыту работы на предыдущих проектах и желанию углубить навыки именно в нативном JS. Для построения пользовательского интерфейса принято решение отказаться от фреймворков (React) в пользу нативного JavaScript с прямым управлением DOM через стандартные API, что позволяет сохранить контроль над производительностью и избежать избыточной сложности. Для взаимодействия с сервером выбран `Fetch API` как современный встроенный механизм браузера,

основанный на промисах. Однако для связи между сервисом аналитики и чатом выбран протокол gRPC-Web, так как он использует бинарный протокол на основе Protocol Buffers, что обеспечивает более высокую скорость сериализации и меньший объём передаваемых данных по сравнению с JSON, а также поддерживает двунаправленную потоковую передачу и автоматическую генерацию клиентского и серверного кода. Для хранения JWT-токена и данных пользователя используется `localStorage`. Для вёрстки применены HTML5, CSS Grid и Flexbox, что обеспечивает адаптивность интерфейса. Средой разработки выбран Visual Studio Code, управление версиями — Git с репозиторием на GitLab. Настроен CI/CD-пайплайн, а для обслуживания статических файлов используется веб-сервер Nginx с поддержкой HTTPS.

Описана общая архитектура фронтенда. Пользователь начинает работу со страницы входа. После аутентификации система проверяет наличие проектов: если их нет, пользователь направляется на создание первого проекта, если проекты есть — переходит к выбору нужного. Далее открывается основной раздел — проект, внутри которого доступны четыре ключевых раздела: «Видео», «Команда», «Аналитика» и «Настройки». Переход между этими разделами происходит с полной перезагрузкой страницы, что упрощает архитектуру и управление состоянием. Однако внутри каждого раздела ключевые операции выполняются без перезагрузки за счёт асинхронных запросов к API и динамического обновления DOM. Такая гибридная архитектура сочетает простоту реализации многостраничных сайтов с отзывчивостью, характерной для одностраничных приложений. Для сбора статистики с YouTube Data API и VK API выделен отдельный сервис аналитики на Python, что изолирует потенциальные проблемы с внешними API (сетевые задержки, сбои, ограничения по частоте запросов) от основной системы.

Второй раздел «Разработка клиентской части веб-приложения» содержит детальное описание реализации всех модулей.

Реализация страниц аутентификации. Страницы входа и регистрации реализованы классами `LoginPage` и `RegistrationPage`. В конструкторе выполняется поиск DOM-элементов через `document.getElementById`.
Метод `initEventListeners` навешивает обработчик отправки формы с отменой стандартного поведения (`e.preventDefault`) для асинхронной обработки. На странице регистрации функция `validateInputs` выполняет последователь-

ные проверки: заполнение всех полей, длина имени (от 3 до 20 символов), длина пароля (не менее 6 символов), совпадение паролей. При ошибке вызывается метод showError с отображением сообщения и анимацией встряски формы (shakeForm), которая добавляет CSS-класс shake с анимацией через @keyframes и удаляет его через setTimeout. Асинхронная отправка данных реализована через Fetch API. При успешном входе (статус 200) данные пользователя и JWT-токен сохраняются в localStorage с помощью setItem, после чего выполняется перенаправление на страницу профиля через window.location.href. При ошибке анализируется HTTP-статус: для 400 — «Пользователь с таким именем уже существует», для остальных — общее сообщение. Во время отправки запроса кнопка блокируется, текст скрывается, отображается спиннер (setLoading), что исключает повторную отправку формы.

Реализация страницы профиля. Страница профиля отображает все проекты, в которых участвует пользователь, а также приглашения от других проектов. Загрузка данных выполняется асинхронно в методе loadChannels. Два запроса — к эндпоинтам /api/connection/user/{userId}/accepted и /api/connection/user/{userId}/pending — выполняются параллельно с помощью Promise.all, что ускоряет загрузку страницы. Для отображения карточек необходимо загрузить детали каждого проекта по отдельности; все такие запросы также выполняются параллельно через Promise.all. После получения данных вызывается метод displayChannels, который создаёт карточки проектов и приглашений. Каждая карточка содержит название проекта, роль пользователя и ссылку на канал. Редактирование профиля позволяет изменить имя пользователя и пароль. Для изменения имени сначала отправляется GET-запрос на проверку уникальности к эндпоинту /api/users/username/{encodeURI Component (newUsername)} (статус 404 означает, что имя свободно). Затем отправляется PUT-запрос на обновление к эндпоинту /api/users/ {currentUserId}/username с телом { newUsername: newUsername }. После успешного ответа обновлённые данные сохраняются в localStorage. Для изменения пароля требуется ввести текущий пароль для верификации (при неверном пароле сервер возвращает 401). Все изменения сохраняются только после успешной валидации.

Реализация страницы списка видео. При загрузке страницы данные о проекте не запрашиваются с сервера каждый раз. Сначала проверяется нали-

чие данных в localStorage по ключу cacheKey, который формируется как project_this.currentProjectId. Если данные были сохранены ранее, они извлекаются с помощью getItem(cacheKey) и преобразуются из JSON-строки в объект через JSON.parse(). Затем вычисляется возраст кэша: из текущего времени Date.now() вычитается метка времени сохранения data.timestamp. Если возраст меньше одной минуты, данные считаются актуальными и загружаются из кэша. При загрузке из кэша из объекта data извлекаются projectName и role, которые присваиваются свойствам this.currentProject.name и this.userRole. Затем вызывается метод updateHeader() для обновления заголовка страницы, после чего асинхронно загружается список видео через this.loadVideos(). Дополнительно запускается метод updateDataInBackground(), который в фоновом режиме проверяет актуальность данных на сервере и при необходимости обновляет кэш и интерфейс.

После загрузки страницы из кэша запускается updateDataInBackground, который асинхронно запрашивает свежие данные с сервера. Внутри этого метода с помощью Promise.all параллельно выполняются запросы к эндпоинту /api/project/{currentProjectId} для получения актуальных данных проекта и к эндпоинту /api/connection/getrole/{currentProjectId}/{currentUserId} для получения роли пользователя. После получения ответов извлечённые данные преобразуются в объекты newProject и newRole с помощью методов projectRes.json() и roleRes.json(). Затем обновляется кэш в localStorage: по ключу cacheKey сохраняется объект, содержащий поля project, role и timestamp (текущее время). Далее выполняется проверка: если название проекта, полученное с сервера (newProject.name), отличается от текущего названия (this.currentProject.name), то свойства this.currentProject и this.currentUser обновляются, и вызывается метод updateHeader() для обновления интерфейса без перезагрузки страницы.

Список видео формируется на основе данных, полученных с сервера и переданных в метод displayVideos(videos) через параметр videos. Для каждого видео вызывается метод this.createVideoCard(video), который создаёт HTML-элемент карточки с названием видео (video.title), датой создания (video.createdAt), дедлайном (video.deadline) и прогресс-баром (video.completionPercentage). Созданная карточка добавляется в контейнер с помощью container.appendChild(card). Если пользователь является вла-

дельцем проекта (значение 'OWNER'), то вызывается метод this.createCreate VideoCard(), который создаёт карточку с символом «+» для создания нового видео. Каждая карточка видео получает CSS-класс в зависимости от текущего статуса видео, который определяется значением поля video.status. Процент завершения подставляется в атрибут style="width: ${video.completion Percentage}%" блока с классом progress-bar и в текст элемента progress-text.

Реализация первого этапа создания видео. При загрузке страницы в методе init() первым делом выполняются операции очистки хранилища. Удаляются два ключа из localStorage: currentVideoId и pendingVideoData. Ключ currentVideoId хранит идентификатор текущего редактируемого видео, а pendingVideoData — временные данные создаваемого видео (название, описание, дедлайн и идентификатор проекта). Если их не удалить, после прерванного создания в хранилище останется мусор. После очистки вызывается this.loadProjectData() для загрузки информации о текущем проекте, а затем this.initEventListeners() для настройки обработчиков событий.

При нажатии кнопки «Далее» вызывается метод saveVideoDataAnd Proceed(), в котором выполняется валидация полей. Извлекаются значения: title из поля videoTitle, description из videoDescription, deadline из videoDeadline. С помощью trim() удаляются лишние пробелы. Проверяется, что title и deadline не пустые. Если обязательное поле не заполнено, вызывается this.showNotification() с сообщением «Заполните обязательные поля». После успешной валидации формируется объект videoData, содержащий поля title, description, deadline и projectId. Объект преобразуется в JSON-строку через JSON.stringify() и сохраняется в localStorage по ключу pendingVideoData с помощью setItem(). После сохранения выполняется перенаправление на второй этап — window.location.href = '../stage2/index.html'. Данные между страницами передаются без отправки на сервер.

Второй этап отвечает за настройку производственных стадий. При загрузке пользователю предлагается выбрать метод: копирование из существующего видео или ручное создание. Если method === 'copy', у блока copyOption удаляется класс hidden через className.remove('hidden'), подгружается список видео проекта (метод loadExistingVideos()), и этапы выбранного видео повторяются. Если method === 'create', отображается блок createOption и вызывается this.initializeCreateOption() для ручного добавления стадий.

При ручном создании стадии сохраняются в локальном массиве this.createdStages. Метод addStageLocally() формирует объект localStage со свойствами: localId (временный идентификатор), title, description, requiredRole, dependsOnLocalId (идентификатор стадии, от которой зависит данная). Использование локальных идентификаторов позволяет выстраивать зависимости между ещё не созданными на сервере стадиями. Объект добавляется в массив через push(), затем вызываются this.displayLocalStage(localStage) для отображения карточки и this.loadStagesForDependency() для обновления выпадающего списка зависимостей.

Создание стадий на сервере происходит в два этапа в методе createStages FromLocalArray(videoId). На первом этапе создаётся пустая карта oldLocalIdToNewServerIdMap = new Map(). Выполняется проход по this.createdStages: для каждой стадии отправляется POST-запрос к /api/stage с полями videoId, title, description, requiredRole, dependsOnStageId. Из ответа извлекается serverStage.id — реальный идентификатор, присвоенный сервером. Сохраняется соответствие: oldLocalIdToNewServerIdMap.set(localStage.localId, serverStage.id). На втором проходе для каждой стадии, у которой есть dependsOnLocalId, по карте получаются newStageId и newDependsOnId, затем отправляется PUT-запрос на /api/stage/{newStageId} с телом { dependsOnStageId: newDependsOnId }. Это позволяет корректно установить зависимости, даже если зависимая стадия была создана позже в локальном массиве.

Реализация канбан-доски и файлового менеджера. Канбан-доска является главным элементом сервиса, объединяющим визуальный конвейер и файловый менеджер. Логика определения доступности стадий реализована в методе checkStageAvailability(). Сначала с помощью this.stages.forEach(stage => stage.available = false) сбрасываются флаги доступности. Стадии без зависимости (!stage.dependsOnStageId) помечаются доступными сразу: stage.available = true. Затем в цикле for (const stage of this.stages) для каждой стадии с зависимостью отправляется асинхронный GET-запрос к /api/stage/${stage.dependsOnStageId}. Ответ преобразуется в dependent StageData, и stage.available устанавливается равным dependent StageData.completed. Таким образом, стадия становится доступной только после завершения зависимой.

Для загрузки файлов используется gRPC-Web. Метод uploadSingleFile WithProgress(file, stageId, index) разбивает файл на чанки по 64 КБ. Сначала загружается protobuf-схема из file_transfer.proto, файл преобразуется в бинарный массив через file.arrayBuffer(). Создаётся массив chunks, первым элементом которого становится закодированное сообщение с метаданными (имя, размер, тип, stageId) из frameMessage(metadataBuffer). В цикле for (let i = 0; i < view.length; i += chunkSize) очередной чанк извлекается через view.slice(i, i + chunkSize), кодируется в protobuf через UploadFileRequest.encode(chunk).finish(), обрамляется через frameMessage() и добавляется в chunks. Все чанки объединяются в newBlob (chunks) и отправляются через xhr.send().

Для отслеживания прогресса используется событие xhr.upload .onprogress. В обработчике проверяется event.lengthComputable, процент вычисляется как Math.round((event.loaded / event.total) * 100) и передаётся в this.updateFileProgress(index, percent, 'uploading'), который обновляет индикатор прогресса в интерфейсе. Каждый файл отображает свой прогресс независимо.

Каждая карточка стадии создаётся методом createStageCard(stage). Внутри проверяются stage.completed (завершена ли), stage.available (доступна ли), роль пользователя (this.isOwner или сравнение this.currentUser Role === stage.requiredRole). В зависимости от условий карточке присваиваются CSS-классы, отображаются или скрываются кнопки редактирования, удаления и завершения. При нажатии на «Завершить стадию» вызывается showComplete ConfirmationOnCard(card, stageId): исходная карточка заменяется на блок с сообщением «Завершить стадию?» и кнопками confirm-btn и cancel-btn.

На странице используется несколько модальных окон: stageInfoModal (просмотр стадии с файлами), uploadFilesModal (загрузка файлов), createStageModal, editStageModal, confirmDeleteModal. Закрытие окон реализовано двумя способами: по клику на фон (проверка e.target === modal) и по нажатию Escape. Механизм реализован в методе initModalCloseOnOutside Click().

Реализация раздела команды. Страница команды отображает всех участников проекта, их роли и количество непрочитанных сообщений в чате. При за-

грузке название проекта сначала показывается из localStorage для мгновенного отклика, затем в фоне загружаются актуальные данные через loadProjectData() с аналогичной системой кэширования (проверка localStorage.getItem(cacheKey) и возраста кэша).

Для отображения количества непрочитанных сообщений используется метод getUnreadCount(userId). Он отправляет GET-запрос к /api/chat/messages ?userId=&user2Id=, фильтрует сообщения с senderId === userId && !isRead и возвращает длину полученного массива. Для периодического обновления значков startPolling() с помощью setInterval(3000) запускает вызов this.updateUnreadCounts(), который проходит по всем карточкам участников, для каждой вызывает getUnreadCount и обновляет отображение значка. Это позволяет видеть новые сообщения без перезагрузки страницы.

Реализация чата. Страница чата использует gRPC-Web с protobuf для передачи данных и polling для получения новых сообщений. Отображение сообщений реализовано в методе displayMessages(), который формирует визуальное представление диалога. В начале объявляются currentDate = null (для разделителей дат) и firstUnreadShown = false (флаг добавления разделителя непрочитанных). Перебор this.messages через forEach(): из message.sentAt создаётся Date и извлекается дата через toLocaleDateString(). Если current Date !== messageDate, добавляется разделитель даты через this.createDate Separator(messageDate). Если !firstUnreadShown и message.id === this. firstUnread MessageId, добавляется this.createUnreadSeparator() и флаг устанавливается в true. Затем сообщение добавляется через container.append Child(this.createMessageElement(message)).

Метод createMessageElement(message) определяет isOwnMessage = message.senderId === this.currentUserId. В message-footer отображаются: пометка «ред.» при message.edited, время отправки, индикатор прочтения (одна галочка для доставленных, две — для прочитанных) только для собственных сообщений.

Для мобильных устройств реализована специальная обработка. setupKeyboard Handling() добавляет обработчик focus на поле ввода, устанавливает this. keyboardVisible = true и через setTimeout(() => this. scrollToBottom(true), 300) прокручивает чат вниз. Обработчик resize проверяет видимость клавиатуры и при уменьшении высоты окна снова прокручи-

вает чат. setupViewport Handling() вычисляет 1% от высоты окна и устанавливает CSS-переменную -vh для корректного отображения элементов с высотой 100vh. Поле ввода — textarea, автоматически увеличивающая высоту до 120 пикселей: метод autoResizeTextarea() сбрасывает высоту через style.height = 'auto', затем вычисляет Math.min(textarea.scrollHeight, 120) и устанавливает новую высоту.

Чат поддерживает редактирование и удаление отправленных сообщений (только для своих). При нажатии на карандаш вызывается startEditing(message): сохраняется this.editingMessageId, поле ввода заполняется текстом сообщения, появляется индикатор «Редактирование...». Кнопка отправки меняет поведение на обновление существующего сообщения через grpcCall('edit-message', 'EditMessageGrpcRequest', 'MessageGrpcResponse', {messageId, content }). При нажатии на крестик вызывается showDeleteConfirmation(messageId): создаётся модальное окно с вопросом «Удалить сообщение?» и предупреждением, что действие нельзя отменить.

Реализация аналитики. Страница аналитики отображает статистику по подключённому каналу (YouTube или VK). loadAnalyticsData() проверяет наличие URL канала в проекте (if (!this.currentProject?.url)), затем формирует запрос к единому эндпоинту /api/analytics?url=${encodeURIComponent (this.currentProject.url)}. Бэкенд анализирует ссылку, определяет платформу и возвращает унифицированный ответ. При успешном ответе (response.ok) данные извлекаются через response.json(), вызываются this.displayChannel Stats(data) для отображения общей статистики и this.displayVideos(data. videos || []) для списка видео.

displayChannelStats обновляет элементы subscribersValue, viewsValue, videosCountValue, форматируя числа через this.formatNumber().formatNumber (num) преобразует числа: если num >= 1000000 — возвращает (num / 1000000). toFixed(1) + 'M', если num >= 1000 — (num / 1000).toFixed(1) + 'K', иначе — num.toString(). displayVideos для каждого видео создаёт карточку с превью (ленивая загрузка через loading="lazy"), названием, просмотрами, лайками и датой публикации. При клике на карточку открывается страница видео на платформе в новой вкладке через window.open(videoUrl, '_blank').

Реализация настроек. Страница настроек доступна только владельцу проекта. Форма редактирования проекта: метод checkFormChanges(), вызы-

ваемый при каждом событии input, сравнивает текущие значения полей с this.originalData.name, description, url. Вычисляется hasChanges, кнопка сохранения получает saveBtn.disabled = !hasChanges, и её класс меняется с btn-secondary на btn-primary при наличии изменений.

В разделе управления участниками при нажатии на кнопку удаления карточка участника заменяется на блок подтверждения (аналогично канбан-доске). Приглашение новых участников: сначала отправляется GET-запрос к /api/users/username/{encodeURIComponent(username)} для получения ID пользователя (статус 404 — пользователь не найден). Затем отправляется POST-запрос на создание связи между проектом и пользователем с выбранной ролью. Доступные для приглашения роли — редактор, оператор, дизайнер (роль OWNER исключена).

Сервис аналитики и интеграция с внешними API. Для сбора статистики разработан отдельный сервис на Python с использованием библиотек requests и aiohttp. Взаимодействие с основным бэкендом организовано по протоколу gRPC (бинарный протокол, строгая типизация через .proto-файлы). Пример запроса к VK API: формирование параметров с access_token и v, отправка GET, обработка ошибок через проверку наличия поля 'error' в ответе. Токены доступа получены из аккаунта разработчика. Для оптимизации использования квот API реализовано кэширование результатов на стороне основного бэкенда.

Развертывание. Развертывание автоматизировано с помощью CI/CD-пайплайна в GitLab. Файл .gitlab-ci.yml описывает этапы: проверка качества кода через ESLint (при ошибках пайплайн останавливается), сборка проекта (очистка CSS и JS), деплой на сервер с помощью rsync (копируются только изменённые файлы, используется SSH-ключ, хранящийся в защищённых переменных GitLab). После копирования выполняется перезагрузка Nginx.

Для изоляции приложения используется Docker: docker-compose.yml описывает контейнер на основе образа nginx:alpine (лёгкая версия около 20 МБ). Статические файлы подключаются в контейнер, пробрасываются порты 80 и 443, настроен restart: always. Конфигурация Nginx: nginx.conf отвечает за отдачу статики, сжатие данных, проксирование запросов /api/ на бэкенд. nginx-ssl.conf настраивает HTTPS с сертификатами, включает современные версии TLS и перенаправление HTTP на HTTPS. Конфигурация бэкенда задаётся в файле config.js через глобальную переменную SERVER.