Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет

информатики и радиоэлектроники»

|  |  |
| --- | --- |
|  | Номер зачетной книжки 15350134 |
|  | Производственная практика зачтена с оценкой  \_\_\_\_\_ (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)  (цифрой) (прописью) |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись руководителя практики от БГУИР)  \_\_\_\_.\_\_\_\_\_.2025 |

**ОТЧЕТ**

**по преддипломной практике**

Место прохождения практики: ООО «Фабрика инноваций и решений»

Сроки прохождения практики: с 10.02.2025 по 23.03.2025

|  |  |  |
| --- | --- | --- |
| Руководитель практики от предприятия:  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Д.С.Кисляк  (подпись руководителя)  М.П. |  | Студент группы 153502  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Д.В.Толстой  (подпись студента)  Руководитель практики от БГУИР  Давыдчик А.В. |

Минск 2025

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc197358140)

[1 Анализ литературных исследований и программных решений 4](#_Toc197358141)

[1.1 Описание и анализ предметной области 4](#_Toc197358142)

[1.2 Сравнительный обзор аналогов программного средства 5](#_Toc197358143)

[2 Моделирование предметной области 9](#_Toc197358144)

[2.1 Описание функциональности программного средства 9](#_Toc197358145)

[2.2 Разработка функциональных требований к программному средству 10](#_Toc197358146)

[3 Технико-экономическое обоснование разработки и реализации на рынке платформы для анализа статистики игрока в компьютерной игре DOTA2 11](#_Toc197358147)

[3.1 Характеристика программного средства, разрабатываемого для реализации на рынке 11](#_Toc197358148)

[3.2 Расчёт инвестиций в разработку программного средства 12](#_Toc197358149)

[3.3 Расчёт экономического эффекта от реализации программного средства на рынке 15](#_Toc197358150)

[3.4 Расчёт показателей экономической эффективности разработки и реализации программного средства на рынке 16](#_Toc197358151)

[3.5 Вывод об экономической целесообразности реализации проектного решения 17](#_Toc197358152)

[4 Frontend 18](#_Toc197358153)

[4.1 Javascript 18](#_Toc197358154)

[4.2 Typescript 29](#_Toc197358155)

[4.3 Vite 31](#_Toc197358156)

[4.4 React 35](#_Toc197358157)

[4.5 Redux и Redux Saga 40](#_Toc197358158)

# **ВВЕДЕНИЕ**

Киберспорт пережил колоссальный рост за последние три десятилетия, превратившись в многомиллиардную индустрию с глобальной потребительской базой. В 2021 году доходы отрасли достигли 1,22 миллиарда долларов, и прогнозируется ее дальнейшее значительное расширение. Профессиональные киберспортивные соревнования привлекают многочисленную международную аудиторию и отличаются значительными финансовыми призами. Киберспорт перестал быть нишевым увлечением и прочно утвердился в мейнстриме развлечений, напрямую конкурируя с традиционными видами спорта за внимание зрителей и финансовые потоки. Ожидается, что к 2025 году глобальная аудитория киберспорта превысит 640 миллионов человек. Развитие киберспорта, подкрепленное впечатляющими показателями зрительской аудитории и финансовой прибылью, делает его важной областью для академических исследований и технологических инноваций. Масштаб индустрии указывает на существенный потенциальный рынок для платформы, специализирующейся на анализе данных [1].

Dota 2 выделяется как одна из наиболее сложных и стратегически глубоких дисциплин в киберспорте. Геймплей этой игры требует от участников глубокого понимания командного взаимодействия, тонкостей противостояния героев и высокого уровня стратегического мышления. Для достижения успеха в Dota 2 игрокам необходимо демонстрировать слаженную командную работу, обладать всесторонним знанием карты и принимать взвешенные стратегические решения в режиме реального времени. Сложная механика игры и многообразие способностей героев обуславливают ее исключительную стратегическую глубину. Dota 2 имеет богатую историю на профессиональной киберспортивной сцене и пользуется поддержкой преданного глобального сообщества. Эта многолетняя история и вовлеченность сообщества подтверждают ее значимость как ключевой киберспортивной дисциплины. Внутренняя сложность и стратегическая насыщенность Dota 2 создают значительный спрос на аналитические инструменты и ресурсы, способные помочь игрокам и командам разобраться в тонкостях игры и оптимизировать свою эффективность. Устоявшееся и активное сообщество представляет собой обширную целевую аудиторию для подобной платформы [2].

В ответ на растущее значение анализа данных в киберспортивной сфере, и, в частности, в сложной экосистеме Dota 2, данный дипломный проект направлен на разработку специализированной платформы. Эта платформа будет посвящена всестороннему анализу статистики игроков в Dota 2. Используя возможности доступных источников данных и применяя передовые современные технологии, проект стремится предоставить пользователям практические знания, которые могут быть непосредственно использованы для улучшения их игрового процесса и способствовать более глубокому пониманию широкого конкурентного ландшафта Dota 2.

# **1 АНАЛИЗ ЛИТЕРАТУРНЫХ ИССЛЕДОВАНИЙ И ПРОГРАММНЫХ РЕШЕНИЙ**

## **1.1 Описание и анализ предметной области**

Компьютерные игры прошли долгий путь от простого развлечения для избранных до доминирующей силы в современной культуре. Сегодня индустрия видеоигр представляет собой многомиллиардный рынок, который по своим масштабам успешно конкурирует с такими гигантами индустрии развлечений, как кино и музыка. При этом аудитория геймеров значительно расширилась, охватывая людей всех возрастов и профессий, а не только подростков.

Одним из первых и наиболее известных примеров компьютерной игры стала Spacewar, разработанная еще в 1962 году студентами Массачусетского технологического института (MIT). Эта игра, в которой два космических корабля сражались друг с другом в условиях гравитации звезды, продемонстрировала интерактивный потенциал вычислительной техники и заложила основу для будущего развития видеоигр. В последующие десятилетия компьютерные игры завоевали широкое признание, непрерывно развиваясь как в плане графики, так и в технологическом отношении [3].

Значительным шагом в эволюции многопользовательских игр стало появление возможности сетевой игры через локальную вычислительную сеть, впервые реализованной в игре Doom 2. Это нововведение коренным образом изменило представление об игровом процессе, позволив игрокам соревноваться не только с искусственным интеллектом, но и с реальными соперниками в режиме реального времени. Таким образом, в компьютерных играх зародился элемент спортивной состязательности, который впоследствии привел к формированию киберспорта.

Объем мирового рынка видеоигр в 2024 году достиг 184,3 миллиарда долларов США, показав незначительный рост в 0,2% по сравнению с 2023 годом. Несмотря на то, что этот показатель немного уступает рекордному пандемийному 2021 году, игровая индустрия продолжает опережать другие сегменты рынка развлечений, такие как кинопрокат и музыкальный бизнес. Прогнозируется дальнейший рост рынка, который, по некоторым оценкам, достигнет 213,3 миллиарда долларов к 2027 году [4].

Современным эталоном киберспортивной дисциплины по праву считается Dota 2 – многопользовательская онлайн-баталия (MOBA), разработанная компанией Valve Corporation и выпущенная в 2013 году. Этот жанр характеризуется глубокой стратегической составляющей, необходимостью быстро принимать решения и слаженным командным взаимодействием. С момента своего релиза Dota 2 превратилась в настоящий культурный феномен, проникнув в различные формы медиа и став одной из самых влиятельных игр в своем жанре.

С течением времени популярность Dota 2 неуклонно росла. Десятки миллионов игроков ежемесячно заходят в игру, чтобы соревноваться, находить новых друзей и получать удовольствие. Киберспортивная сцена Dota 2 особенно развита, с многочисленными турнирами и лигами, привлекающими лучших игроков со всего мира. Крупнейшим ежегодным событием является The International, который отличается рекордными призовыми фондами и привлекает миллионы зрителей.

Несмотря на конкуренцию со стороны таких проектов, как League of Legends и других MOBA-игр, Dota 2 остается одной из самых популярных и влиятельных игр в своем жанре. По данным SteamCharts на апрель 2025 года, ежемесячная аудитория игры составляет около 393 тысяч активных игроков , а пиковое количество пользователей, одновременно находящихся в игре, достигало 630 тысяч по данным за тот же период. Эти цифры свидетельствуют о сохраняющейся популярности и значимости Dota 2 в мировом игровом сообществе [5].

## **1.2 Сравнительный обзор аналогов программного средства**

Разрабатываемое решение является платформой для анализа статистики игрока в компьютерной игре Dota 2, и на рынке представлено небольшое количество подобных решений: Stratz, Dotabuff, OpenDota.

Суть сравнительного анализа аналогов разрабатываемого программного средства заключается в выделении их достоинств и недостатков для последующего определения функциональных возможностей, которые необходимо реализовать в разрабатываемой платформе обучения и аттестации. Сравнительный анализ позволяет учесть опыт существующих решений, выявить их сильные стороны и определить направления для инноваций.

Stratz – аналитическая платформа для Dota 2, предоставляющая статистику матчей и игроков. Пример интерфейса Stratz приведен на рисунке 1.1.

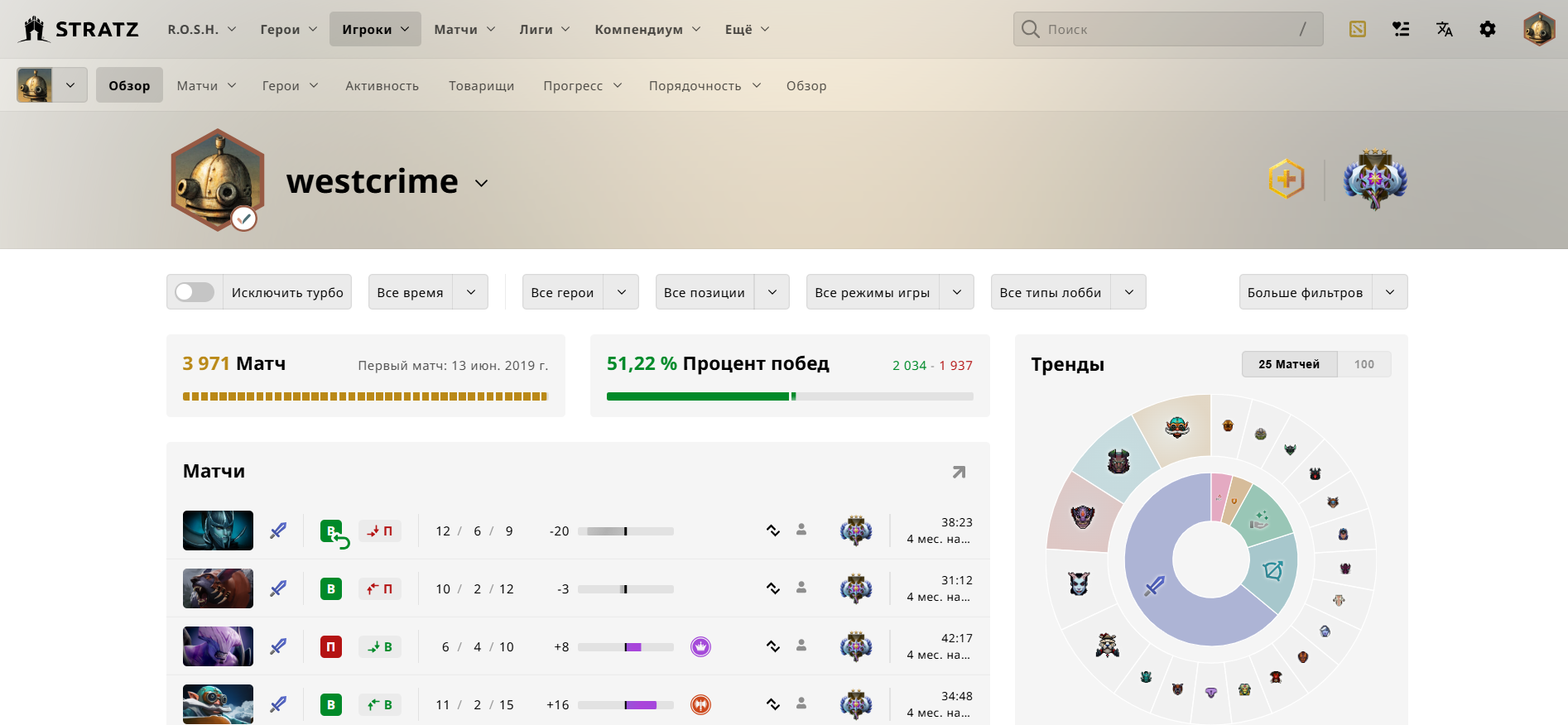


Рисунок 1.1 – Интерфейс платформы Stratz

Stratz позволяет анализировать:

– статистику матчей (KDA, GPM, XPM, win rate);

– историю игр и прогресс игрока;

– эффективность героев и их сборки;

– командные показатели и стратегии.

Преимущества Stratz:

– детальная аналитика матчей;

– удобные графики и таблицы;

– API для разработчиков.

Недостатки Stratz:

– отсутствие персонализированных рекомендаций на основе ИИ;

– ограниченный анализ слабых мест игрока;

– сложный интерфейс для новичков.

Dotabuff – сервис статистики для Dota 2 с базовой аналитикой. Пример интерфейса Dotabuff приведен на рисунке 1.2.

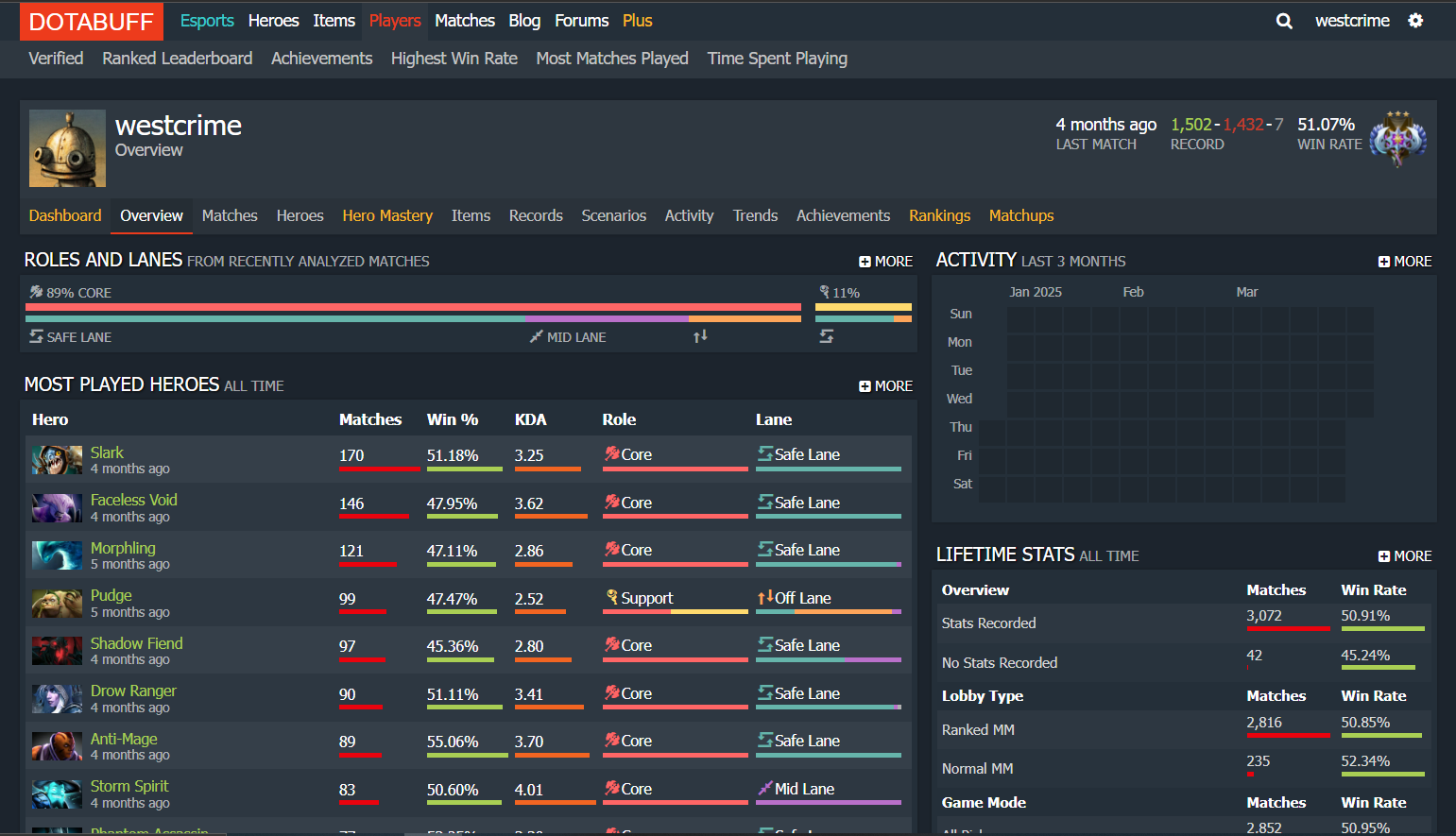


Рисунок 1.2 – Интерфейс платформы Dotabuff

Dotabuff предоставляет:

– общую статистику игрока;

– рейтинги героев и предметов;

– историю матчей.

Преимущества Dotabuff:

– простота использования;

– доступ к базовым метрикам.

Недостатки Dotabuff:

– отсутствие углубленного анализа;

– нет рекомендаций по улучшению игры.

OpenDota – открытая платформа для анализа матчей Dota 2 с API. Пример интерфейса Dotabuff приведен на рисунке 1.3.



Рисунок 1.3 – Интерфейс платформы OpenDota

OpenDota включает:

– разбор матчей по таймлайну;

– статистику по героям и игрокам;

– инструменты для разработчиков.

Преимущества OpenDota:

– открытый доступ к данным;

– возможность кастомизации отчетов.

Недостатки OpenDota:

– сложный интерфейс;

– нет автоматизированных советов для игроков.

Проведенный анализ современных платформ для анализа игровой статистики в Dota 2 выявил ряд характерных тенденций и проблем в данной предметной области. Большинство существующих решений предлагают базовый набор аналитических функций, однако их реализация часто страдает либо избыточной сложностью интерфейса, либо ограниченностью возможностей для глубокого анализа игровых данных.

Облачные платформы, такие как Stratz и Dotabuff, обеспечивают удобный доступ к базовой статистике матчей и минимальный порог входа для новых пользователей. Однако их возможности по персонализированному анализу слабых мест игрока и генерации рекомендаций остаются крайне ограниченными. Открытые системы типа OpenDota предлагают более гибкий доступ к данным через API, но требуют значительных технических знаний для полноценного использования и не предоставляют готовых аналитических решений для обычных игроков.

На основе анализа аналогов программного средства в дальнейшем были выделены основные функции, которые должны быть реализованы.

# **2 МОДЕЛИРОВАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ**

## **2.1 Описание функциональности программного средства**

Для описания функциональности программного средства была выбрана UML диаграмма вариантов использования, которая отражает отношения между актерами и прецедентами и позволяет описать систему на концептуальном уровне. Основные элементы диаграммы – актер и прецедент.

UML предназначен для определения, визуализации, проектирования и документирования программных систем.

На основе технического была составлена диаграмма вариантов использования, изображённая на рисунке 2.1.

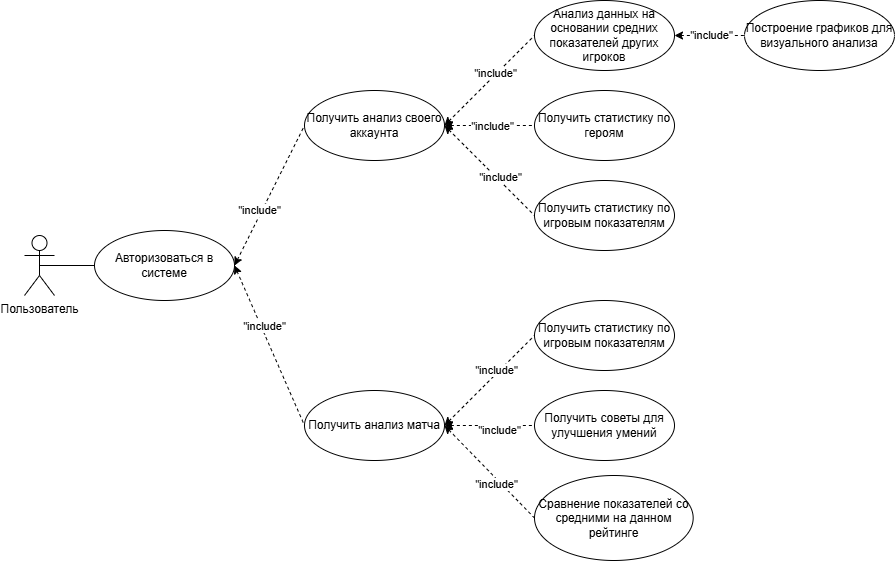


Рисунок 2.1 – Диаграмма вариантов использования

В соответствии с рисунком 2.1 можно выделить два основных составляющих элемента: актер и прецедент.

Актер – роль внешнего объекта программного средства, прямо взаимодействующая с ее частями. Участником может быть человек, другая система, подсистема или класс.

Прецедент – описание множества последовательных событий, выполняемых программным средством, а также отображение взаимодействия между участниками и системой.

Пользователь в системе анализа статистики Dota 2 получает доступ к комплексному инструментарию для работы с игровыми данными. После авторизации через Steam-аккаунт и привязки профиля Dota 2, система предоставляет возможность глубокого анализа игровой деятельности.

Основной функционал включает просмотр детализированной статистики аккаунта с отображением уровня, текущего MMR и общего количества сыгранных матчей, а также графиков динамики изменения рейтинга за выбранные периоды времени. Пользователи могут изучать персонализированную статистику по героям, включая показатели винрейта, KDA и эффективности для каждого персонажа, с выделением наиболее успешных героев для текущего уровня игры.

Система предлагает анализ ключевых игровых показателей (GPM, XPM, нанесенный урон, лечение), позволяющий сравнивать текущие результаты с предыдущими периодами. На основе собранных данных формируются персонализированные рекомендации по улучшению навыков, выделяющие слабые места и предлагающие конкретные способы их устранения.

Дополнительно реализована функция сравнения личных показателей со средними значениями для текущего рейтинга, что помогает пользователям объективно оценивать свой уровень игры и определять приоритетные направления для совершенствования. Все данные могут быть обновлены по запросу для получения актуальной информации с серверов Steam.

## **2.2 Разработка функциональных требований к программному средству**

Рассмотрим функциональные требования для данной платформы:

– авторизация в системе через аккаунт Steam;

– просмотр основной информации профиля (никнейм, id и т. д.);

– просмотр рекордов пользователя (общее количество убийств, помощи и т. д.);

– просмотр основной статистики (общий процент побед, ранг и т. д.);

– просмотр истории матчей;

– подробный анализ матча с игровыми показателями;

– получение советов, сгенерированные на основе ИИ;

# **3 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ И РЕАЛИЗАЦИИ НА РЫНКЕ ПЛАТФОРМЫ ДЛЯ АНАЛИЗА СТАТИСТИКИ ИГРОКА В КОМПЬЮТЕРНОЙ ИГРЕ DOTA2**

## **3.1 Характеристика программного средства, разрабатываемого для реализации на рынке**

Целью разработки является создание специализированной платформы для анализа статистики игроков в DOTA 2, которая предоставляет пользователям удобный инструмент для глубокого анализа их игровых данных. Платформа использует открытые данные из публичных API связанных с данной игрой, визуализирует их в понятном виде (графики, схемы, таблицы) и предлагает персонализированные рекомендации для улучшения навыков игрока на основе анализа матчей с использованием искусственного интеллекта.

Приложение будет иметь интуитивно понятный интерфейс, позволяющий пользователям быстро ориентироваться и находить нужные функции. Оно предоставит возможность авторизации через игровую платформу Steam. Просмотр различных аспектов игрока, выбор промежутка времени анализа, вариативное представление данных. Возможность детального анализа отдельных матчей (таймлайн событий, ключевые моменты, эффективность действий). Сравнение статистики игрока с усредненными показателями других игроков аналогичного уровня. Генерация персонализированных советов для улучшения навыков (например, улучшение фарма, выбор героев, стратегии для конкретных ролей).

Ограниченный доступ к данной платформе можно получить бесплатно для анализа 3 матчей в целях завлечения пользователей, далее доступ будет предоставляться по ежемесячной платной подписке.

Игровая индустрия является одной из самых динамично развивающихся отраслей. Этот рынок показывает устойчивый и активный рост, и на нем уже присутствует несколько аналогов описанной выше платформы.

1 Stratz API. Stratz предоставляет доступ к данным матчей и статистике игроков, но не предлагает персонализированных рекомендаций на основе ИИ. Stratz ориентирован на разработчиков и продвинутых пользователей, что делает его менее доступным для новичков.

2 Dotabuff. Dotabuff предоставляет базовую статистику и анализ матчей, но не предлагает персонализированных рекомендаций. Отсутствует интеграция ИИ для анализа слабых мест игрока. Интерфейс Dotabuff ориентирован на общую статистику, а не на индивидуальный анализ.

3 OpenDota. OpenDota предоставляет доступ к данным матчей и API для разработчиков, но не предлагает удобного интерфейса для анализа и рекомендаций. Отсутствует функционал для генерации советов на основе ИИ.

Разрабатываемая платформа предлагает уникальное сочетание глубокого анализа данных, удобного интерфейса и персонализированных рекомендаций на основе ИИ, что выделяет ее среди существующих аналогов. Она ориентирована на широкий круг пользователей, от новичков до профессионалов, и предоставляет инструменты для улучшения навыков и понимания игры. На рынке подобных решений с интеграцией ИИ практически нет, что открывает возможности для занятия уникальной ниши.

## **3.2 Расчёт инвестиций в разработку программного средства**

Расчёт затрат на основную заработную плату разработчиков производится исходя из количества людей, которые занимаются разработкой программного продукта, месячной зарплаты каждого участника процесса разработки и сложности выполняемой ими работы. Затраты на основную заработную плату рассчитаны по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (3.1) |

где коэффициент премий и иных стимулирующих выплат;

категории исполнителей, занятых разработкой программного средства;

часовая заработная плата исполнителя *i*-й категории, р;

трудоемкость работ, выполняемых исполнителем *i*-йкатегории, ч.

Разработкой серверной части приложения занимается backend-разработчик. Написание графического интерфейса приложения лежит на frontend-разработчике. Задачами backend-разработчика являются создание серверной части с интеграцией искусственного интеллекта. Frontend-разработчик возьмет на себя разработку графического интерфейса, коммуникации между сервером и веб приложением, обработку пользовательского ввода.

Месячная заработная плата основана на медианных показателях для Junior backend-разработчика за 2024 год по Республике Беларусь, которая составляет примерно 600 Долларов США в месяц, а для Junior frontend-разработчика около 550 Долларов США. По состоянию на 5 марта 2025 года, 1 Доллар США по курсу Национального Банка Республики Беларусь составляет 3,2223 Белорусских рублей.

В перерасчёте на Белорусские рубли месячные оклады для backend-разработчика и frontend-разработчика соответственно составляют составляет 1 772,22 и 1 933,38 Белорусских рублей соответственно.

Часовой оклад исполнителей высчитывается путём деления их месячного оклада на количество рабочих часов в месяце, то есть 168 часов.

Коэффициент премии приравнивается к единице, так как она входит сумму заработной платы. Затраты на основную заработную плату приведены в таблице:

Таблица 1.1 – Затраты на основную заработную плату

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Категория исполнителя | Месячный оклад, р | Часовой оклад, р | Трудоёмкость работ, ч | Итого, р |
| Frontend-разработчик | 1 772,22 | 10,55 | 168 | 1 772,22 |
| Backend-разработчик | 1 933,38 | 11,51 | 168 | 1 933,38 |
| Итого | | | | 3 705,6 |
| Премия и иные стимулирующие выплаты (0%) | | | | 0 |
| Всего затраты на основную заработную плату разработчиков | | | | 3 705,6 |

Расчёт затрат на дополнительную заработную плату команды разработчиков рассчитывается по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (3.2) |

где норматив дополнительной заработной платы.

Значение норматива дополнительной заработной платы принимаем равным 10 %.

Размер отчислений на социальные нужды определяется согласно ставке отчислений принимаем равным 34%. Расчёт отчислений на социальные нужды вычисляется по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (3.3) |

где норматив отчислений в ФСЗН.

Расчёт затрат на прочие расходы определяется при помощи норматива прочих расчётов. Эта величина имеет значение 30%. Расчёт прочих расходов вычисляется по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (3.4) |

где норматив прочих расходов.

Для того, чтобы рассчитать расходы на реализацию, необходимо знать норматив расходов на неё. Принимаем значение норматива равным 3%. Формула, которая использована для расчёта расходов на реализацию:

|  |  |  |
| --- | --- | --- |
|  |  | (3.5) |

где Нр – норматив расходов на реализацию.

Определяем общую сумму затрат на разработку и реализацию как сумму ранее вычисленных расходов: на основную заработную плату разработчиков, дополнительную заработную плату разработчиков, отчислений на социальные нужды, расходы на реализацию и прочие расходы. Значение определяется по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (3.6) |

Таким образом, величина затрат на разработку программного средства высчитывается по указанной выше формуле и указана в таблице:

Таблица 1.2 – Затраты на разработку

|  |  |  |
| --- | --- | --- |
| Название статьи затрат | Формула/таблица для расчёта | Значение, р. |
| 1. Основная заработная плата разработчиков | См. таблицу 1.1 | 3 705,6 |
| 2. Дополнительная заработная плата разработчиков |  | 370,05 |
| 3. Отчисление на социальные нужды |  | 1 385,72 |
| 4. Прочие расходы |  | 1 111,68 |
| 5. Расходы на реализацию |  | 111,17 |
| 6. Общая сумма затрат на разработку и реализацию |  | 6 684,22 |

## **3.3 Расчёт экономического эффекта от реализации программного средства на рынке**

Для расчёта экономического эффекта организации-разработчика программного средства, а именно чистой прибыли, необходимо знать такие параметры как объем продаж, цену реализации и затраты на разработку.

Для оценки экономического эффекта программного продукта, предположим, что около 10 000 человек будут заинтересованы данным приложением. Из них 6 000 человек будут использовать его, а 2 500 приобретут расширенную версию. Цена на расширенную версию приложения составляет 4,99 долларов США. Таким образом, отпускная цена копии программного средства составляет 16,08 Белорусских рубля.

Для расчёта прироста чистой прибыли необходимо учесть налог на добавленную стоимость, который высчитывается по следующей формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (3.7) |

где N – количество копий(лицензий) программного продукта, реализуемое за год, шт.;

– отпускная цена копии программного средства, р.;

– количество приобретённых лицензий;

– ставка налога на добавленную стоимость, %.

Ставку налога на добавленную стоимость по состоянию принимаем равной 20%. Используя данное значение, посчитаем НДС:

|  |  |  |
| --- | --- | --- |
|  |  |  |

Посчитав налог на добавленную стоимость, можно рассчитать прирост чистой прибыли, которую получит разработчик от продажи программного продукта. Для этого используется формула:

|  |  |  |
| --- | --- | --- |
|  |  | (3.8) |

где *N* – количество копий(лицензий) программного продукта, реализуемое за год, шт.;

Цотп – отпускная цена копии программного средства, р.;

– сумма налога на добавленную стоимость, р.;

Нп – ставка налога на прибыль, %;

– рентабельность продаж копий.

Ставку налога на прибыль принимаем равной 20%. Рентабельность продаж копий взята в размере 40%. Зная ставку налога и рентабельность продаж копий (лицензий), рассчитывается прирост чистой прибыли для разработчика:

|  |  |  |
| --- | --- | --- |
|  |  |  |

## **3.4 Расчёт показателей экономической эффективности разработки и реализации программного средства на рынке**

Для того, чтобы оценить экономическую эффективность разработки и реализации программного средства на рынке, необходимо рассмотреть результат сравнения затрат на разработку данного программного продукта, а также полученный прирост чистой прибыли за год.

Сумма затрат на разработку меньше суммы годового экономического эффекта, поэтому можно сделать вывод, что такие инвестиции окупятся менее, чем за один год.

Таким образом, оценка экономической эффективности инвестиций производится при помощи расчёта рентабельности инвестиций (Return on Investment, ROI). Формула для расчёта ROI:

|  |  |  |
| --- | --- | --- |
|  |  | (3.9) |

где – прирост чистой прибыли, полученной от реализации программного средства на рынке информационных технологий, р.;

Зр – затраты на разработку и реализацию программного средства, р.

|  |  |  |
| --- | --- | --- |
|  |  |  |

## **3.5 Вывод об экономической целесообразности реализации проектного решения**

Проведённые расчёты технико-экономического обоснования позволяют сделать предварительный вывод о целесообразности разработки данного программного продукта. Общая сумма затрат на разработку и реализацию составила Белорусских рублей, а отпускная цена была установлена на уровне 16,08 Белорусских рублей.

Прирост чистой прибыли за год, исходя из предполагаемого объёма продаж в размере 2500 расширенных версий в год, составляет 10 720,00 Белорусских рублей. Рентабельность инвестиций за год составляет .

Это означает, что разработка данного программного продукта является целесообразной и реализация программного средства по установленной цене имеет смысл.

Однако, следует учитывать возможные риски, связанные с конкуренцией со стороны аналогов, что может привести к незамеченности продукта на рынке. Кроме того, высокая рентабельность связана с рисками, и расчётные результаты были получены при предполагаемом объёме продаж в 2500 копий в год.

Тем не менее, при поддержке проект может получить долгосрочное и успешное развитие, и количество проданных копий может превысить предполагаемое количество.

# **4 Frontend**

## **4.1 Javascript**

JavaScript – это интерпретируемый язык программирования, который используют для написания frontend- и backend-частей сайтов, а также мобильных приложений. Часто в текстах и обучающих материалах название языка сокращают до JS. Это язык программирования высокого уровня, то есть код на нем понятный и хорошо читается.

JS поддерживают все популярные браузеры. Во frontend-части сайтов язык используют для создания интерактива (анимаций, всплывающих форм, автозаполнения), так как он связан с HTML и CSS и может ими манипулировать. В backend-части с языком JavaScript работают на платформе Node.js. С ее помощью, например, разрабатывают серверные веб-приложения и подключают библиотеки. В поисковике Google на JavaScript работает строка автозаполнения, а Netflix, Uber, eBay используют его в своем backend. Уже 6 лет JS – самый популярный язык среди разработчиков по версии GitHub [6].

События, в результате которых появился JavaScript, разворачивались в течение шести месяцев, с мая по декабрь 1995 года. Компания Netscape Communications уверенно прокладывала себе путь в области веб-технологий. Её браузер Netscape Communicator успешно отвоевывал позиции у NCSA Mosaic, первого популярного веб-браузера. Netscape была создана людьми, принимавшими участие в разработке Mosaic в ранние 90-е. Теперь, с деньгами и независимостью, у них было всё необходимое для поиска способов дальнейшего развития веб-технологий. Именно это послужило толчком для рождения JavaScript.

Основатель Netscape Communications и бывший участник команды Mosaic Марк Андриссен (рисунок 4.1) считал, что веб должен стать более динамичным. Анимации, взаимодействие с пользователями и другие виды интерактивности должны стать неотъемлемой частью интернета будущего. Веб нуждался в лёгком скриптовом языке (или языке сценариев – прим. ред.), способном работать с DOM, который в те дни не был стандартизирован. Существовало одно «но», являвшееся на тот момент серьёзным вызовом: этот язык не должен был предназначаться для крупных разработчиков и прочих людей, имевших отношение к инженерной стороне вопроса. Java в те дни уже активно развивалась и твёрдо заняла эту нишу. Таким образом, новый скриптовый язык должен был предназначаться для совершенно иной аудитории — дизайнеров. Очевидно, что веб был статичным, а HTML был достаточно молод и прост в освоении даже для тех, кто не имел ничего общего с программированием. Следовательно, всё, что должно было стать частью браузера и сделать веб более динамичным, должно быть максимально понятным для далёких от программирования людей. Из этого предположения родилась идея Mocha, который должен был стать тем самым простым, динамичным и доступным скриптовым языком.



Рисунок 4.1 – Марк Адриссен

И тут появляется Брендан Айк (рисунок 4.2), отец JavaScript. Айк должен был разработать для Netscape “Scheme” для браузера. Scheme – это динамичный, мощный и функциональный диалект языка программирования Lisp с максимально упрощённым синтаксисом. Вебу требовалось что-то подобное: простое в освоении, динамичное, немногословное и мощное. Айк не стал упускать шанса поработать над тем, что ему нравилось, и присоединился к команде.



Рисунок 4.2 – Брэндан Айк

Перед командой была поставлена задача подготовить работающий прототип в кратчайшие сроки. Sun Microsystems заканчивала работу над своим языком программирования Java, на тот момент называвшимся Oak, и Netscape Communications была уже готова заключить с компанией контракт, чтобы сделать Java доступным в своем браузере. Так зачем же понадобился Mocha (первое название JavaScript)? Зачем нужно было создавать абсолютно новый язык программирования при наличии готовой альтернативы? Дело в том, что Java не был предназначен для той аудитории, на которую ориентировался Mocha – скриптеры, любители, дизайнеры. Java был слишком большим и навороченным для того, чтобы выполнять эту роль. Основная идея заключалась в том, что Java должен был предназначаться для крупных разработчиков и профессиональных программистов, в то время как Mocha должен был использоваться для небольших скриптовых задач. Другими словами, Mocha должен был стать скриптовым компаньоном для Java по принципу, аналогичному тому, как взаимодействуют C/C++ и Visual Basic на платформе Windows.

Инженеры Netscape приступили к детальному изучению Java. Они даже начали разрабатывать собственную виртуальную машину Java, однако проект быстро свернули, так как она не могла достичь идеальной совместимости с виртуальной машиной Sun Microsystems.

Проблема скорейшего выбора языка стояла как никогда остро. Возможными кандидатами были Python, Tcl и Scheme. Айк должен был действовать быстро. По сравнению с конкурентами у него были два преимущества: свобода в определении набора необходимых возможностей и прямая связь с заказчиком. К несчастью, имело место и очевидное неудобство: для принятия огромного количества важных решений времени практически не было. JavaScript, a.k.a. Mocha, был рождён именно в таких условиях. В течение нескольких недель был подготовлен рабочий прототип, который затем был интегрирован в Netscape Communicator.

То, что должно было стать аналогом Scheme для браузера, вылилось в нечто совершенно иное. Рукой Айка управляли необходимость закрыть сделку с Sun и сделать Mocha скриптовым компаньоном для Java. Синтаксис должен был быть максимально близким Java. Помимо этого, от Java была унаследована семантика для большого количества устоявшихся идиом. Таким образом, Mocha был совсем не похож на Scheme. Он выглядел, как динамический Java, под оболочкой которого скрывался гибрид Scheme и Self.

Прототип Mocha был интегрирован в Netscape Communicator в мае 1995 года. Через очень короткий промежуток времени он был переименован в LiveScript, так как в тот момент слово live выглядело очень привлекательным с точки зрения маркетологов. В декабре 1995 года сделка между Netscape Communications и Sun была закрыта: Mocha/LiveScript был переименован в JavaScript и преподносился в качестве скриптового языка для выполнения небольших клиентских задач в браузере, в то время, как Java был полноценным профессиональным языком программирования для разработки сложных веб-компонентов.

Первая версия JavaScript заложила все те фундаментальные особенности, которыми этот язык знаменит и поныне. В частности, его объектная модель и функциональные особенности уже присутствовали в первой версии.

Трудно сказать, как развивались бы события, если бы Айк не успел предоставить рабочий прототип вовремя. Python, Tcl, Scheme, рассматривавшиеся в качестве альтернативы, были совершенно не похожи на Java. Sun было бы трудно принять в качестве языка-компаньона для Java варианты, в корне отличавшиеся от него. С другой стороны, Java долгое время был важной частью веба. Если бы Sun не являлись определяющим фактором, у Netscape было бы намного больше свободы в выборе языка. Но стала бы Netscape разрабатывать собственный язык или воспользовалась одним из существующих? Этого мы никогда не узнаем.

Когда Sun и Netscape закрыли сделку, и Mocha/LiveScript был переименован в JavaScript, встал ребром очень важный вопрос: что будет с конкурентами? Хоть Netscape и набирал популярность, становясь самым используемым браузером, Microsoft занималась активной разработкой Internet Explorer. С самых первых дней JavaScript показал настолько удивительные возможности в плане взаимодействия с пользователем, что соперничающим браузерам не оставалось ничего иного, кроме как в кратчайшие сроки найти готовые решения, представлявшие собой рабочие реализации JavaScript. В тот момент (и ещё достаточно долго после этого) веб-стандарты оставались достаточно слабыми. Поэтому Microsoft разработала свою реализацию JavaScript, назвав ее JScript. Убрав из названия слово Java, они смогли избежать возможных проблем с владельцами торговой марки. Однако, JScript отличался не только названием. Небольшие различия в реализации – в частности, подход к некоторым DOM функциям – оставили рябь, которая будет ощущаться ещё долгие годы. Бои за JavaScript шли на гораздо большем количестве фронтов, чем названия и таймлайны, и многие причуды этого языка появились благодаря им. Первая версия JScript появилась в Internet Explorer 3.0, увидевшем свет в августе 1996 года.

Реализация JavaScript получила свое собственное название и в Netscape. Версия, выпущенная вместе с Netscape Navigator 2.0, была известна, как Mocha. Осенью 1996 года Айк переписал большую часть Mocha, чтобы разобраться с техническими огрехами и недоработками, возникшими, как следствие спешки при разработке. Новая версия была названа SpiderMonkey. Это название используется по сей день в JavaScript-движке браузера Firefox, внука Netscape Navigator.

В течение нескольких лет JScript и SpiderMonkey были единственными движками JavaScript. Особенности обоих движков, не всегда совместимые, определили вектор развития веба на ближайшие годы.

Первой большой переменой для JavaScript после его выпуска стала стандартизация ECMA. ECMA – ассоциация, созданная в 1961 году с целью стандартизации информационных и коммуникационных систем.

Работа над стандартизацией JavaScript началась в ноябре 1996 года. Стандарту, над которым работала группа TC-39, был присвоен идентификационный номер ECMA-262. К тому моменту JavaScript активно использовался на многих веб-страницах. В этом пресс-релизе 1996 года указано количество в 300000 страниц, использующих JavaScript.

Стандартизация стала для молодого языка не только важным шагом, но и серьезным вызовом. Она открыла JavaScript для большей аудитории и дала возможность сторонним разработчикам принимать участие в развитии языка. Она также помогла держать других разработчиков в рамках. В те времена бытовало опасение, что Microsoft или кто-либо ещё могут слишком сильно отклониться от оригинальной реализации языка, что могло привести к фрагментации.

Из-за проблем с торговой маркой ECMA не могла использовать JavaScript в качестве названия. После непродолжительных дебатов было решено, что описанный стандартом язык программирования будет назван ECMAScript. На сегодняшний день JavaScript это всего лишь коммерческое название ECMAScript [7].

Первый стандарт ECMAScript был основан на версии JavaScript, входившей в состав Netscape Navigator 4 и не включал в себя важные особенности, такие как регулярные выражения, JSON, исключения и важные методы для встроенных объектов. Тем не менее, в браузере он работал намного лучше. Версия 1 была выпущена в июне 1997 года.

Вторая версия, ECMAScript 2, была выпущена в июне 1998 года, чтобы исправить несостыковки между ECMA и стандартом ISO для JavaScript (ISO/IEC 16262) и не включала в себя никаких изменений самого языка.

Интересной особенностью этой версии JavaScript было то, что интерпретатор должен был самостоятельно решать, что делать с неотловленными ошибками (и в большинстве случаев оставленными как неклассифицированные). Причиной этому стало то, что исключения ещё не были частью языка на тот момент.

После ECMAScript 2 работа продолжилась и первые большие изменения языка увидели свет. Новая версия включала в себя:

1 Регулярные выражения.

2 Блок do-while.

3 Исключения и try/catch блоки.

4 Больше встроенных функций для строк и массивов.

5 Форматирование численных выходных данных.

6 Операторы in и instanceof.

7 Улучшенная обработка ошибок.

ECMAScript 3 был выпущен в декабре 1999 года.

Эта версия ECMAScript получила очень широкое распространение. Все крупные браузеры того времени поддерживали её и продолжали поддерживать в течение многих лет. Даже сегодня многие транспайлеры в качестве выходного языка могут указывать этот стандарт. Это сделало ECMAScript 3 фундаментом для многих библиотек, даже когда были выпущены более поздние версии стандарта.

Хоть JavaScript использовался практически повсеместно, он всё ещё оставался клиентским языком программирования. Многие из его нововведений позволили ему приблизиться к тому, чтобы вырваться из этой клетки.

Netscape Navigator 6, выпущенный в ноябре 2000 года, поддерживал ECMAScript 3. Спустя почти полтора года был выпущен Firefox, браузер, основанный на кодовой базе Netscape Navigator и также поддерживавший ECMAScript 3. Бок о бок с Internet Explorer эти браузеры делали всё возможное для дальнейшего роста и развития JavaScript.

AJAX (asynchronous JavaScript and XML) — технология, появившаяся на свет в годы ECMAScript 3. Хоть она и не являлась частью стандарта, Microsoft встроила некоторые расширения для JavaScript в Internet Explorer 5. Одним из таких расширений была функция XMLHttpRequest в виде управляющего элемента ActiveX XMLHTTP. Эта функция позволяла браузеру выполнять асинхронные HTTP-запросы серверу, тем самым позволяя страницам обновляться на лету. Хотя само название AJAX было придумано значительно позже, сама техника активно использовалась в то время.

Применение XMLHttpRequest оказалось успешным, и годами позже было стандартизировано группами WHATWG и W3C.

Постоянная эволюция функциональности, разработчики, вносящие что-то новое в язык и встраивающие эти новинки в свои браузеры, до сих пор являются основополагающими факторами в развитии JavaScript и связанных с ним стандартов, таких, как CSS и HTML. Связь между отдельными группами в те дни была очень слабая, что привело к задержкам и фрагментации. Честно говоря, в наши дни разработка JavaScript организована куда лучше благодаря процедурам, позволяющим заинтересованным группам вносить свои предложения.

К сожалению, следующие несколько лет не принесли JavaScript ничего хорошего. Вместе с началом работы над ECMAScript 4 в сообществе, разбившемся на группы, начались разногласия. Одна группа утверждала, что JavaScript необходимо сделать языком для разработки крупных приложений. Эта группа предлагала множество новых опций большого объёма, требовавших внесения кардинальных изменений. Другая группа находила подобный вектор развития недопустимым. Отсутствие компромиссов и сложность некоторых предлагавшихся улучшений отодвигали выход ECMAScript 4 всё дальше и дальше.

Работа над ECMAScript 4 началась вскоре после выхода третьей версии в 1999 году. Большое количество интересных нововведений обсуждалось в Netscape. Однако интерес к ним со временем иссяк, и в 2003 году работа над новой версией ECMAScript остановилась. Был выпущен промежуточный отчёт, и некоторые разработчики, такие, как Adobe (ActionScript) и Microsoft (JScript.NET) использовали его в качестве основы для собственных движков. В 2005 году AJAX и XMLHttpRequest смогли вновь разжечь интерес к новой версии JavaScript и TC-39 возобновила работу. Проходили годы и набор нововведений рос всё больше и больше.

Комитет, разрабатывавший ECMAScript 4, включал в себя Adobe, Mozilla, Opera (неофициально) и Microsoft. Yahoo вошла в игру, когда большинство решений по стандартам и возможностям были уже приняты, прислав Дугласа Крокфорда, влиятельного JavaScript-разработчика, который тут же раскритиковал большинство новшеств, получив мощную поддержку со стороны представителя Microsoft.

То, что началось как сомнения, быстро переросло в сильную оппозицию к JavaScript. Microsoft наотрез отказывалась утверждать любую часть ECMAScript 4 и была готова к любым действиям, включая судебные тяжбы, чтобы не дать стандарту возможности быть утвержденным. К счастью, члены комитета смогли избежать судебных разбирательств. Однако из-за разногласий ECMAScript 4 продолжал топтаться на месте.

Решения, принятые на этой встрече:

1 Сфокусироваться на работе над ES3.1 при полном сотрудничестве всех участников и выпустить две совместимые версии в начале следующего года.

2 Начать работу над следующим шагом после ES3.1, который будет включать в себя синтаксические расширения, но более скромные, нежели те, которые предлагались для ES4 как в плане семантических, так и синтаксических инноваций.

3 Ради всеобщего блага было решено отказаться от некоторых из предложений ES4: пакетов, пространства имен и ранней связки. Они были признаны бесполезными для веба. Это решение – ключ к Гармонии.

4 Прочие идеи и цели ES4 были перефразированы с целью достичь консенсуса внутри комитета, например, понятие классов, основанное на имеющихся концептах ES3, совмещённых с предлагаемыми улучшениями ES3.1.

В результате ушло почти восемь лет на то, чтобы закончить разработку ECMAScript 4. Тяжелый урок для всех, кто принимал участие.

Слово «Гармония», появившееся в перечисленных решениях, стало названием проекта в будущих версиях. Harmony станет альтернативой, с которой все будут согласны. После выхода ECMAScript 3.1 (в виде версии 5, как будет рассказано чуть ниже) ECMAScript Harmony стала тем местом, где обсуждаются все новые идеи относительно JavaScript.

В 2008 году, после долгих боев, развернувшихся вокруг ECMAScript 4, сообщество сфокусировалось на работе над ECMAScript 3.1, отправив ECMAScript 4 на свалку. В 2009 году ECMAScript 3.1 был полностью завершён и одобрен всеми участниками комитета. Так как ECMAScript 4 считался своеобразным вариантом ECMAScript несмотря на отсутствие хоть какого-либо релиза, было решено переименовать ECMAScript 3.1 в ECMAScript 5, чтобы избежать недоразумений.

ECMAScript 5 стал одной из самых поддерживаемых версий JavaScript, став также целью компиляции многих транспайлеров. ECMAScript 5 получил полную поддержку в браузерах Firefox 4 (2011), Chrome 19 (2012), Safari 6 (2012), Opera 12.10 (2012) и Internet Explorer 10 (2012).

ECMAScript 5 был достаточно скромным улучшением ECMAScript 3, включавшим в себя:

1 Геттеры/сеттеры.

2 Разделители-запятые в массивах и объектах.

3 Возможность использовать зарезервированные слова в качестве свойств объекта.

4 Новые методы объектов (create, defineProperty, keys, seal, freeze, getOwnPropertyNames и т.д.).

5 Новые методы массивов (isArray, indexOf, every, some, map, filter, reduce и т.д.).

6 String.prototype.trim и доступ к свойствам.

7 Новые методы Date (toISOString, now, toJSON).

8 Привязывание функций.

9 JSON.

10 Неизменяемые глобальные объекты (undefined, NaN, Infinity).

11 Строгий режим.

12 Другие небольшие изменения (parseInt игнорирует ведущие нули, функции в throw имеют значение и т.д.).

Ни одно из этих изменений не требовало внесения изменений в синтаксис. Геттеры и сеттеры в то время уже неофициально поддерживались некоторыми браузерами. Новые методы объектов должны были улучшить большое программирование, дав программистам больше инструментов для проверки соблюдения определённых инвариантов (Object.seal, Object.freeze, Object.createProperty). Строгий режим также стал мощным инструментом в этой области, позволив избежать большого числа ошибок. Дополнительные методы массивов улучшили определённые функциональные паттерны (map, reduce, filter, every, some. Ещё одним большим нововведением является JSON: формат данных, основанный на JavaScript, который теперь поддерживается нативно благодаря JSON.stringify и JSON.parse. Другие изменения касаются небольших улучшений, основанных на практическом опыте. В целом, ECMAScript 5 был небольшим улучшением, которое приукрасило JavaScript в плане юзабилити как для небольших скриптов, так и для более объемных проектов. Тем не менее, большое количество хороших идей, предлагавшихся для ECMAScript 4, так и не были реализованы и ждали своего возвращения в ECMAScript Harmony.

ECMAScript 5 получил обновление в 2011 году под названием ECMAScript 5.1. Этот релиз вносил ясность в некоторые неоднозначные пункты стандарта, но никаких новых возможностей в нем не было. Все новые возможности были запланированы для следующего большого релиза ECMAScript [8].

План ECMAScript Harmony стал основой для последующих улучшений JavaScript. Многие идеи из ECMAScript 4 канули в лету ради всеобщего блага, однако некоторые были пересмотрены. ECMAScript 6, позже переименованный в ECMAScript 2015, должен был принести большие перемены. Почти все обновления, так или иначе влиявшие на синтаксис, были отложены именно для этой версии. К 2015 году комитет, наконец, смог побороть все внутренние разногласия, и ECMAScript 6 увидел свет. Большинство производителей браузеров уже работали над поддержкой этой версии, однако до сих пор не все браузеры имеют полную совместимость с ECMAScript 2015.

Выход ECMAScript 2015 стал причиной резкого роста популярности транспайлеров, таких, как Babel или Traceur. Благодаря тому, что производители этих транспайлеров следили за работой технического комитета, у многих людей появилась возможность испытать преимущества ECMAScript 2015 задолго до его выхода.

Некоторые из основных возможностей ECMAScript 4 были реализованы в этой версии с несколько иным подходом. Например, классы в ECMAScript 2015 – это нечто большее, чем просто синтаксический сахар поверх прототипов. Подобный подход облегчает разработку и внедрение новых возможностей.

Краткий список новых возможностей включает в себя:

1 Let (лексическая) и const (неизменяемая) привязки.

2 Стрелочные функции (короткие анонимные функции) и лексическое this.

3 Классы (синтаксический сахар поверх прототипов).

4 Улучшения объектных литералов (вычисляемые ключи, укороченные определения методов и т.д.).

5 Шаблонные строки.

6 Промисы.

7 Генераторы, итерируемые объекты, итераторы и for..of.

8 Параметры функций по умолчанию и оператор rest.

9 Spread-синтакис.

10 Деструктуризация.

11 Модульный синтаксис.

12 Новые коллекции (Set, Map, WeakSet, WeakMap).

13 Прокси и Reflect.

14 Тип данных Symbols.

15 Типизированные массивы.

16 Наследование классов.

17 Оптимизация хвостовой рекурсии.

18 Упрощённая поддержка Unicode.

19 Двоичные и восьмеричные литералы.

Все эти возможности открыли JavaScript для ещё большего количества программистов и внесли существенный вклад в большое программирование.

Некоторых может удивить, как могло такое количество новых возможностей проскочить мимо процесса стандартизации, во время которого был загублен ECMAScript 4. Хотелось бы отметить, что большинство наиболее агрессивных инноваций ECMAScript 4, таких, как пространства имён или опциональное типирование, были забыты и к ним больше не возвращались, в то время, как другие были переосмыслены с учётом возникших возражений. Работа над ECMAScript 2015 была очень тяжёлой и заняла почти шесть лет (и даже больше, учитывая время, необходимое на реализацию). Но сам факт того, что технический комитет ECMAScript смог справиться с таким трудным заданием, стал добрым знамением.

В 2016 году увидело свет небольшое обновление ECMAScript. Эта версия стала результатом нового процесса подготовки, принятого в TC-39. Все новые предложения должны пройти через четыре стадии. Предложение, достигшее четвёртой стадии, имеет все шансы быть включенным в следующую версию ECMAScript (однако комитет имеет право отложить его для более поздней версии). Таким образом, каждое предложение разрабатывается индивидуально (разумеется, с учётом его взаимодействия с другими предложениями), не тормозя разработку ECMAScript.

Если предложение готово к включению в стандарт, и достаточное количество других предложений достигло четвёртой стадии, в свет выходит новая версия ECMAScript.

Версия, выпущенная в 2016 году, была очень маленькой. Она включала в себя:

1 Оператор возведения в степень (\*\*).

2 Array.prototype.includes.

3 Несколько незначительных поправок (генераторы не могут быть использованы с new и т.д.).

Самым важным предложением, достигшим четвёртой стадии, является async/await. это расширение синтаксиса для JavaScript, которое делает работу с промисами более приятной.

Другие предложения, достигшие четвёртой стадии, совсем небольшие:

1 Object.values и Object.entries.

2 Выравнивание строк.

3 Object.getOwnPropertyDescriptors.

4 Разделители-запятые в параметрах функций.

Все эти предложения предназначены для релиза 2017 года, однако комитет имеет право отложить их до следующего релиза. Но даже одно лишь дополнение в лице async/await будет потрясающим.

Будущее на этом не заканчивается. Давайте посмотрим на некоторые другие предложения, чтобы получить представление о том, что ждёт нас впереди. Вот несколько самых интересных:

1 SIMD API.

2 Асинхронные итераторы (async/await + итерация).

3 Стрелочные генераторы.

4 Операции с 64-битными целыми числами.

5 Области (изоляции состояний).

6 Общая память и Atomics.

JavaScript всё больше становится похож на язык общего назначения. Но есть ещё одна большая деталь в будущем JavaScript, которая внесёт свои коррективы.

Огромное количество библиотек и фреймворков, появившихся после выхода ECMAScript 5, а также общее развитие языка, сделали JavaScript интересной целью для других языков. Для больших кодовых структур функциональная совместимость является ключевой потребностью. Возьмите, к примеру, игры. Самым распространённым языком, на котором пишутся игры, является C++, благодаря чему их можно портировать на большое количество архитектур. Тем не менее портирование для браузера Windows или консольной игры считалось невыполнимой задачей. Однако это стало возможным благодаря стремительному развитию и небывалой эффективности сегодняшних виртуальных машин JavaScript. Именно для выполнения подобных задач на свет появились инструменты вроде Emscripten.

Быстро сориентировавшись в ситуации, Mozilla начала работу над тем, чтобы сделать JavaScript подходящей целью для компиляторов. Так на свет появился Asm.js – подмножество JavaScript, идеально подходящее в качестве подобной цели. Виртуальные машины JavaScript могут быть оптимизированы для распознавания этого подмножества и производства кода, намного лучшего, чем тот, который генерируют текущие виртуальные машины. Благодаря JavaScript браузеры медленно становятся новой целью для компиляторов.

И всё же существуют огромные ограничения, которые не в состоянии преодолеть даже Asm.js. В JavaScript необходимо внести такие изменения, которые расходятся с его текущим предназначением. Нужно что-то совершенно иное для того, чтобы сделать веб достойной целью для других языков программирования. И именно для этого предназначен WebAssembly – низкоуровневый язык программирования для веба. Любая программа может быть скомпилирована в WebAssembly при помощи подходящего компилятора и затем запущена в подходящей виртуальной машине (виртуальные машины JavaScript могут предоставить необходимый уровень семантики). Первые версии WebAssembly имеют стопроцентную совместимость со спецификацией Web.js. WebAssembly обещает не только более быстрое время загрузки (байт-код обрабатывается быстрее, чем текст), но и возможность оптимизации, недоступной в Asm.js. Представьте себе интернет с идеальной функциональной совместимостью между JavaScript и вашим языком программирования.

На первый взгляд это может помешать росту JavaScript, но на самом деле всё совершенно иначе. Благодаря тому, что другие языки и фреймворки получат функциональную совместимость с JavaScript, он сможет продолжать свое развитие в качестве языка общего назначения. И WebAssembly является необходимым инструментом для этого.

В настоящий момент dev-версии Chrome, Firefox и Microsoft Edge имеют начальную поддержку WebAssembly и способны проигрывать демо-приложения.

История JavaScript длинна и полна неожиданных поворотов. Изначально предложенный в качестве «Scheme для веба», он позаимствовал свой синтаксис у Java. Его первый прототип был разработан за несколько недель. Подстраиваясь под требования рынка, он сменил три названия менее чем за два года, после чего был стандартизирован и получил название, болee подходящее для кожного заболевания. После трёх успешных релизов язык варился в адских котлах почти восемь лет. Затем, благодаря успеху одной-единственной технологии (AJAX), сообщество смогло побороть противоречия и возобновить разработку. Версия 4 была заброшена, а небольшое обновление, известное, как версия 3.1, было переименовано в версию 5. Версия 6 провела в разработке много лет (опять), но на этот раз комитет успешно закончил работу, сменив цифру в названии на 2015. Это было очень большое обновление, и его реализация заняла много времени. В результате JavaScript получил второе дыхание. Сообщество оживилось как никогда до этого. Благодаря Node.js, V8 и другим проектам JavaScript поднялся на высоты, о которых разработчики первой версии даже не задумывались, а благодаря Asm.js и WebAssembly он взлетит ещё выше. Активные предложения, пребывающие в разных стадиях, делают будущее JavaScript чистым и безоблачным. Пройдя долгий путь, полный неожиданных поворотов и препятствий, JavaScript остаётся одним из самых успешных языков в истории программирования. И это – лучшее доказательство его надежности. Всегда ставьте на JavaScript.

## **4.2 Typescript**

TypeScript – это язык программирования со статической типизацией, позиционирующий себя, как язык расширяющий возможности JavaScript.

TypeScript код компилируется в высококачественный JavaScript код с которым в дальнейшем можно запускать как на клиентской стороне (браузер), так и на стороне сервера (nodejs). Качество сгенерированного кода сопоставимо с кодом, написанным профессиональным разработчиком с большим стажем. Мультиплатформенный компилятор TypeScript отличается высокой скоростью компиляции и распространяется по лицензии Apache, а его разработка сопровождается высокоэффективной поддержкой со стороны разработчиков со всего мира.

Разработчиком языка TypeScript является Андерс Хейлсберг, так же известный как создатель языков Turbo Pascal, Delphi, C#. С момента своего анонсирования компанией MicroSoft в 2012 году, TypeScript не перестает развиваться и склоняет все больше профессиональных разработчиков писать свои программы на нем. Поэтому на текущий момент, практически невозможно найти библиотеку, которая бы не была портирована на TypeScript. Мотивацией, к созданию TypeScript, послужила увеличивающаяся сложность приложений, которые перестали уступать своим старшим братьям, desktop приложениям.

Прежде всего TypeScript предназначен для выявления ошибок на этапе компиляции, а не на этапе выполнения. Кроме того, за счет системы типов, разработчики получают такие возможности, как подсказки и переходы по коду, которые значительно ускоряют процесс разработки. Помимо этого, система типов, в значительной степени, избавляет разработчиков от комментирования. Отпадает потребность в комментировании происхождения кода, которая в отличии от предназначения кода, занимает большую часть времени. Также при уделении малого внимания архитектуре, система типов накладывает ограничения, которые выявляют её проблемы на более ранних этапах, что значительно снижает стоимость перепроектирования.

Если при создании нового проекта планируется использовать JavaScript код, оставшийся от предыдущих проектов, то это не составит никакой проблемы. Компилятор TypeScript отлично справляется с динамическим JavaScript кодом, включенным в свою типизированную среду и даже выявляет в нем ошибки. Кроме того, при компиляции .ts файлов в .js, дополнительно генерируются файлы декларации .d.ts, с помощью которых, разработчики, которые пишут свои программы исключительно на JavaScript, будут иметь полноценный автокомплит.

Опытным разработчикам TypeScript значительно сокращает время на устранение ошибок и выявление багов, которые, порой, не так просто отыскать в динамической среде JavaScript. Кроме того, сокращается объем комментариев, на которые тоже уходит немало времени.

В случае, если для разработчика, TypeScript является первым типизированным языком, то они должны знать, что его изучение, значительно ускорит процесс их профессионального роста, так как типизированный мир откроет аспекты программирования, которые в динамических языках просто не очевидны.

Помимо это, TypeScript позволяет писать более понятный-читаемый код, который максимально описывает предметную область, за счет чего архитектура становится более выраженной, а разработка неявно увеличивает профессиональный уровень программиста.

В заключение, всё это, в своей совокупности, сокращает время разработки программы, снижая её стоимость и предоставляя разработчикам возможность поскорее приступить к реализации нового, ещё более интересного, проекта [9].

## **4.3 Vite**

### **4.3.1 Архитектура Vite**

Vite – это инструмент сборки, призванный обеспечить более быструю и эффективную разработку современных веб-проектов. Одним из ключевых принципов Vite является стремление к созданию легковесного ядра с минимальным набором API. Такой подход означает, что в основную функциональность Vite включены только самые необходимые инструменты и возможности, поддерживающие наиболее распространенные сценарии разработки веб-приложений. Функциональность, которая может быть реализована в виде внешних плагинов, намеренно выносится за пределы ядра. Это решение продиктовано желанием обеспечить долгосрочную поддерживаемость и стабильность основного инструмента. Подобная архитектура позволяет Vite оставаться гибким и адаптируемым к различным потребностям разработчиков, не раздувая при этом его кодовую базу.

Для обеспечения расширяемости Vite использует плагинную систему, основанную на Rollup. Это означает, что разработчики могут добавлять в свои проекты дополнительную функциональность, устанавливая и настраивая соответствующие плагины. В качестве примера такого плагина можно привести vite-plugin-pwa, который обеспечивает поддержку Progressive Web Apps. Существует обширная коллекция поддерживаемых сообществом плагинов, охватывающих широкий спектр задач и потребностей.

Тесное сотрудничество между командами Vite и Rollup играет важную роль в обеспечении совместимости плагинов между обычными проектами Rollup и проектами Vite. Это стратегическое решение позволяет Vite использовать зрелую и обширную экосистему плагинов Rollup, значительно расширяя его возможности без необходимости разработки всей дополнительной функциональности с нуля. В долгосрочной перспективе планируется переносить необходимые расширения в основной API плагинов Rollup, что свидетельствует о стремлении к интероперабельности в рамках JavaScript-инструментария.

### **4.3.2 Работа с модулями и зависимостями в экосистеме Vite**

Vite активно внедряет и продвигает использование современных веб-технологий и стандартов. Одним из ключевых аспектов этой философии является требование написания исходного кода с использованием модулей ECMAScript (ESM). ESM представляет собой стандартный формат модулей в современном JavaScript, который обладает рядом преимуществ по сравнению с более старыми форматами, такими как CommonJS. К этим преимуществам относятся улучшенный статический древовидный шейкинг (tree-shaking), лучшая поддержка нативных модулей в браузерах и более эффективная загрузка кода. Таким образом, Vite способствует переходу на более современные и производительные методы разработки.

Для обеспечения работы с ESM, Vite требует предварительной сборки зависимостей, которые не соответствуют этому формату. Этот процесс включает преобразование таких зависимостей в формат ESM перед их использованием в проекте. Для этой цели часто используются такие быстрые инструменты, как esbuild. Предварительная сборка зависимостей позволяет Vite более эффективно использовать встроенные механизмы загрузки модулей в браузере, что приводит к улучшению производительности во время разработки и сокращению времени загрузки страниц.

Vite также поощряет использование веб-воркеров с применением стандартного синтаксиса new Worker(). Веб-воркеры позволяют запускать JavaScript-код в фоновых потоках, что особенно полезно для выполнения ресурсоемких задач без блокировки основного потока пользовательского интерфейса. Поддержка веб-воркеров отражает стремление Vite помочь разработчикам создавать отзывчивые и производительные веб-приложения, способные эффективно использовать возможности современных браузеров.

Важно отметить, что Vite не предназначен для непосредственного использования модулей Node.js в браузере. Это связано с различиями в доступных API между средами Node.js и браузеров, а также с потенциальным увеличением размера итоговой сборки при включении серверного кода в браузерный. Такой подход способствует более четкому разделению между frontend- и backend-частями приложения, что приводит к лучшей организации кода и снижению риска возникновения проблем с совместимостью.

### **4.3.3 Механизмы внутренней оптимизации Vite**

Производительность является одним из фундаментальных приоритетов Vite с момента его создания. Это проявляется как в скорости работы сервера разработки, особенно в части горячей замены модулей (HMR), так и в оптимизации производственных сборок для обеспечения быстрого времени загрузки конечными пользователями.

Архитектура сервера разработки Vite специально разработана для обеспечения исключительно быстрой работы HMR даже в крупных проектах. Это достигается за счет использования нативных возможностей браузеров по работе с ESM и выполнения точечных обновлений без полной перезагрузки страницы. Такой подход значительно улучшает опыт разработчиков, предоставляя практически мгновенную обратную связь при внесении изменений в код, что повышает продуктивность и делает процесс разработки более итеративным.

Для выполнения ресурсоемких задач, таких как предварительная сборка зависимостей, Vite использует нативные инструменты, такие как esbuild, известный своей высокой скоростью работы. Для компиляции кода в рамках некоторых фреймворков, например React, может использоваться SWC, также написанный на языке, отличном от JavaScript, что обеспечивает значительное повышение производительности. Основная часть кода Vite написана на JavaScript, что обеспечивает баланс между скоростью и гибкостью, а также облегчает участие широкого круга разработчиков в проекте. В некоторых случаях, особенно при использовании определенных фреймворков, плагины Vite могут интегрироваться с Babel для компиляции пользовательского кода, когда требуются более сложные преобразования или поддержка устаревших браузеров.

На этапе сборки для продакшена Vite в настоящее время использует Rollup. Этот выбор обусловлен тем, что для конечного приложения приоритет отдается размеру бандла и наличию обширной экосистемы плагинов, позволяющих проводить глубокую оптимизацию, такую как разделение кода и удаление неиспользуемого кода (tree-shaking). Хотя такие инструменты, как esbuild, могут обеспечивать более высокую скорость сборки, зрелая экосистема плагинов Rollup и его возможности по оптимизации размера бандла делают его предпочтительным вариантом для создания готовых к развертыванию приложений.

Внутреннее устройство Vite продолжает развиваться, и команда проекта активно изучает и внедряет новые библиотеки и подходы для дальнейшего улучшения опыта разработчиков, сохраняя при этом стабильный API. Это свидетельствует о стремлении Vite оставаться передовым инструментом в области веб-разработки, постоянно адаптируясь к новым вызовам и возможностям.

### **4.3.4 Vite как универсальная платформа для веб-фреймворков**

Хотя Vite является мощным инструментом для непосредственной разработки приложений, он также задумывался как отличная основа для создания новых веб-фреймворков. Ядро Vite остается фреймворк-агностичным, предоставляя необходимую инфраструктуру сборки без навязывания каких-либо конкретных фреймворковых соглашений. Такая универсальность делает Vite привлекательным выбором для разработчиков фреймворков, позволяя им сосредоточиться на уникальных особенностях своих инструментов, используя при этом эффективный процесс сборки, предоставляемый Vite.

Для различных популярных UI-фреймворков, таких как React, Vue.js, Svelte и другие, существуют хорошо разработанные и активно поддерживаемые плагины. Эти плагины обеспечивают бесшовную интеграцию основной функциональности Vite со специфическими требованиями и соглашениями каждого фреймворка, предоставляя разработчикам адаптированный опыт.

Vite предоставляет обширный JavaScript, который позволяет авторам фреймворков глубоко интегрироваться с процессом сборки Vite и настраивать опыт разработки для своих пользователей. Гибкость и контроль, обеспечиваемые этим API, позволяют создавать высокоуровневые абстракции и инструменты, отвечающие конкретным потребностям фреймворка.

Vite также включает встроенную поддержку примитивов для рендеринга на стороне сервера (SSR), которые часто встречаются во фреймворках более высокого уровня, но в Vite предоставляются на более низком уровне для облегчения разработки фреймворков. Поддержка SSR имеет важное значение для улучшения начальной производительности загрузки страниц, SEO и обеспечения лучшего пользовательского опыта для определенных типов веб-приложений. Включение этих примитивов на уровне ядра демонстрирует понимание Vite критической роли SSR в современной веб-разработке и стремление предоставить авторам фреймворков фундаментальные строительные блоки для легкого включения этой важной функции.

Плагины Vite также могут способствовать обмену функциональностью и общими шаблонами между различными фреймворками, способствуя повторному использованию кода и согласованности в экосистеме веб-разработки. Кроме того, Vite совместим и может интегрироваться с различными backend-фреймворками, такими как Ruby и Laravel. Это позволяет использовать Vite для управления frontend-активами в приложениях, построенных с использованием этих backend-фреймворков, обеспечивая современный и эффективный опыт разработки для full-stack проектов.

### **4.3.5 Активная экосистема и сообщество**

Развитие и успех Vite являются результатом совместных усилий не только основной команды Vite, но и сопровождающих фреймворки и плагины, широкого сообщества пользователей и других заинтересованных сторон в экосистеме веб-разработки. Этот дух сотрудничества имеет решающее значение для стимулирования инноваций, удовлетворения разнообразных потребностей сообщества и обеспечения долгосрочной устойчивости проекта.

Проектам, которые используют Vite, рекомендуется активно участвовать в его разработке, внося исправления ошибок, предлагая новые функции и предоставляя ценные отзывы. Участие сообщества играет важную роль в определении будущего направления развития проекта и обеспечении его соответствия постоянно меняющимся потребностям пользователей.

Команда Vite тесно сотрудничает с сопровождающими ключевых проектов в экосистеме Vite, чтобы минимизировать риск возникновения регрессий с каждым новым выпуском. Это взаимодействие помогает обеспечить плавный процесс обновления для пользователей этих проектов и способствует общей стабильности экосистемы.

Для обеспечения стабильности используются такие инструменты, как vite-ecosystem-ci, который автоматически запускает наборы тестов CI основных проектов, основанных на Vite, для выбранных pull request'ов в репозитории Vite. Этот строгий процесс автоматического тестирования помогает понять потенциальное влияние изменений в Vite на более широкую экосистему до их официального выпуска, позволяя команде выявлять и устранять любые регрессии на ранних этапах. Внедрение и активное использование vite-ecosystem-ci демонстрируют твердую приверженность поддержанию стабильности и совместимости Vite с окружающей экосистемой, значительно снижая вероятность критических изменений и обеспечивая более надежный и предсказуемый опыт для пользователей.

Основная цель этих совместных усилий и мер по обеспечению стабильности заключается в том, чтобы устранять любые выявленные регрессии до того, как они затронут конечных пользователей, и дать проектам возможность своевременно и уверенно обновляться до новых версий Vite [10].

Философия Vite основана на нескольких ключевых принципах: стремление к легкому и расширяемому ядру, ориентация на современные стандарты веб-разработки, прагматичный подход к производительности, позиционирование Vite как фундамента для создания веб-фреймворков и признание важности активной экосистемы и сообщества. Эти принципы в совокупности способствуют достижению основной цели Vite – обеспечить более быстрый, эффективный и приятный опыт разработки современных веб-приложений.

## **4.4 React**

React представляет собой JavaScript-библиотеку, разработанную для создания интерактивных пользовательских интерфейсов. В отличие от полнофункциональных фреймворков, React фокусируется исключительно на уровне представления, что позволяет разработчикам гибко интегрировать его с другими инструментами и библиотеками в зависимости от потребностей проекта. Эта библиотека получила широкое распространение в современной веб-разработке благодаря своей эффективности, компонентной архитектуре и богатой экосистеме. React используется для разработки как одностраничных, так и многостраничных приложений, а также для создания сложных пользовательских интерфейсов, примером чему служит его применение в таких известных платформах, как Netflix и Facebook. В данном руководстве будет подробно рассмотрена философия React, его ключевые концепции, преимущества использования и история развития.

В основе React лежат несколько ключевых принципов проектирования, которые определяют его подход к разработке пользовательских интерфейсов:

1 Компонентно-ориентированная архитектура. Фундаментальной идеей React является компонентно-ориентированная архитектура. Это означает, что пользовательский интерфейс разбивается на иерархию независимых и многократно используемых компонентов. Каждый компонент отвечает за отображение определенной части интерфейса, обладая собственной логикой и контролируя свой внешний вид. Такая модульность позволяет командам разработчиков параллельно работать над различными частями приложения, не затрагивая код друг друга. Более того, React спроектирован таким образом, что компоненты, созданные разными разработчиками или командами, могут беспрепятственно комбинироваться для построения сложных приложений.

2 Декларативный подход. React придерживается декларативного подхода к разработке пользовательских интерфейсов. Вместо того чтобы описывать пошаговые инструкции для обновления DOM (Document Object Model), разработчик лишь указывает желаемое состояние пользовательского интерфейса, а React берет на себя ответственность за эффективное обновление DOM для достижения этого результата. Этот подход упрощает разработку, делая код более предсказуемым и фокусируя внимание разработчика на том, как должен выглядеть интерфейс в различных ситуациях, а не на деталях его обновления. Кроме того, React абстрагирует работу с DOM, что снижает вероятность ошибок и облегчает поддержку кода [11].

3 Однонаправленный поток данных. Концепция однонаправленного потока данных является еще одним важным принципом React. Данные в React-приложении передаются в одном направлении – сверху вниз по иерархии компонентов через так называемые пропсы (props). Пропсы являются неизменяемыми (read-only) для дочерних компонентов. Для обеспечения обратной связи, когда дочернему компоненту необходимо сообщить родителю об изменении или инициировать обновление данных, используются функции обратного вызова (callback functions), которые родительский компонент передает дочернему через пропсы. Такой подход делает поток информации в приложении более прозрачным и предсказуемым, что значительно упрощает отладку и понимание структуры приложения [12].

4 Виртуальный DOM. Для повышения производительности React использует виртуальный DOM (Virtual Document Object Model). React создает в памяти виртуальное представление DOM. При изменении состояния компонента React сравнивает новый виртуальный DOM с его предыдущей версией, определяя минимальное количество изменений, которые необходимо внести в реальный DOM. Затем React обновляет только те части реального DOM, которые действительно изменились. Операции с виртуальным DOM выполняются значительно быстрее, чем непосредственные манипуляции с реальным DOM. Этот механизм позволяет React эффективно обновлять пользовательский интерфейс, минимизируя затраты ресурсов и обеспечивая высокую производительность даже в сложных приложениях с частыми обновлениями [13].

5 Синтаксис JSX. React использует специальный синтаксис под названием JSX (JavaScript XML). JSX представляет собой расширение синтаксиса JavaScript, которое позволяет разработчикам писать HTML-подобные структуры непосредственно внутри своего JavaScript-кода. Такой подход облегчает описание структуры пользовательского интерфейса и делает код более интуитивно понятным и читабельным. Несмотря на внешнее сходство с HTML, JSX-код компилируется в обычный JavaScript с использованием функции React.createElement, что позволяет React эффективно управлять DOM [14].

В React существуют два основных типа компонентов: функциональные и классовые. Функциональные компоненты являются основными строительными блоками UI в React. Они представляют собой обычные JavaScript-функции, которые возвращают JSX. С появлением хуков в React 16.8 функциональные компоненты получили возможность управлять собственным состоянием и выполнять побочные эффекты, что ранее было доступно только классовым компонентам. Классовые компоненты создаются с использованием синтаксиса классов ES6 и обладают специальными методами жизненного цикла, такими как componentDidMount, componentDidUpdate и componentWillUnmount, которые позволяют выполнять определенные действия на различных этапах существования компонента. Несмотря на то, что функциональные компоненты с хуками стали более предпочтительным подходом для большинства задач, понимание классовых компонентов все еще может быть полезным при работе с устаревшим кодом или некоторыми сторонними библиотеками.

Состояние (state) и пропсы (props) являются ключевыми механизмами для управления данными в React-приложениях. Состояние представляет собой внутренние данные компонента, которые могут изменяться со временем и влиять на его отображение. Изменение состояния компонента приводит к его повторной отрисовке. Пропсы, напротив, являются способом передачи данных от родительских компонентов к дочерним. Пропсы являются неизменяемыми внутри дочернего компонента и используются для его настройки и передачи ему необходимой информации. Этот механизм обеспечивает однонаправленный поток данных в приложении.

Начиная с версии 16.8, React предоставляет хуки – специальные функции, которые позволяют "подключаться" к функциональности React из функциональных компонентов. Для управления состоянием наиболее часто используются хуки useState, useReducer и useContext. Хук useState является самым простым и распространенным способом добавления состояния в функциональный компонент. Он возвращает пару значений: текущее состояние и функцию для его обновления. Хук useReducer представляет собой более мощную альтернативу useState и используется для управления сложным состоянием, особенно когда следующее состояние зависит от предыдущего или когда логика обновления состояния включает несколько подзначений. Он принимает функцию-редьюсер, начальное состояние и возвращает текущее состояние и функцию dispatch для отправки действий в редьюсер. Хук useContext предназначен для обмена данными между компонентами без необходимости явной передачи пропсов через каждый уровень. Он позволяет получать доступ к значению контекста, созданного с помощью createContext, из любого дочернего компонента, находящегося в дереве компонентов ниже соответствующего провайдера контекста.

Хук useEffect используется для выполнения так называемых побочных эффектов в функциональных компонентах. Побочные эффекты представляют собой действия, которые выходят за рамки обычного потока данных React, такие как запросы к API, подписка на события браузера или установка таймеров. Хук useEffect принимает два аргумента: функцию настройки (setup function), которая содержит логику эффекта, и необязательный массив зависимостей. Функция настройки может возвращать функцию очистки (cleanup function), которая выполняется при размонтировании компонента или перед следующим запуском эффекта, что позволяет предотвращать утечки памяти и другие проблемы. Массив зависимостей используется для контроля времени выполнения эффекта: эффект запускается только в том случае, если значения хотя бы одной из зависимостей изменились с момента последнего рендеринга. Существует также хук useLayoutEffect, который является вариантом useEffect и запускается синхронно после всех изменений DOM. Он используется для выполнения эффектов, связанных с макетом и измерениями DOM, например, для получения размеров элемента перед его отрисовкой.

React рекомендует использовать композицию вместо наследования для повторного использования кода между компонентами. Композиция в React достигается путем передачи компонентов в качестве пропсов и использования специального пропса children, который позволяет родительскому компоненту вставлять произвольный JSX в дочерний компонент. Наследование, напротив, может привести к тесной связи между компонентами, что затрудняет их поддержку и масштабирование. Композиция обеспечивает большую гибкость и лучшую организацию кода, позволяя создавать сложные пользовательские интерфейсы путем объединения более простых и многократно используемых компонентов.

Использование React в веб-разработке предоставляет ряд значительных преимуществ:

1 Повышенная производительность. Благодаря использованию виртуального DOM, React обеспечивает высокую производительность веб-приложений. Виртуальный DOM позволяет обновлять только те части пользовательского интерфейса, которые действительно изменились, минимизируя количество операций с реальным DOM, которые являются ресурсоемкими. React использует эффективный алгоритм сравнения (diffing) виртуального DOM с его предыдущей версией, что позволяет быстро определить необходимые изменения и применить их к реальному DOM.

2 Многократное использование компонентов. Компонентная архитектура React способствует многократному использованию кода. Разработчики могут создавать переиспользуемые UI-компоненты, которые могут быть использованы в различных частях приложения, что значительно сокращает время и усилия на разработку. Это также обеспечивает консистентность пользовательского интерфейса по всему приложению.

3 Большое и активное сообщество. React обладает одним из самых больших и активных сообществ среди JavaScript-библиотек. Это означает наличие огромного количества ресурсов для обучения (документация, туториалы, курсы) и решения возникающих проблем (форумы, сообщества в социальных сетях). Активное сообщество также способствует постоянному развитию и улучшению библиотеки, а также созданию большого количества готовых к использованию библиотек и инструментов.

4 SEO-оптимизация. React позволяет создавать SEO-дружелюбные веб-приложения. Одним из ключевых факторов является поддержка рендеринга на стороне сервера (SSR – Server-Side Rendering). SSR позволяет генерировать HTML-код страницы на сервере и отправлять его в браузер пользователя, что ускоряет первоначальную загрузку страницы и облегчает индексацию контента поисковыми системами.

5 Кроссплатформенная разработка с React Native. Экосистема React включает React Native – фреймворк, который позволяет использовать знания React для разработки нативных мобильных приложений под iOS и Android. Это позволяет разработчикам использовать единую кодовую базу для создания как веб-, так и мобильных приложений, что значительно повышает эффективность разработки и снижает затраты ресурсов.

6 Простота изучения и использования. React считается относительно простым в изучении и использовании, особенно для разработчиков, уже знакомых с JavaScript, HTML и CSS. Основные концепции React достаточно интуитивны, а наличие обширной документации и учебных материалов облегчает процесс освоения библиотеки [15].

История развития React началась в 2011 году, когда инженер Facebook Джордан Волке создал первую версию библиотеки. На его работу оказал влияние XHP – компонентный HTML-фреймворк для PHP, также разработанный в Facebook. Первоначально React использовался внутри Facebook для разработки динамичных пользовательских интерфейсов, в частности для новостной ленты. Позже, в 2012 году, React был применен для разработки ленты Instagram.

Знаковым событием в истории React стало открытие его исходного кода в мае 2013 года на конференции JSConf US. Несмотря на первоначальный скептицизм со стороны сообщества, React быстро завоевал популярность благодаря своим инновационным подходам, особенно использованию виртуального DOM для повышения производительности. Активное участие сообщества разработчиков способствовало дальнейшему развитию библиотеки, появлению новых инструментов и расширению ее возможностей.

Ключевыми этапами в развитии React стали выпуск React Native в 2015 году, который позволил использовать React для разработки мобильных приложений, и выход версии React 16 (Fiber) в 2017 году, которая принесла значительные улучшения производительности и поддержку асинхронного рендеринга. В 2018 году была представлена версия React 16.8, в которой появились Hooks – функции, позволившие использовать состояние и другие возможности React в функциональных компонентах без написания классов. Развитие React продолжается и сегодня, с регулярным выпуском новых функций и улучшений, таких как Server Components и Actions [16].

React зарекомендовал себя как мощный и гибкий инструмент для современной веб-разработки. Его ключевые преимущества, такие как высокая производительность благодаря виртуальному DOM, возможность многократного использования компонентов, обширное и активное сообщество, поддержка SEO-оптимизации и кроссплатформенной разработки с помощью React Native, делают его одним из самых популярных и востребованных решений среди разработчиков. История развития React демонстрирует его постоянное стремление к инновациям и улучшению, о чем свидетельствует появление таких значимых нововведений, как архитектура Fiber и хуки. В настоящее время команда React продолжает активно работать над новыми функциями, такими как React Compiler для автоматической оптимизации производительности, Server Components для улучшения работы с сервером и SEO, а также Actions для упрощения обработки данных на клиенте и сервере. Эти разработки указывают на то, что React остается в авангарде веб-технологий и продолжит играть важную роль в развитии веб-приложений в будущем.

## **4.5 Управление состоянием**

В мире разработки React-приложений управление состоянием является краеугольным камнем для создания интерактивных и динамичных пользовательских интерфейсов. Компонентная архитектура React сама по себе обеспечивает эффективный способ организации UI, но по мере роста сложности приложения возникает потребность в более централизованном и предсказуемом подходе к управлению данными, которые используются этими компонентами. Проблема «пробрасывания пропсов» через множество уровней компонентов и поддержание консистентности состояния в больших приложениях становятся серьезными вызовами [17].

### **4.5.1 Redux**

Именно здесь на сцену выходит Redux – библиотека JavaScript, предназначенная для управления и обновления состояния приложения. Redux предлагает архитектурный паттерн, основанный на нескольких ключевых принципах, которые делают управление состоянием более прозрачным и предсказуемым.

Основными принципами Redux являются:

1 Единственный источник истины: Глобальное состояние всего приложения хранится в одном месте – в так называемом сторе. Такой подход упрощает создание универсальных приложений, где состояние сервера может быть легко передано клиенту. Кроме того, единое дерево состояния облегчает отладку и инспектирование приложения, а также позволяет реализовать такие функции, как отмена/повтор действия. Централизация состояния обеспечивает предсказуемость и упрощает отладку, предоставляя единую точку для инспектирования данных приложения. Наличие всего состояния в одном месте позволяет легко отслеживать, как и почему изменяются данные, что особенно полезно в больших и сложных приложениях.

2 Состояние доступно только для чтения: Единственный способ изменить состояние – это отправить действие (action), объект, описывающий произошедшее событие. Это гарантирует, что ни представления, ни сетевые запросы не могут напрямую изменять состояние. Вместо этого они выражают намерение преобразовать состояние. Такой подход обеспечивает предсказуемость изменений, поскольку все они централизованы и происходят в строгом порядке, что исключает возможность возникновения гонок состояний. Действия являются простыми объектами JavaScript, что позволяет их логировать, сериализовать и воспроизводить для отладки и тестирования.

3 Изменения производятся чистыми функциями (редьюсерами): Чтобы определить, как дерево состояния преобразуется действиями, используются чистые функции, называемые редьюсерами. Редьюсеры принимают предыдущее состояние и действие, а затем возвращают новое состояние. Важно помнить, что редьюсеры должны возвращать новые объекты состояния, а не мутировать существующее состояние. Использование чистых функций для обновления состояния гарантирует, что для заданного состояния и действия результат всегда будет одинаковым, что повышает предсказуемость и облегчает тестирование.

Ключевыми концепциями Redux являются:

1 Стор (Store). Это контейнер, который хранит состояние приложения. Redux предусматривает наличие только одного стора во всем приложении. Концепция единого стора упрощает архитектуру, предоставляя унифицированную точку доступа к состоянию приложения. Наличие одного централизованного хранилища данных облегчает обмен информацией между различными частями приложения без необходимости сложного проброса пропсов.

2 Действия (Actions). Это обычные объекты JavaScript, которые описывают намерение изменить состояние. Каждый объект действия должен иметь свойство type, указывающее на тип действия. Действия служат четкими и сериализуемыми описаниями событий, которые произошли или произойдут в приложении, что облегчает отладку и потенциально позволяет реализовать такие функции, как повторное воспроизведение действий. Представляя изменения состояния в виде простых объектов, Redux упрощает логирование, инспектирование и даже повторное воспроизведение действий, что является неоценимым при отладке и понимании поведения приложения с течением времени.

3 Редьюсеры (Reducers). Это чистые функции, которые принимают текущее состояние и действие, а затем возвращают новое состояние. Редьюсеры не должны мутировать существующее состояние. Требование неизменяемости в редьюсерах гарантирует, что изменения состояния являются предсказуемыми и помогает в реализации таких функций, как отладка с возможностью «путешествия во времени». Всегда возвращая новый объект состояния вместо изменения существующего, Redux поддерживает четкую историю изменений состояния, что важно для отладки и реализации функций отмены/повтора.

4 Диспетчер (Dispatch). Это функция, доступная в сторе, которая используется для отправки действий в редьюсеры. Механизм диспетчеризации обеспечивает контролируемый способ запуска обновлений состояния, гарантируя, что все изменения проходят через определенные редьюсеры. Централизуя обновления состояния через функцию dispatch, Redux обеспечивает строгий однонаправленный поток данных, что облегчает отслеживание того, как и почему изменяется состояние приложения [18].

Использование Redux для управления состоянием дает ряд преимуществ: централизованное управление состоянием, предсказуемые изменения состояния, улучшенные инструменты отладки (Redux DevTools), упрощенный поток данных, улучшенная поддержка кода, лучшая организация работы в команде, совместимость с экосистемой React, возможность отладки с “путешествием во времени” и поддержка сохранения состояния. Эти преимущества в совокупности способствуют созданию более надежных, поддерживаемых и масштабируемых приложений, особенно при работе со сложной логикой состояния.

Рассмотрение использования Redux в React-проекте целесообразно, когда приложение имеет сложное состояние или большой объем данных для отслеживания, когда несколько компонентов должны совместно использовать одно и то же состояние, когда требуется централизованная система управления состоянием, когда приложение требует реактивного состояния, а также для обеспечения долгосрочной поддержки и эффективной работы в команде. Redux наиболее ценен в приложениях со значительным глобальным состоянием, которое необходимо совместно использовать и предсказуемо управлять им во многих компонентах. Для небольших приложений с ограниченным состоянием или простой иерархией компонентов накладные расходы Redux могут быть неоправданными. Однако по мере роста сложности приложений Redux предоставляет необходимую структуру и инструменты для эффективного управления состоянием [19].

В контексте Redux побочные эффекты представляют собой операции, которые взаимодействуют с внешним миром, такие как асинхронные операции, вызовы API и т.д. Редьюсеры, как упоминалось ранее, должны быть чистыми функциями и не должны содержать логику, взаимодействующую с внешним миром (вызовы API, таймеры и т.д.). Примеры побочных эффектов включают получение данных, доступ к кешу браузера, взаимодействие с API, установку таймеров и многое другое.

Обработка побочных эффектов непосредственно в редьюсерах или компонентах имеет ряд ограничений. Во-первых, это нарушает принцип чистых редьюсеров, что делает изменения состояния непредсказуемыми и затрудняет тестирование. Во-вторых, компоненты перегружаются асинхронной логикой, что усложняет их поддержку и повторное использование. Обработка побочных эффектов в основном потоке Redux или непосредственно в компонентах нарушает разделение ответственности и снижает предсказуемость и тестируемость приложения. Redux силен своей предсказуемостью управления состоянием через чистые редьюсеры. Внесение побочных эффектов непосредственно в этот поток подрывает эти преимущества и приводит к более сложному и трудноотлаживаемому коду [20].

### **4.5.2 Redux Saga**

Redux Saga представляет собой библиотеку-посредник (middleware) для Redux, предназначенную для управления побочными эффектами. Ее философия заключается в том, чтобы сделать побочные эффекты более управляемыми, эффективными в исполнении, легкими в тестировании и устойчивыми к ошибкам. Redux Saga использует возможности ES6 Generators, что делает асинхронные потоки более читабельными, простыми в написании и тестировании. Библиотека позволяет сохранять чистоту действий (actions) и вдохновлена принципами функционального реактивного программирования.

Концептуальная модель Redux Saga представляет саги как отдельные потоки в вашем приложении. Эти саги работают в фоновом режиме и несут исключительную ответственность за побочные эффекты. Их можно запускать, приостанавливать и отменять из основного приложения с помощью обычных действий Redux. Саги имеют доступ ко всему состоянию приложения Redux и могут отправлять действия Redux. Аналогия «сага как отдельный поток» помогает понять, как побочные эффекты управляются независимо от основного потока Redux, что приводит к лучшей организации и контролю асинхронных операций. Представляя саги как фоновые процессы, разработчики могут проектировать сложные асинхронные рабочие процессы, не блокируя основной поток UI и не загромождая логику компонентов.

Использование Redux Saga дает ряд преимуществ: упрощает обработку сложного асинхронного кода, обеспечивает лучший контроль над вызовами API (отмена, устранение дребезжания, повторные попытки), улучшает тестируемость за счет изоляции логики в генераторах, делает компоненты более чистыми за счет отделения асинхронной логики, хорошо масштабируется для больших приложений, предлагает более декларативный подход к управлению побочными эффектами и обладает лучшими возможностями обработки ошибок. Эти преимущества делают Redux Saga мощным инструментом для управления сложными асинхронными сценариями структурированным, тестируемым и поддерживаемым способом. Предоставляя выделенный механизм для обработки побочных эффектов, Redux Saga устраняет ограничения базового Redux и предлагает более надежное решение для реальных приложений со сложными асинхронными требованиями [21].

Основные концепции Redux Saga:

1 Саги. Саги реализованы как функции-генераторы (функции с символом \*). Для выражения логики саги из генератора возвращаются обычные объекты JavaScript, называемые эффектами. Саги выступают в роли оркестраторов побочных эффектов, реагируя на отправленные действия и инициируя другие действия или асинхронные операции. Использование функций-генераторов позволяет сагам приостанавливать и возобновлять выполнение, что делает асинхронный код похожим на синхронный и облегчает его управление. Эта особенность упрощает ментальную модель работы с асинхронными операциями, поскольку разработчики могут писать последовательно выглядящий код, обрабатывающий неблокирующие операции.

2 Эффекты. Эффекты – это декларативные объекты, содержащие информацию, которая интерпретируется посредником. Они создаются с помощью функций из пакета redux-saga/effects.

3 Каналы. Введение в управление внешними событиями. Каналы предоставляют способ взаимодействия саг с источниками событий вне обычного потока действий Redux, такими как WebSockets или другие внешние API. Это позволяет сагам структурированно управлять сложными потоками данных в реальном времени и другими сценариями, управляемыми событиями.

4 Наблюдатели. Наблюдатели – это саги, ответственные за прослушивание определенных действий, отправленных в стор Redux. Они обычно используют такие эффекты, как takeEvery, takeLatest или take для прослушивания действий. Часто наблюдатели fork или call рабочие саги для обработки фактической логики побочных эффектов. Наблюдатели обеспечивают четкое разделение ответственности, делегируя фактическую обработку побочных эффектов рабочим сагам, что делает код более организованным и понятным. Этот паттерн помогает управлять сложными асинхронными рабочими процессами, имея выделенные саги, ответственные за запуск и обработку определенных типов действий.

5 Рабочие саги. Рабочие саги – это функции-генераторы, содержащие фактическую логику побочных эффектов. Они выполняют такие задачи, как вызовы API, манипуляции с данными и отправка других действий. Рабочие саги обычно вызываются сагами-наблюдателями. Рабочие саги инкапсулируют конкретную логику для каждого побочного эффекта, что облегчает их тестирование и поддержку в изоляции. Разделяя прослушивание действий и фактическое выполнение задачи, рабочие саги способствуют повторному использованию кода и упрощают управление различными типами побочных эффектов в приложении [22].

Redux предоставляет предсказуемое и централизованное управление состоянием в сложных React-приложениях. Redux Saga, в свою очередь, предлагает элегантное и тестируемое решение для обработки асинхронных операций и побочных эффектов. Разработчикам следует рассматривать использование Redux, когда приложение имеет значительное глобальное состояние, которое необходимо совместно использовать и предсказуемо управлять им. Redux Saga является предпочтительным выбором при работе со сложной асинхронной логикой, требующей контроля, тестируемости и поддерживаемости.

В конечном счете, Redux и Redux Saga, при правильном использовании, представляют собой мощную комбинацию для создания масштабируемых и поддерживаемых React-приложений со сложным управлением состоянием и асинхронными требованиями. Разработчикам необходимо тщательно оценивать потребности своего проекта и выбирать подходящие инструменты в зависимости от сложности приложения и опыта команды.

# **5 Backend**

## **5.1 C#**

### **5.1.1 История создания и развития языка**

Разработка языка программирования C# началась в период с 1998 по 2001 год под руководством группы инженеров Microsoft, возглавляемой Андерсом Хейлсбергом и Скоттом Вильтаумотом. Этот язык был задуман как основной инструмент для разработки приложений на платформе Microsoft.NET. Первая версия C# была официально выпущена в феврале 2002 года вместе с появлением среды разработки Microsoft Visual Studio.NET. C# принадлежит к семейству C-подобных языков, что отражается в его синтаксисе, который имеет много общего с C++ и Java. Название “C#” имеет символическое значение, указывая на эволюцию от языков C и C++ [23].

Одним из побудительных мотивов для создания C# компанией Microsoft стало стремление упростить разработку приложений для операционной системы Windows, а также, вероятно, предложить альтернативу популярному в то время языку Java. Со временем язык C# был стандартизирован организациями ECMA (как ECMA-334) и ISO/IEC (как ISO/IEC 23270). На текущий момент последней версией языка, упомянутой в доступных материалах, является C# 13, релиз которого состоялся 12 ноября 2024 года одновременно с выходом платформы.NET 9. Важно отметить, что развитие C# тесно связано с эволюцией платформы .NET, начиная с .NET Framework и переходя к .NET Core, который впоследствии стал просто .NET.

Разработка C# в конце 1990-х и начале 2000-х годов совпала с периодом активного развития интернета и возникновением потребности в современных языках, ориентированных на платформы. Это позволяет предположить, что C# был разработан с учетом задач создания распределенных и веб-приложений. Двойная мотивация Microsoft при создании C# – упрощение разработки под Windows и создание конкуренции Java – демонстрирует стратегический шаг компании по укреплению своей экосистемы и соперничеству с Sun Microsystems. Постоянное развитие C#, о чем свидетельствуют новые версии и функциональные возможности, такие как представленные в C# 9-13, подчеркивает непрерывные инвестиции Microsoft в этот язык и стремление поддерживать его актуальность в соответствии с последними тенденциями в программировании и потребностями индустрии. Таким образом, C# обладает богатой историей, уходящей корнями в стратегию Microsoft.NET, черпая вдохновение из зарекомендовавших себя языков и одновременно нацеливаясь на решение современных задач разработки. Его стандартизация и непрерывное развитие являются ключевыми аспектами его устойчивой значимости [24].

### **5.1.2 Основные характеристики и особенности C#**

**СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ**

[1] Platform of Growth: An Analysis of Earnings and Social Media Engagement Among Esports Athletes [Электронный ресурс]. – Режим доступа: https://scholarcommons.sc.edu/cgi/viewcontent.cgi?article=8321&context=etd. – Дата доступа: 29.04.2025

[2] How Data Analytics is Shaping Competitive Esports [Электронный ресурс]. – Режим доступа: https://pley.gg/how-data-analytics-is-shaping-competitive-esports/. – Дата доступа: 29.04.2025

[3] History of Video Games [Электронный ресурс]. – Режим доступа: https://www.ebsco.com/research-starters/history/history-video-games. – Дата доступа: 29.04.2025

[4] The Gaming Industry in 2024 by the Numbers – A Review by GamesIndustry [Электронный ресурс]. – Режим доступа: https://app2top.com/news/the-gaming-industry-in-2024-by-the-numbers-a-review-by-gamesindustry-276003.html. – Дата доступа: 29.04.2025

[5] The Evolution of Dota 2 TI and Its Impact on Gameplay [Электронный ресурс]. – Режим доступа: https://gameflip.com/blog/the-evolution-of-dota-2-ti-and-its-impact-on-gameplay. – Дата доступа: 29.04.2025

[6] JavaScript. [Электронный ресурс]. – Режим доступа: https://blog.skillfactory.ru/glossary/javascript/. – Дата доступа: 29.04.2025

[7] Краткая история JavaScript. Часть 1. [Электронный ресурс]. – Режим доступа: https://habr.com/ru/companies/livetyping/articles/324196/. – Дата доступа: 29.04.2025

[8] Краткая история JavaScript. Часть 2. [Электронный ресурс]. – Режим доступа: https://habr.com/ru/companies/livetyping/articles/324506/. – Дата доступа: 29.04.2025

[9] Что такое и для чего нужен TypeScript. [Электронный ресурс]. – Режим доступа: https://nauchikus.gitlab.io/typescript-definitive-guide/book/contents/Общее – Что такое и для чего нужен TypeScript.html. – Дата доступа: 29.04.2025

[10] Project Philosophy. [Электронный ресурс]. – Режим доступа: https://vite.dev/guide/philosophy.html – Дата доступа: 29.04.2025

[11] Начало работы. [Электронный ресурс]. – Режим доступа: https://react.dev/ – Дата доступа: 29.04.2025

[12] React Data Flow Simplified: Manage State Like A Pro. [Электронный ресурс]. – Режим доступа: https://www.dhiwise.com/blog/design-converter/react-data-flow-simplified-manage-state-like-a-pro – Дата доступа: 29.04.2025

[13] Why use React for web development in 2025: benefits, cases, top examples. [Электронный ресурс]. – Режим доступа: https://solveit.dev/blog/why-use-react-for-web-development – Дата доступа: 29.04.2025

[14] Why Use React? Top Benefits for Web Development. [Электронный ресурс]. – Режим доступа: https://www.netguru.com/blog/why-use-react – Дата доступа: 29.04.2025

[15] Benefits of Using ReactJS For Custom Web Development. [Электронный ресурс]. – Режим доступа: https://www.geeksforgeeks.org/benefits-of-using-reactjs-for-custom-web-development/ – Дата доступа: 29.04.2025

[16] Знакомство с React JS. [Электронный ресурс]. – Режим доступа: https://purpleschool.ru/blog/ReactJS – Дата доступа: 29.04.2025

[17] React: освоение фреймворка JavaScript. [Электронный ресурс]. – Режим доступа: https://open.zeba.academy/react-freymvork-javascript/ – Дата доступа: 29.04.2025

[18] Three Principles. [Электронный ресурс]. – Режим доступа: https://redux.js.org/understanding/thinking-in-redux/three-principles – Дата доступа: 29.04.2025

[19] What Is Redux? [Электронный ресурс]. – Режим доступа: https://www.koombea.com/blog/what-is-redux/ – Дата доступа: 29.04.2025

[20] The Complete Guide to Redux Saga With Real life example [Электронный ресурс]. – Режим доступа: https://draken.hashnode.dev/the-complete-guide-to-redux-saga-with-real-life-example – Дата доступа: 29.04.2025

[21] About Redux-Saga [Электронный ресурс]. – Режим доступа: https://redux-saga.js.org/docs/About/ – Дата доступа: 29.04.2025

[22] What is Redux Saga? [Электронный ресурс]. – Режим доступа: https://www.geeksforgeeks.org/what-is-redux-saga/ – Дата доступа: 29.04.2025

[23] Язык программирования C#: краткая история, возможности и перспективы [Электронный ресурс]. – Режим доступа: https://timeweb.com/ru/community/articles/chto-takoe-csharp – Дата доступа: 29.04.2025

[24] Язык С#: где используют, что пишут, как появился и чем хорош [Электронный ресурс]. – Режим доступа: https://skillbox.ru/media/code/yazyk-s-gde-ispolzuyut-chto-pishut-kak-poyavilsya-i-chem-khorosh/ – Дата доступа: 29.04.2025

[20] What is Redux Saga? [Электронный ресурс]. – Режим доступа: https://www.geeksforgeeks.org/what-is-redux-saga/ – Дата доступа: 29.04.2025