МИНИСТЕРСТВО ПРОСВЕЩЕНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ ПЕДАГОГИЧЕСКИЙ УНИВЕРСИТЕТ им. А. И. ГЕРЦЕНА»

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И ТЕХНОЛОГИЧЕСКОГО ОБРАЗОВАНИЯ
Кафедра информационных технологий и электронного обучения

Основная профессиональная образовательная программа
Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль) «Технологии разработки программного обеспечения»
форма обучения – очная

АНАЛИТИЧЕСКИЙ ОБЗОР

по теме: Web-технологии (Web service design)

Обучающегося 4 курса
Красникова Даниила Ярославича

Научный руководитель:
кандидат физико-математических наук,
доцент кафедры ИТиЭО
Жуков Николай Николаевич

Санкт-Петербург
2025

1. ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В современном мире информационных технологий веб-сервисы представляют собой фундаментальную основу для построения распределенных систем и обеспечения взаимодействия между различными программными компонентами. Web service design – проектирование веб-сервисов – является критически важной областью, определяющей эффективность, надежность и масштабируемость современных информационных систем.

Актуальность темы обусловлена несколькими ключевыми факторами. Во-первых, стремительное развитие цифровой экономики требует создания надежных механизмов обмена данными между системами различных организаций. Большинство современных приложений используют API для интеграции с внешними сервисами. Во-вторых, распространение микросервисной архитектуры делает понимание принципов проектирования веб-сервисов необходимым условием для создания современных программных решений. В-третьих, растущие требования к безопасности данных и производительности систем требуют глубокого понимания архитектурных подходов и технологических решений.

Цель данного аналитического обзора – систематизация знаний в области проектирования веб-сервисов, анализ существующих архитектурных подходов, технологических решений и выявление современных тенденций развития.

Для достижения поставленной цели необходимо решить следующие задачи:

1. изучить теоретические основы и ключевые понятия веб-сервисов;

2. проанализировать архитектурные принципы построения веб-сервисов, в особенности REST-архитектуру;
3. рассмотреть технологический стек и протоколы передачи данных;
4. исследовать подходы к проектированию API и обработке ошибок;
5. изучить методы обеспечения безопасности веб-сервисов;
6. рассмотреть современные практики документирования и тестирования;
7. выявить актуальные тенденции развития технологий веб-сервисов.

Практическая значимость работы заключается в том, что систематизированные знания могут быть использованы при проектировании новых веб-сервисов, при модернизации существующих систем, а также в образовательном процессе для подготовки специалистов в области информационных технологий. Представленный анализ архитектурных подходов, технологий и практик позволяет принимать обоснованные решения при выборе инструментов и методов разработки веб-сервисов.

Структура работы построена таким образом, чтобы последовательно раскрыть все ключевые аспекты проектирования веб-сервисов. В первой главе рассматриваются теоретические основы веб-сервисов, ключевые понятия и типы веб-сервисов. Вторая глава посвящена детальному анализу REST-архитектуры и её основополагающих принципов. Третья глава описывает технологический стек, включая протоколы и форматы обмена данными. Четвертая глава рассматривает практические аспекты проектирования API. Пятая глава посвящена вопросам безопасности веб-сервисов. Шестая глава анализирует современные подходы к разработке, включая микросервисную архитектуру и контейнеризацию.

1 ТЕОРЕТИЧЕСКИЕ ОСНОВЫ ВЕБ-СЕРВИСОВ

Веб-сервис представляет собой программную систему, которая предоставляет функции или данные через интернет по стандартизированным протоколам для взаимодействия между другими программами, независимо от их платформы или языка, например, API для получения погоды, платежные шлюзы или сервисы онлайн-банкинга, автоматизирующие бизнес-процессы и позволяющие интегрировать разные системы. Согласно определению консорциума W3C, веб-сервис – это программное обеспечение, идентифицируемое с помощью URI (Uniform Resource Identifier), интерфейсы и связи которого определяются, описываются и обнаруживаются посредством XML-артефактов. Данное определение подчеркивает ключевые характеристики веб-сервисов: они являются самодостаточными программными модулями, доступными через стандартные интернет-протоколы, способными взаимодействовать независимо от платформы и языка программирования.

Фундаментальные характеристики веб-сервисов определяют их роль в современной архитектуре программных систем. Слабая связанность компонентов позволяет им развиваться независимо друг от друга, что критически важно в распределенных системах с множеством участников разработки. Независимость от платформы и языка программирования обеспечивает универсальность решений – веб-сервис, написанный на Java, может взаимодействовать с клиентом на Python или JavaScript без необходимости адаптации. Использование стандартных протоколов передачи данных, в первую очередь HTTP и HTTPS, гарантирует совместимость различных систем и снижает барьеры для интеграции. Возможность автоматического обнаружения и динамической интеграции через механизмы service discovery упрощает построение сложных распределенных систем.

В современной практике проектирования веб-сервисов выделяют два основных архитектурных подхода, каждый из которых имеет свои преимущества и области применения. SOAP (Simple Object Access Protocol) веб-сервисы представляют формальный подход, основанный на строгих стандартах. SOAP использует XML для структурирования сообщений и определяет формат конвертов для передачи данных. Этот подход характеризуется строгой типизацией данных, что предотвращает множество ошибок на этапе разработки; наличием формальных контрактов в виде WSDL (Web Services Description Language), описывающих все аспекты веб-сервиса; поддержкой сложных транзакций и стандартов безопасности на уровне сообщений через WS-Security. SOAP-сервисы широко применяются в корпоративных системах, особенно в финансовом секторе и государственных учреждениях, где критически важны надежность, безопасность, формальная верификация взаимодействия и поддержка устаревших систем.

RESTful (Representational State Transfer) веб-сервисы представляют альтернативный архитектурный стиль, основанный на принципах, сформулированных Роем Филдингом в его докторской диссертации в 2000 году. REST не является протоколом или стандартом, а представляет собой набор архитектурных ограничений, которые при совместном применении определяют характеристики системы. REST-сервисы используют стандартные HTTP-методы (GET, POST, PUT, DELETE, PATCH) для манипулирования ресурсами, что делает API интуитивно понятным для разработчиков, знакомых с веб-технологиями. Они не сохраняют состояние клиента между запросами (stateless), что существенно упрощает масштабирование и повышает надежность системы. REST-сервисы поддерживают механизмы кэширования через стандартные HTTP-заголовки, что может значительно снизить нагрузку на сервер и улучшить производительность. Они обеспечивают единообразный интерфейс взаимодействия через концепцию ресурсов и их представлений. Благодаря относительной простоте, гибкости и

эффективности, REST стал доминирующим подходом в разработке современных веб-API, особенно в области мобильных приложений, одностраничных веб-приложений и публичных API.

Выбор между SOAP и REST зависит от конкретных требований проекта и контекста использования. SOAP предпочтителен в сценариях, требующих формальных контрактов и строгой типизации, что обеспечивает дополнительный уровень безопасности на этапе разработки; поддержки ACID-транзакций для обеспечения согласованности данных в распределенных системах; строгой безопасности на уровне сообщений через WS-Security, что важно для финансовых операций; интеграции с устаревшими корпоративными системами, многие из которых построены на SOAP. REST более подходит для современных веб-приложений, где важны скорость разработки и гибкость; мобильных платформ, где критичны размер сообщений и эффективность использования батареи; публичных API, где простота использования и понятность документации являются ключевыми факторами; систем с высокими требованиями к производительности и масштабируемости.

2 АРХИТЕКТУРНЫЕ ПРИНЦИПЫ REST

REST представляет собой набор архитектурных ограничений, которые при совместном применении определяют характеристики и поведение распределенной гипермедиа-системы. Важно понимать, что эти ограничения не являются технологической спецификацией или протоколом, а представляют архитектурный стиль – способ мышления о системе, который может быть реализован различными способами с использованием различных технологий.

Клиент-серверная архитектура является первым и наиболее фундаментальным ограничением REST. Это ограничение требует четкого разделения ответственности между пользовательским интерфейсом (клиент) и хранением данных (сервер). Клиент отвечает за представление данных пользователю, сбор пользовательского ввода и управление пользовательской сессией на стороне клиента. Сервер управляет хранением данных, реализует бизнес-логику, обеспечивает согласованность и целостность данных. Такое разделение обеспечивает множество преимуществ: компоненты системы могут эволюционировать независимо друг от друга, что критически важно в современной разработке с множеством команд; упрощается масштабирование за счет возможности репликации серверов; обеспечивается гибкость в выборе клиентских платформ – один сервер может обслуживать веб-клиенты, мобильные приложения и другие системы; снижается сложность серверного кода за счет отсутствия необходимости управлять пользовательским интерфейсом.

Отсутствие состояния (stateless) является одним из наиболее важных и часто неправильно понимаемых ограничений REST. Это ограничение требует, чтобы каждый запрос от клиента к серверу содержал всю информацию, необходимую для понимания и обработки этого запроса. Сервер не должен хранить контекст клиентской сессии между запросами. Вся информация о

состоянии сессии хранится на клиенте. Это ограничение имеет далеко идущие последствия для архитектуры системы. Stateless подход существенно упрощает масштабирование системы, так как любой сервер в кластере может обработать любой запрос без необходимости обращения к общему хранилищу состояний или синхронизации между серверами. Повышается надежность системы, так как сбой одного сервера не приводит к потере состояния пользовательских сессий. Упрощается балансировка нагрузки, так как нет необходимости в sticky sessions. Улучшается видимость системы для мониторинга и отладки, так как каждый запрос самодостаточен. Однако stateless подход имеет и недостатки: увеличивается размер сетевого трафика, так как информация о состоянии передается в каждом запросе; может снизиться производительность для некоторых типов взаимодействия, где передача полного контекста неэффективна; усложняется клиентский код, так как клиент должен управлять состоянием.

Кэшируемость является критически важным принципом для производительности веб-систем. REST требует, чтобы ответы сервера явно или неявно помечались как кэшируемые или некэшируемые. Если ответ кэшируем, клиент или промежуточный прокси получают право повторно использовать данные ответа для эквивалентных запросов в будущем. HTTP предоставляет богатый набор механизмов для управления кэшированием. Заголовок Cache-Control определяет директивы кэширования, такие как max-age, no-cache, no-store, public, private. Заголовок ETag (entity tag) предоставляет механизм условных запросов для проверки, изменился ли ресурс. Заголовок Last-Modified указывает время последней модификации ресурса. Заголовок Expires определяет абсолютное время истечения кэша. Эффективное использование кэширования может радикально улучшить производительность системы: снижается нагрузка на серверы за счет обработки меньшего количества запросов; уменьшается задержка для клиентов, так как данные могут быть получены из локального кэша или кэша CDN; снижается

потребление сетевой пропускной способности; улучшается масштабируемость системы. Однако кэширование требует тщательного проектирования стратегий инвалидации и обновления данных для предотвращения использования устаревшей информации.

Единообразный интерфейс является центральной особенностью REST-архитектуры, отличающей её от других сетевых архитектурных стилей. Это ограничение упрощает общую архитектуру системы и улучшает видимость взаимодействий, но требует компромисса – интерфейс может быть менее эффективным для конкретных нужд приложения, чем специализированный интерфейс. Единообразие достигается через четыре дополнительных ограничения. Идентификация ресурсов требует, чтобы каждый ресурс был идентифицируем через стабильный URI. Манипулирование ресурсами через представления означает, что клиенты взаимодействуют с представлениями ресурсов, а не с самими ресурсами напрямую. Самоописываемые сообщения требуют, чтобы каждое сообщение содержало достаточно информации для описания того, как его обрабатывать. HATEOAS (Hypermedia As The Engine Of Application State) требует, чтобы клиенты взаимодействовали с приложением исключительно через гипермедиа, предоставляемую сервером.

Концепция ресурсов лежит в основе REST-архитектуры и требует детального понимания. Ресурс – это любая информация, которая может быть именована: документ, изображение, временная служба (например, прогноз погоды), коллекция других ресурсов, нефизический объект (например, бизнес-процесс или концепция). Ресурсы идентифицируются с помощью URI, которые обеспечивают глобальное пространство имен. Критически важно понимать различие между ресурсом и его представлением. Ресурс – это абстрактная концепция, представление – это конкретная сериализация состояния ресурса в определенный момент времени. Один ресурс может иметь множество представлений в различных форматах: JSON для веб и мобильных приложений, XML для корпоративных систем, HTML для браузеров,

изображения различных форматов и разрешений. Клиент запрашивает предпочтительное представление через механизм согласования содержимого (content negotiation), используя заголовки Accept и Accept-Language.

3 ТЕХНОЛОГИЧЕСКИЙ СТЕК И ПРОТОКОЛЫ

HTTP (Hypertext Transfer Protocol) является фундаментальным протоколом прикладного уровня для веб-сервисов. Протокол определяет структуру сообщений, методы запросов и коды состояния ответов, обеспечивая стандартизированное взаимодействие между клиентами и серверами. HTTP развивался на протяжении десятилетий, и каждая новая версия вносила значительные улучшения в производительность и функциональность.

HTTP/1.1, стандартизированный в RFC 2616 в 1999 году и обновленный в RFC 7230-7235 в 2014 году, ввел несколько ключевых улучшений. Постоянные соединения (persistent connections) позволяют повторно использовать TCP-соединение для множественных HTTP-запросов, устраняя накладные расходы на установление нового соединения для каждого запроса. Конвейерная обработка (HTTP pipelining) позволяет отправлять несколько запросов без ожидания ответов на предыдущие, хотя эта функция имеет ограниченную поддержку. Улучшенное кэширование через расширенные заголовки Cache-Control обеспечивает более гибкое управление кэшированием. Поддержка виртуальных хостов через заголовок Host позволяет одному IP-адресу обслуживать множество доменов. Частичные запросы через заголовки Range позволяют загружать части файла, что критично для потокового видео и возобновляемых загрузок.

HTTP/2, стандартизированный в RFC 7540 в 2015 году, представил радикальные изменения в способе передачи данных. Мультиплексирование позволяет отправлять множественные запросы и ответы одновременно по одному TCP-соединению, устраняя проблему блокировки начала строки (head-of-line blocking) на уровне приложения. Это особенно заметно улучшает производительность для веб-страниц с множеством ресурсов. Сжатие заголовков с использованием специализированного алгоритма HPACK

уменьшает накладные расходы, особенно заметные при частых запросах с похожими заголовками. В типичных веб-приложениях заголовки могут составлять значительную часть трафика. Серверная push-отправка (server push) позволяет серверу проактивно отправлять ресурсы клиенту до их явного запроса. Например, при запросе HTML-страницы сервер может немедленно отправить связанные CSS и JavaScript файлы, не дожидаясь их запроса браузером. Приоритизация потоков позволяет клиенту указывать относительную важность различных запросов, помогая серверу оптимизировать использование пропускной способности. Бинарный протокол вместо текстового делает парсинг более эффективным и менее подверженным ошибкам.

HTTP/3, основанный на протоколе QUIC (Quick UDP Internet Connections), представляет следующий этап эволюции и решает фундаментальные ограничения TCP. Переход от TCP к UDP на транспортном уровне позволяет избежать блокировки начала строки на транспортном уровне, что было узким местом в HTTP/2. Встроенное шифрование TLS 1.3 обеспечивает более быструю установку безопасного соединения — в некоторых случаях данные могут быть отправлены уже в первом пакете (0-RTT). Улучшенная обработка потери пакетов позволяет продолжать передачу других потоков при потере пакета одного потока. Быстрая миграция соединений позволяет сохранять соединение при смене сети (например, при переключении с Wi-Fi на мобильную сеть), что особенно важно для мобильных приложений. Встроенная коррекция ошибок и контроль перегрузок оптимизированы для современных сетевых условий.

HTTPS обеспечивает безопасную передачу данных через шифрование с использованием протоколов TLS. TLS обеспечивает три критически важных свойства безопасности. Конфиденциальность достигается через симметричное шифрование данных после установки соединения, используя алгоритмы вроде AES-256-GCM. Аутентичность обеспечивается через цифровые сертификаты

X.509, выданные доверенными центрами сертификации, что позволяет клиентам верифицировать идентичность сервера. Целостность гарантируется через криптографические хеш-функции, предотвращающие незаметное изменение данных в процессе передачи. Современные стандарты безопасности требуют использования TLS 1.2 или выше, с предпочтением TLS 1.3, который предлагает улучшенную производительность через упрощенное рукопожатие и безопасность через удаление устаревших алгоритмов. Важно использовать надежные криптографические алгоритмы, поддерживать forward secrecy для защиты прошлых сессий даже при компрометации долгосрочных ключей, правильно настраивать сертификаты с достаточной длиной ключа и реализовывать HSTS для принудительного использования HTTPS.

Форматы обмена данными играют критическую роль в веб-сервисах, определяя, как информация структурируется и передается между системами. JSON (JavaScript Object Notation) стал де-факто стандартом для RESTful сервисов. Преимущества JSON включают компактность по сравнению с XML, читаемость для человека, что упрощает отладку, нативную поддержку в JavaScript и большинстве современных языков программирования, простоту парсинга и генерации. JSON поддерживает ограниченный набор типов данных: строки, числа, логические значения (true/false), null, объекты (коллекции пар ключ-значение), массивы (упорядоченные списки значений). Эта простота является одновременно преимуществом и ограничением. Отсутствие встроенной поддержки дат требует использования строковых представлений (обычно ISO 8601), отсутствие типа для бинарных данных требует кодирования в Base64, отсутствие комментариев может затруднить документирование в самом JSON.

XML остается важным форматом, особенно в корпоративных системах и SOAP-сервисах. Преимущества XML включают строгую схему валидации через XSD, обеспечивающую проверку структуры и типов данных на этапе интеграции, поддержку пространств имен для избежания конфликтов имен

при объединении схем из разных источников, расширяемость через возможность добавления новых элементов и атрибутов без нарушения обратной совместимости, поддержку сложных иерархических структур данных. XML также предоставляет мощные инструменты для трансформации (XSLT) и запросов (XPath, XQuery). Недостатками являются многословность, приводящая к большему размеру сообщений, более высокие требования к вычислительным ресурсам для парсинга и генерации, сложность для чтения человеком по сравнению с JSON.

4 ПРОЕКТИРОВАНИЕ И БЕЗОПАСНОСТЬ API

Эффективное проектирование API требует соблюдения установленных практик и принципов. Ресурсо-ориентированный подход составляет основу REST API и предполагает использование существительных во множественном числе для именования endpoints. Вместо глаголов действия (createUser, getUserById) используются пути к ресурсам: POST /users для создания, GET /users/{id} для получения конкретного пользователя, GET /users для получения коллекции, PUT /users/{id} для полного обновления, PATCH /users/{id} для частичного обновления, DELETE /users/{id} для удаления. Такой подход делает API интуитивно понятным и самодокументируемым. Иерархические URL-адреса выражают отношения между ресурсами: GET /users/{userId}/orders получает заказы конкретного пользователя, POST /orders/{orderId}/items добавляет элемент к заказу, GET /organizations/{orgId}/departments/{deptId}/employees получает сотрудников конкретного отдела. Вложенность должна быть ограничена двумя-тремя уровнями для избежания излишней сложности.

HTTP-методы семантически отображают операции над ресурсами. GET используется для чтения ресурсов и должен быть безопасным (не изменяет состояние сервера) и идемпотентным (множественные идентичные запросы имеют тот же эффект, что и один запрос). POST создает новые ресурсы или выполняет операции, которые не подходят под другие методы. POST не идемпотентен – два идентичных POST-запроса создадут два ресурса. PUT используется для полного обновления существующих ресурсов и является идемпотентным. PATCH применяется для частичного обновления, отправляя только измененные поля. DELETE удаляет ресурсы и также идемпотентен. Идемпотентность критична для надежности распределенных систем, так как позволяет безопасно повторять запросы при сетевых сбоях без риска дублирования эффектов.

Правильное использование HTTP-кодов состояния является важнейшим аспектом проектирования API. Коды 2xx обозначают успешные операции: 200 OK для успешного GET или PUT возвращает ресурс в теле ответа, 201 Created для успешного POST возвращает созданный ресурс и указывает URI в заголовке Location, 204 No Content для успешного DELETE или PUT когда нет данных для возврата. Коды 4xx обозначают ошибки клиента: 400 Bad Request для невалидного синтаксиса запроса или нарушения бизнес-правил, 401 Unauthorized для отсутствующих или невалидных учетных данных аутентификации, 403 Forbidden когда сервер понял запрос но отказывается его выполнять из-за недостатка прав, 404 Not Found для несуществующих ресурсов, 409 Conflict при конфликте состояния, 422 Unprocessable Entity для семантически некорректных данных, 429 Too Many Requests при превышении лимита. Коды 5xx обозначают серверные ошибки: 500 Internal Server Error для непредвиденных сбоев, 502 Bad Gateway при проблемах с upstream сервером, 503 Service Unavailable при временной недоступности.

Версионирование API является критически важной практикой для эволюции системы без нарушения работы существующих клиентов. Существует несколько подходов. Версия в URL является наиболее явным и популярным методом: /v1/users, /api/v2/products. Преимущества: простота, очевидность, легкость поддержки разных версий на разных серверах, простота кэширования и маршрутизации. Недостатки: загрязняет URI, нарушает принцип REST о постоянстве идентификаторов. Версия в заголовке: Accept: application/vnd.company.api+json;version=2. Преимущества: не изменяет URI, считается более RESTful. Недостатки: менее очевидно, сложнее тестировать в браузере, могут быть проблемы с кэшированием. Версия через параметр запроса: /users?version=1. Преимущества: проста в реализации. Недостатки: не рекомендуется как основной метод, загрязняет параметры.

Безопасность веб-сервисов охватывает множество аспектов. OAuth 2.0 является стандартом индустрии для делегированной авторизации. Протокол

позволяет приложениям получать ограниченный доступ к ресурсам пользователя без раскрытия учетных данных. OAuth 2.0 определяет несколько grant types. Authorization Code Grant используется для серверных приложений: пользователь перенаправляется на сервер авторизации, после аутентификации возвращается код, который обменивается на токен доступа. Это наиболее безопасный flow. Client Credentials Grant применяется для межсервисного взаимодействия без контекста пользователя. Resource Owner Password Credentials Grant используется для доверенных клиентов, но не рекомендуется для новых приложений из-за рисков безопасности. Refresh Token позволяет получать новые access tokens без повторной аутентификации пользователя.

JWT (JSON Web Tokens) обеспечивает компактный механизм передачи информации. Токен состоит из трех частей, закодированных в Base64URL и разделенных точками: заголовок (header) содержит тип токена и алгоритм подписи, полезная нагрузка (payload) содержит утверждения (claims) – данные о пользователе и метаданные токена, подпись позволяет верифицировать, что токен не был изменен. JWT широко используется в stateless аутентификации: сервер не хранит сессии, вся необходимая информация содержится в токене, что упрощает масштабирование. Однако важно понимать, что JWT по умолчанию только подписан, но не зашифрован – payload можно декодировать, поэтому чувствительная информация не должна храниться в токене. Для конфиденциальных данных следует использовать JWE (JSON Web Encryption). JWT должны иметь короткое время жизни и включать проверки для предотвращения replay attacks.

Защита от атак требует комплексного подхода. Rate limiting предотвращает DoS-атаки путем ограничения количества запросов от одного клиента. Распространенные алгоритмы включают token bucket (наиболее гибкий), leaky bucket и fixed window. При превышении лимита возвращается 429 Too Many Requests с заголовками X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset. CORS контролирует доступ к ресурсам из

различных доменов. Механизм same-origin policy браузера запрещает JavaScript обращаться к данным на другом домене. CORS позволяет серверу явно указать разрешенные домены через Access-Control-Allow-Origin. Для сложных запросов браузер сначала отправляет preflight запрос OPTIONS. Валидация входных данных критична для защиты от injection-атак. Все пользовательские данные должны рассматриваться как потенциально опасные.

5 СОВРЕМЕННЫЕ ПРАКТИКИ И ТЕХНОЛОГИИ

OpenAPI Specification (ранее Swagger) стала индустриальным стандартом для описания RESTful API. Спецификация определяет машиночитаемый формат (YAML или JSON) для полного описания API. Она включает информацию о доступных endpoints и операциях на каждом endpoint, параметрах операций (path, query, header, body), структуре запросов и ответов с примерами, методах аутентификации и схемах безопасности, метаданных API (версия, контакты, лицензия), серверах API и их URL. OpenAPI позволяет автоматически генерировать интерактивную документацию через Swagger UI или ReDoc, клиентские SDK для множества языков программирования, серверные заглушки для быстрого прототипирования, тестовые наборы для валидации API. Это обеспечивает согласованность между документацией и реализацией и значительно ускоряет разработку и интеграцию.

GraphQL представляет альтернативный подход к проектированию API, предложенный Facebook в 2015 году. В отличие от REST, где структура данных определяется сервером, GraphQL позволяет клиентам описывать структуру требуемых данных через декларативный язык запросов. Преимущества включают устранение over-fetching (получение избыточных данных) и under-fetching (необходимость множественных запросов для получения связанных данных), строгую типизацию через схему GraphQL, поддержку introspection для автоматической генерации документации, возможность получения всех нужных данных в одном запросе, эволюцию API без версионирования через механизм deprecation полей. Недостатки включают более сложное кэширование по сравнению с REST, потенциальные проблемы с производительностью при глубоких или множественных запросах, сложность реализации rate limiting, необходимость в специализированных инструментах и библиотеках. GraphQL наиболее полезен для сложных

клиентских приложений с разнообразными требованиями к данным, особенно мобильных приложений, где важна минимизация сетевого трафика.

Тестирование веб-сервисов требует комплексного подхода на нескольких уровнях. Модульные тесты проверяют отдельные функции и методы в изоляции, используя mock-объекты для внешних зависимостей. Они быстры, легко поддерживаются и обеспечивают хорошую основу для рефакторинга. Интеграционные тесты верифицируют взаимодействие между компонентами системы, включая обращения к реальной базе данных и внешним сервисам. Contract testing, особенно актуальный для микросервисной архитектуры, обеспечивает совместимость между поставщиками и потребителями API. Инструменты вроде Pact позволяют потребителю определить ожидания от API (контракт), которые затем автоматически проверяются на стороне поставщика. Это предотвращает breaking changes и упрощает независимое развитие сервисов. End-to-end тесты проверяют систему целиком через пользовательский интерфейс, имитируя реальные пользовательские сценарии. Нагрузочное тестирование оценивает производительность под различными уровнями нагрузки, помогая выявить узкие места, определить пределы масштабируемости и проверить поведение системы при стрессовых условиях.

Микросервисная архитектура представляет подход к построению приложений как набора небольших, независимо развертываемых сервисов. Каждый микросервис инкапсулирует специфическую бизнес-возможность, имеет собственную базу данных (принцип database-per-service) и взаимодействует с другими через легковесные механизмы, обычно HTTP/REST API или системы обмена сообщениями. Преимущества микросервисов включают технологическую гибкость — каждый сервис может использовать наиболее подходящий стек технологий, независимое масштабирование компонентов системы в зависимости от нагрузки, улучшенную отказоустойчивость через изоляцию сбоев, организационные

преимущества – команды могут работать автономно над отдельными сервисами, упрощенное понимание кодовой базы – каждый сервис имеет ограниченную область ответственности. Однако микросервисы вводят значительные сложности: распределенные транзакции требуют паттернов вроде Saga, мониторинг и отладка усложняются из-за распределенной природы системы, увеличивается операционная сложность развертывания и управления множеством сервисов, возникают проблемы с консистентностью данных между сервисами, усложняется тестирование системы целиком. Микросервисы оправданы для крупных, сложных приложений с большими командами разработки, но могут быть излишними для небольших проектов.

Контейнеризация с использованием Docker радикально изменила способ развертывания и эксплуатации веб-сервисов. Контейнеры предоставляют легковесную виртуализацию на уровне операционной системы, инкапсулируя приложение и все его зависимости в изолированную среду выполнения. Преимущества включают консистентность окружений – контейнер работает идентично на машине разработчика, тестовом и продакшн-серверах, устраняя проблему работает у меня, портируемость между различными инфраструктурами и облачными провайдерами, изоляцию – приложения не конфликтуют друг с другом, эффективность – контейнеры используют меньше ресурсов чем виртуальные машины, быстрый старт – контейнеры запускаются за секунды, версионирование через Docker images и Docker registries. Docker Compose позволяет определять и запускать многоконтейнерные приложения через декларативные YAML-файлы.

Kubernetes стал де-факто стандартом оркестрации контейнеров, автоматизируя развертывание, масштабирование и управление контейнеризованными приложениями в кластере машин. Kubernetes обеспечивает автоматическое распределение контейнеров по узлам кластера с учетом ресурсов, self-healing – автоматический перезапуск упавших контейнеров и замена нездоровых узлов, горизонтальное

автомасштабирование на основе метрик загрузки CPU или custom metrics, service discovery и балансировку нагрузки для распределения трафика между подами, автоматические обновления и откаты через rolling updates и rollback механизмы, управление конфигурацией и секретами через ConfigMaps и Secrets, persistent storage через интеграцию с различными системами хранения. Kubernetes имеет крутую кривую обучения, но для крупных микросервисных приложений преимущества оркестрации значительно перевешивают сложность.

Serverless computing и Functions-as-a-Service (FaaS) представляют следующий уровень абстракции инфраструктуры. Платформы вроде AWS Lambda, Azure Functions и Google Cloud Functions позволяют разработчикам деплоить и выполнять код без управления серверами. Код выполняется в ответ на события (HTTP-запросы, изменения в базе данных, сообщения в очереди, расписание), автоматически масштабируется от нуля до тысяч одновременных выполнений в зависимости от нагрузки, оплачивается только за фактическое время выполнения с точностью до миллисекунд. Преимущества включают отсутствие управления серверами – полная абстракция инфраструктуры, автоматическое масштабирование без настройки, оптимальную экономику для переменной нагрузки, быструю разработку и деплой функций. Ограничения включают время выполнения (обычно до 15 минут), холодный старт – задержка при первом вызове функции после периода неактивности, ограничения на размер deployment package, сложность отладки и мониторинга распределенных систем, потенциальный vendor lock-in. Serverless идеален для event-driven архитектур, обработки событий и триггеров, микросервисов и функций с непостоянной нагрузкой, фоновых задач и scheduled jobs.

ЗАКЛЮЧЕНИЕ

Проведенный аналитический обзор позволил систематизировать знания в области проектирования веб-сервисов и достичь поставленной цели исследования. В ходе работы были решены все поставленные задачи: изучены теоретические основы и ключевые понятия веб-сервисов, проанализированы архитектурные принципы REST, рассмотрен технологический стек и протоколы передачи данных, исследованы подходы к проектированию API, изучены методы обеспечения безопасности, рассмотрены современные практики документирования и тестирования, выявлены актуальные тенденции развития технологий веб-сервисов.

Анализ показал, что веб-сервисы прошли значительный путь эволюции от SOAP-систем к RESTful API и современным подходам вроде GraphQL и gRPC. REST-архитектура, основанная на принципах Роя Филдинга, доминирует в современной практике благодаря простоте, масштабируемости и соответствию архитектуре веба. Ключевые ограничения REST — клиент-серверное разделение, отсутствие состояния, кэшируемость и единообразный интерфейс — обеспечивают масштабируемость, производительность и надежность систем.

Технологический стек продолжает активно развиваться. Переход от HTTP/1.1 к HTTP/2 и HTTP/3 демонстрирует постоянное улучшение производительности сетевых протоколов. JSON стал стандартом де-факто для обмена данными в RESTful сервисах, хотя XML сохраняет значимость в корпоративных системах. Выбор технологий должен основываться на анализе требований проекта, а не на модных трендах.

Проектирование API требует баланса между техническими требованиями и удобством использования. Правильное использование HTTP-методов и кодов состояния, продуманное именование ресурсов, эффективная

обработка ошибок, грамотное версионирование — все эти аспекты влияют на удобство использования и долгосрочную стоимость поддержки API.

Безопасность веб-сервисов является многогранной областью. OAuth 2.0 и JWT стали стандартизированными механизмами аутентификации и авторизации. Защита требует применения множества механизмов: валидации входных данных, rate limiting, правильной настройки CORS, использования HTTPS с современными версиями TLS. Безопасность должна быть интегрирована в архитектуру системы с самого начала.

Современные тенденции радикально трансформируют подходы к разработке. Микросервисная архитектура предлагает гибкость и независимость развития компонентов. Контейнеризация с Docker обеспечивает консистентность окружений. Kubernetes стал стандартом оркестрации. Serverless абстрагирует инфраструктуру до уровня функций. Каждая технология решает определенные проблемы и вводит свои компромиссы.

Документирование и тестирование являются критически важной частью жизненного цикла веб-сервисов. OpenAPI Specification стандартизировала описание REST API, превратив документацию в машиночитаемый артефакт. Комплексная стратегия тестирования обеспечивает качество и надежность системы на всех уровнях.

Важно понимать, что не существует универсального решения для всех сценариев. REST доминирует в большинстве случаев, но GraphQL предпочтительнее для сложных клиентских приложений, gRPC — для внутреннего межсервисного взаимодействия, SOAP остается релевантным в корпоративных системах. Выбор подхода должен основываться на анализе требований проекта, а не на модных трендах.

Практическая значимость работы подтверждается тем, что систематизированные знания могут быть использованы при проектировании новых веб-сервисов, модернизации существующих систем и в образовательном процессе. Представленный анализ позволяет принимать обоснованные решения при выборе архитектурных подходов, технологий и методов разработки.

Успешное проектирование веб-сервисов требует комплексного подхода, сочетающего глубокое понимание архитектурных принципов и технологий с практическим опытом и способностью критически оценивать компромиссы между различными решениями. Только такой подход позволяет создавать веб-сервисы, которые являются надежными, масштабируемыми, безопасными и эффективными, обеспечивая долгосрочную ценность для бизнеса и качественный опыт для пользователей.

ЛИТЕРАТУРА

1. Аникеев, Д. В. Архитектура информационных систем : учебное пособие / Д. В. Аникеев. — Рязань : РГРТУ, 2022. — 72 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/380360 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.
2. Арбатская, О. А. Информационно-коммуникационные технологии : учебно-методическое пособие / О. А. Арбатская. — Улан-Удэ : ВСГИК, 2020. — 64 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/158638 (дата обращения: 21.12.2025). — Режим доступа: для авториз. пользователей.
3. Баланов, А. Н. Цифровые платформы и системы : учебное пособие для вузов / А. Н. Баланов. — Санкт-Петербург : Лань, 2024. — 452 с. — ISBN 978-5-507-49532-0. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/424577 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.
4. Баланов, А. Н. Бэкенд-разработка веб-приложений: архитектура, проектирование и управление проектами : учебное пособие для СПО / А. Н. Баланов. — 2-е изд., стер. — Санкт-Петербург : Лань, 2026. — 68 с. — ISBN 978-5-507-51307-9. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/510023 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.
5. Вержаковская, М. А. Управление, проектирование и разработка информационных систем, баз данных и Web-ресурсов с использованием современных языков программирования : учебное пособие / М. А. Вержаковская, В. Ю. Аронов. — Самара : ПГУТИ, 2022. — 186 с. — Текст : электронный // Лань : электронно-библиотечная система. —

URL: https://e.lanbook.com/book/411533 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.

6. Гельбух, С. С. Сети ЭВМ и телекоммуникации. Архитектура и организация : учебное пособие / С. С. Гельбух. — Санкт-Петербург : Лань, 2022. — 208 с. — ISBN 978-5-8114-3474-9. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/206585 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.
7. Инструментальное программное обеспечение разработки и проектирования информационных систем : учебное пособие / А. А. Куликов, В. Т. Матчин, А. В. Синицын, В. В. Литвинов. — Москва : РТУ МИРЭА, 2022. — 263 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/311003 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.
8. Исаева, И. А. Информационные системы управления корпоративным контентом : учебное пособие / И. А. Исаева. — Москва : РТУ МИРЭА, 2023 — Часть 1 — 2023. — 70 с. — ISBN 978-5-7339-1723-8. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/331511 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.
9. Лагунова, А. Д. Архитектура интеграции : учебное пособие / А. Д. Лагунова, Д. М. Перегудова. — Москва : РТУ МИРЭА, 2024. — 100 с. — ISBN 978-5-7339-2220-1. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/421106 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.
10. Лоре, А. Проектирование веб-API : руководство / А. Лоре ; перевод с английского Д. А. Беликова. — Москва : ДМК Пресс, 2020. — 440 с. — ISBN 978-5-97060-861-6. — Текст : электронный // Лань : электронно-

библиотечная система. — URL: https://e.lanbook.com/book/179498 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.

11. Никулова, Г. А. Web-дизайн. Приемы адаптивного Web-дизайна: технологии Flexbox и CSS Grid : учебное пособие / Г. А. Никулова, А. С. Терлецкий. — Липецк : Липецкий ГПУ, 2021. — 69 с. — ISBN 978-5-907461-41-3. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/228698 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.
12. Нурмагомедова, Н. Х. WEB- технологии. Курс лекций : учебное пособие / Н. Х. Нурмагомедова, Г. Г. Исаева. — Махачкала : ДГПУ, 2022. — 81 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/262442 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.
13. Смоленцева, Т. Е. Базовые и прикладные информационные технологии. Разработка Web-приложений : учебно-методическое пособие / Т. Е. Смоленцева. — Москва : РТУ МИРЭА, 2021. — 78 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/218702 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.
14. Соловьева, О. М. Web-программирование : учебно-методическое пособие / О. М. Соловьева. — Санкт-Петербург : СПбГУТ им. М.А. Бонч-Бруевича, 2023. — 123 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/425960 (дата обращения: 21.12.2025). — Режим доступа: для авториз. пользователей.
15. Сычев, А. В. Web-технологии : учебное пособие / А. В. Сычев. — Воронеж : ВГУ, 2021. — 163 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL:

https://e.lanbook.com/book/455018 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.

16. Соснин, П. И. Архитектурное моделирование автоматизированных систем : учебник для вузов / П. И. Соснин. — 2-е изд., стер. — Санкт-Петербург : Лань, 2024. — 180 с. — ISBN 978-5-507-49488-0. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/393065 (дата обращения: 21.12.2025). — Режим доступа: для авториз. пользователей.

17. Актуальные вопросы обеспечения комплексной безопасности. Часть 1: материалы национальной научно-практической конференции с международным участием, посвященной 35-летию МЧС России и 95-летию Оренбургского ГАУ : материал конференции / ответственный редактор А. Д. Тарасов. — Оренбург : Оренбургский ГАУ, 2025. — 1799 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/510083 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.

18. Новые математические методы и компьютерные технологии в проектировании, производстве и научных исследованиях: материалы XXV Республиканской научной конференции студентов и аспирантов (Гомель, 21–23 марта 2022 г.) : материалы конференции / редакторы С. П. Жогаль [и др.]. — Гомель : ГГУ имени Ф. Скорины, 2022. — 323 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/329672 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.

19. Технологии веб-сервисов : учебно-методическое пособие / А. М. Дергачев, Ю. Д. Кореньков, И. П. Логинов, А. Г. Сафронов. — Санкт-Петербург : НИУ ИТМО, 2021. — 100 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL:

https://e.lanbook.com/book/283676 (дата обращения: 12.12.2025). — Режим доступа: для авториз. пользователей.

20. ГОСТ 7.32-2017. Система стандартов по информации, библиотечному и издательскому делу. Отчет о научно-исследовательской работе. Структура и правила оформления. — М.: Стандартинформ, 2017. — 32 с.