Министерство образования Республики Беларусь

Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Жизненный цикл разработки программного обеспечения

ОТЧЕТ
по лабораторной работе № 3
«Исследование архитектурного решения»

Студенты: М.Д. Солодков
М.И. Борисевич
А.Е. Мальченко
И.Р. Лагодич

Преподаватель: Д.А. Жалейко

МИНСК 2025

ВВЕДЕНИЕ

В данной лабораторной работе проводится исследование архитектурного решения для разрабатываемой системы. Архитектура программного обеспечения играет важную роль в создании надежных, масштабируемых и поддерживаемых приложений, определяя структуру системы, взаимодействие её компонентов и обеспечивая выполнение функциональных и нефункциональных требований. В рамках работы будут рассмотрены как теоретические аспекты проектирования архитектуры, так и практические шаги по анализу и улучшению существующего решения.

Работа разделена на три основные части. В первой части основное внимание уделяется проектированию архитектуры системы на высоком уровне абстракции. Будут изучены теоретические основы, описанные в «Руководстве Microsoft по моделированию приложений», а также определены ключевые аспекты, такие как тип приложения, стратегия развёртывания, выбор технологий, показатели качества и пути реализации сквозной функциональности. Результатом этой части станет структурная схема приложения, представленная в виде функциональных блоков или диаграмм UML, которая отражает архитектуру «To Be».

Во второй части работы проводится анализ существующей архитектуры на основе реального кода, используемого в системе. С помощью автоматизированных средств обратной инженерии будут сгенерированы диаграммы классов, отражающие текущее состояние системы. Это позволит получить представление об архитектуре «As Is» и выявить её особенности.

Третья часть работы посвящена сравнению архитектур «As Is» и «To Be». На основе выявленных различий будут предложены пути улучшения архитектуры, учитывающие принципы проектирования, архитектурные стили и шаблоны. Это позволит не только проанализировать текущее состояние системы, но и наметить направления для её дальнейшего развития и оптимизации.

1 ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ

1.1 Определения типа приложения

Выбор типа приложения – ключевой этап проектирования, определяющий архитектуру и подходы к разработке. В данном случае создается система для проведения тестов скорости интернет-соединения (SpeedTest), состоящая из двух частей: веб-приложения (backend) и настольного приложения (frontend).

Выбранный тип приложения обусловлен следующими факторами:

- Разделение обязанностей: Веб-приложение (backend) выполняет обработку данных, хранение информации о серверах и предоставление API для клиента; Настольное приложение (frontend) предоставляет удобный интерфейс для пользователя и выполняет измерения скорости соединения.
- Производительность и надежность: Обработка измерений происходит на клиентской стороне, снижая нагрузку на сервер; Централизованное хранение информации о серверах в JSON-файле позволяет легко обновлять список доступных серверов.
- Гибкость развертывания: Backend разворачивается на сервере, обеспечивая API-доступ к данным; Frontend распространяется как отдельное приложение, которое пользователи могут установить на свои устройства.
- Масштабируемость: Веб-архитектура позволяет легко добавлять новые серверы без изменения клиентской части; В дальнейшем можно интегрировать облачные технологии для балансировки нагрузки.

Таким образом, выбранное решение с разделением на веб-приложение (backend) и настольное приложение (frontend) обеспечивает удобство использования, надежность и масштабируемость.

1.2 Выбор стратегии развёртывания

Для развертывания системы SpeedTest было выбрано клиент-серверное развертывание с разделением на два уровня (2-tier architecture), поскольку оно обеспечивает баланс между производительностью, удобством масштабирования и простотой взаимодействия между компонентами:

- Настольное приложение (frontend) выполняет роль клиента, взаимодействующего с сервером для получения списка доступных серверов тестирования и передачи результатов тестов.
- Веб-приложение (backend) обрабатывает запросы клиента, предоставляет API, хранит информацию о серверах в JSON-файле и в перспективе может работать с базой данных

Преимущества выбранного развертывания:

- Простота реализации и поддержки – минимальное количество взаимодействующих компонентов снижает сложность разработки и тестирования.
- Гибкость и возможность масштабирования – в дальнейшем можно интегрировать балансировщики нагрузки или расширить систему до 3-уровневой архитектуры.
- Высокая производительность – клиент выполняет измерения скорости локально, снижая нагрузку на сервер.
- Локальная обработка данных – настольное приложение выполняет тест скорости без задержек, вызванных серверной обработкой.

Возможности дальнейшего расширения:

При увеличении нагрузки и числа пользователей возможен переход на 3-уровневую архитектуру с добавлением отдельного слоя базы данных для хранения истории тестов и аналитики. Также можно использовать облачные технологии для балансировки нагрузки и кэширования запросов.

На текущем этапе 2-уровневое развертывание является оптимальным решением для SpeedTest, обеспечивая удобство развертывания, стабильную работу и возможность масштабирования в будущем.

1.3 Обоснование выбора технологии

Ключевым фактором при выборе технологий является соответствие требованиям проекта, топологии развертывания и архитектурным принципам. Учитывались производительность, удобство развертывания, безопасность и масштабируемость.

Для серверной части выбран ASP.NET Core, так как этот фреймворк обеспечивает высокую производительность, удобство разработки и поддержку асинхронных операций, что критично для работы с сетевыми запросами. ASP.NET Core поддерживает кроссплатформенность, что позволяет запускать серверную часть как в Windows-среде, так и на Linux-серверах. Использование данного фреймворка также предоставляет гибкость при интеграции с другими технологиями, включая облачные решения.

В качестве хранилища данных выбран JSON-файл, так как это удобный и лёгкий формат для хранения и обработки информации о серверах SpeedTest. JSON обеспечивает простоту чтения и записи данных, что упрощает развертывание системы без необходимости использовать полноценную реляционную базу данных. Данный подход подходит для проекта, где информация о серверах может обновляться динамически, а требования к сложным SQL-запросам отсутствуют.

Для клиентской части настольного приложения выбран WPF (Windows Presentation Foundation), так как он позволяет создавать удобный графический интерфейс с высокой производительностью и широкими возможностями кастомизации. WPF поддерживает шаблоны, биндинг данных и

масштабируемость UI, что позволяет создавать удобный и отзывчивый интерфейс для пользователей. Это особенно важно для приложения SpeedTest, где важна визуализация результатов тестов скорости и интерактивные элементы управления.

Выбранные технологии обеспечивают баланс между производительностью и гибкостью, что особенно важно для проекта SpeedTest. ASP.NET Core позволяет эффективно обрабатывать запросы, JSON – хранить и управлять данными в удобном формате, а WPF – создавать современный и удобный интерфейс. Такой стек технологий соответствует требованиям веб-приложения и стратегии развертывания, обеспечивая надежность, безопасность и высокую производительность.

1.4 Показатели качества

При проектировании системы SpeedTest ключевыми показателями качества являются:

Производительность – критически важна для быстрого выполнения тестов скорости сети. ASP.NET обеспечивает высокую скорость обработки запросов, а асинхронный подход позволяет минимизировать задержки. В десктопном клиенте C# оптимизированное взаимодействие с сетью и кеширование данных позволяют ускорить работу.

Безопасность – так как система работает с потенциально конфиденциальными данными пользователей (например, IP-адреса, провайдеры), важно обеспечить надежную защиту. ASP.NET предоставляет встроенные механизмы аутентификации, авторизации и защиты от атак (CSRF, XSS, SQL-инъекций). JSON-файл для хранения серверов защищен механизмами контроля доступа.

Масштабируемость – важный параметр, так как количество пользователей может расти. Использование микросервисной архитектуры на основе ASP.NET позволяет легко расширять систему. Клиентская часть, написанная на C#, обеспечивает гибкость в расширении функционала.

Удобство и простота использования – обеспечивается за счет интуитивно понятного интерфейса в десктопном приложении. Разделение логики на сервисы и контроллеры в backend улучшает читаемость кода и удобство поддержки.

Надежность – система устойчива к сбоям за счет механизмов логирования, мониторинга и обработки ошибок. ASP.NET и C# предлагают встроенные инструменты для диагностики и анализа работы приложения.

Возможность повторного использования – архитектура построена так, чтобы отдельные компоненты (например, сервисы работы с JSON, модуль измерения скорости, механизмы логирования) можно было использовать в других проектах без значительных доработок.

Эти показатели помогают создать эффективное, удобное и производительное решение для тестирования скорости интернет-соединения.

1.5 Решение о путях реализации сквозной функциональности

Для обеспечения надежности, безопасности и производительности системы реализованы следующие механизмы:

Протоколирование — централизованный журнал событий фиксирует ключевые операции в системе, упрощая отладку и анализ.

Обработка исключений — ошибки перехватываются на границах слоев и обрабатываются централизованно, предотвращая утечки данных.

Связь между слоями — REST API с поддержкой HTTP/HTTPS обеспечивает безопасность и минимизирует сетевые задержки.

Связь между слоями будет осуществляться через REST API с использованием HTTP/HTTPS протоколов. Это обеспечит минимальное количество сетевых вызовов и защиту передаваемых данных. Для повышения производительности будут использоваться асинхронные запросы.

Кэширование будет применяться для ускорения работы клиента, чтобы не выполнять долгие операции с получением сервером.

Эти механизмы повышают стабильность, безопасность и производительность системы, обеспечивая удобство в поддержке и масштабировании.

1.6 Структурная схема приложения

Структурная схема приложения в виде функциональных блоков представлена на рисунке 1.1.

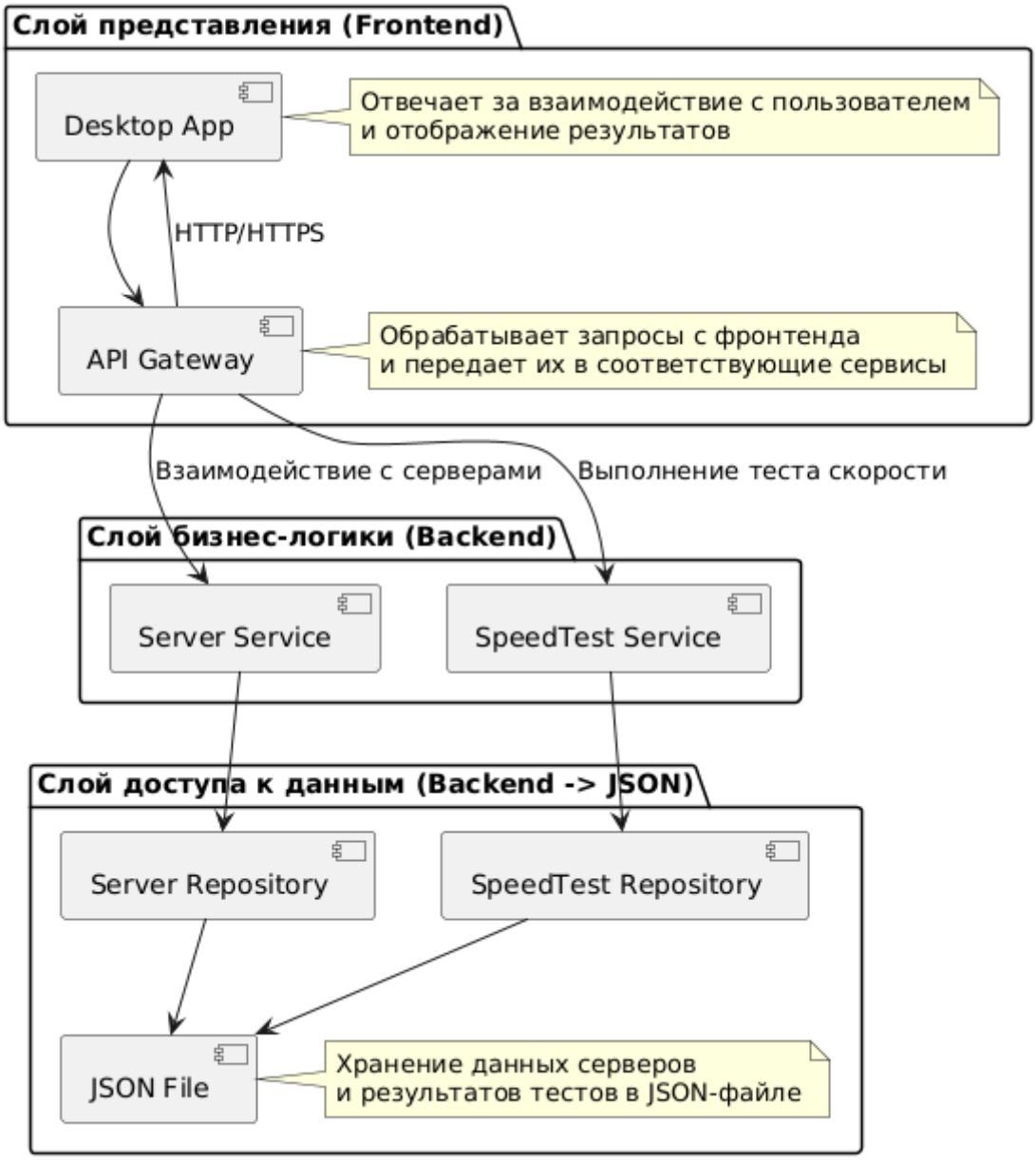

Рисунок 1.1 – Схема приложения

2 АНАЛИЗ АРХИТЕКТУРЫ

На данном этапе проведён анализ существующей архитектурысистемы, сформированной в ходе первого Sprint Review. С использованием инструмента обратной инженерии была сгенерирована диаграмма классов, отображающая структуру ключевых компонентов. Эта диаграмма классов представлена на рисунке 2.1.

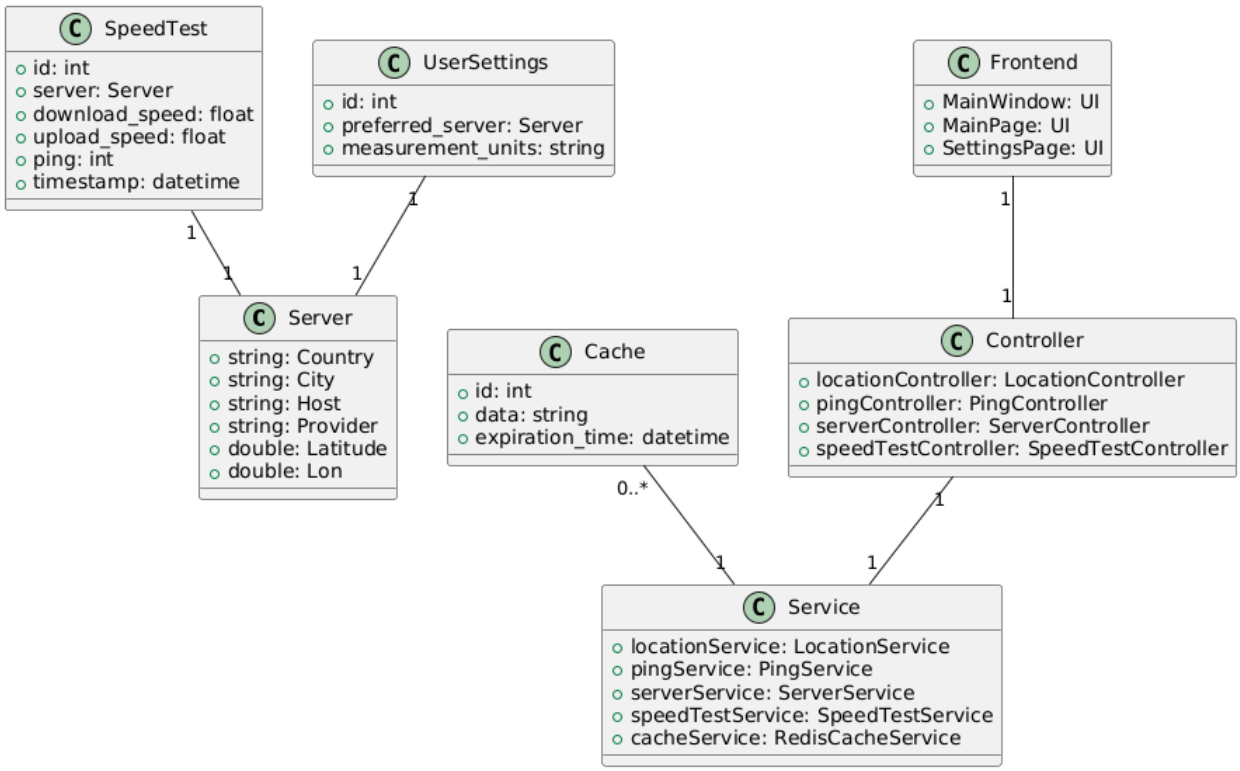

Рисунок 2.1 – Диаграмма классов

Связи между классами отражают их отношения, выявленные в исходном коде. Диаграмма демонстрирует основные модели системы:

SpeedTest – класс, отвечающий за проведение тестов скорости интернета.

Server – содержит информацию о серверах, используемых для тестирования.

UserSettings – хранит настройки пользователей, включая предпочтительный сервер и единицы измерения.

Cache – реализует механизм кэширования данных, чтобы оптимизировать работу системы.

Controller – управляет взаимодействием между пользователем и системой, обрабатывая запросы.

Service – предоставляет основные сервисы, такие как работа с серверами, измерение скорости и управление кэшем.

Frontend – отвечает за пользовательский интерфейс и отображение данных.

3 СРАВНЕНИЕ И РЕФАКТОРИНГ

3.1 Сравнение «As is» и «To be»

As is (текущая архитектура)

- Приложение разделено на две основные части: WEB (backend) и DESKTOP (frontend).
- Backend представляет собой ASP.NET приложение с разделением на основные модули: Cache, Model, Service, Controller.
- Данные серверов хранятся в JSON-файле, что упрощает работу с конфигурацией, но не масштабируется для больших объемов данных.
- Frontend реализован как WPF-приложение, содержащее модули Model, Service, Views, Helpers.
- Взаимодействие между frontend и backend вероятно осуществляется через REST API.
- Отсутствует API Gateway, frontend напрямую отправляет запросы к backend.
- Отсутствует четкое разделение логики и слоев абстракции в backend, например, репозитории для работы с данными могут отсутствовать.

To be (целевая архитектура)

- Введение API Gateway как единой точки входа для маршрутизации запросов между frontend и backend..
- Разделение backend на логические сервисы (например, отдельный сервис для работы с серверами, хранения результатов тестирования и кэша).
- Использование базы данных (SQL/NoSQL) вместо JSON-файлов для хранения данных о серверах.
- Внедрение уровней абстракции: репозитории, сервисы, контроллеры для улучшения масштабируемости и сопровождения.
- Развитие frontend: возможно, переход на более современный UI-фреймворк (например, Blazor или ElectronJS) для лучшего взаимодействия с пользователем.
- Добавление очередей сообщений (Kafka, RabbitMQ) для асинхронной обработки запросов и улучшения отказоустойчивости.

3.2 Выделенные отличия и их причины

Отличие	Причина
Введение API Gateway	Централизованная маршрутизация, защита и балансировка нагрузки
Использование базы данных вместо JSON	

	Уменьшение ограничений по хранению данных, улучшение масштабируемости
Разделение backend на сервисы	Облегчение сопровождения и улучшение отказоустойчивости
Возможное внедрение асинхронного взаимодействия	Повышение производительности за счет распределенной обработки

3.3 Пути улучшения архитектуры

1. Оптимизация взаимодействия между сервисами
   - Внедрение gRPC или GraphQL для оптимизированной работы с данными.
   - Использование кеширования (Redis, MemoryCache) для ускорения работы API.
2. Обновление структуры данных
   - Перенос хранения серверов в базу данных (PostgreSQL, MongoDB) для масштабируемости.
   - Добавление логирования и мониторинга (например, Prometheus, ELK-stack) для анализа работы системы.
3. Развитие frontend
   - Рассмотрение перехода на Blazor/ElectronJS для улучшенной интеграции с backend.
   - Улучшение UI/UX с учетом отзывчивого дизайна и адаптивности.
4. Безопасность
   - Внедрение OAuth2/OpenID Connect для безопасной аутентификации пользователей.
   - Реализация механизма rate limiting для предотвращения перегрузки системы.
5. Горизонтальное масштабирование
   - Разделение базы данных на мастер-слейв репликацию.
   - Внедрение автоматического масштабирования (Autoscaling) в облачных решениях (AWS, Azure, Kubernetes).

Эти изменения сделают систему более гибкой, отказоустойчивой и удобной в эксплуатации.