Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Кузбасский государственный технический университет
имени Т. Ф. Горбачева»

Институт профессионального образования

Кафедра информатики и информационных систем

Составитель К. В. Ерошевич

Осуществление интеграции программных модулей

Методические материалы

Рекомендовано цикловой методической комиссией
Информационных систем и программирования
в качестве электронного издания
для использования в образовательном процессе

Кемерово 2026

Рецензент:

К. А. Кулиничев, преподаватель первой квалификационной категории кафедры информатики и информационных систем

Ерошевич Кирилл Владимирович

Осуществление интеграции программных модулей : методические материалы для обучающихся специальности СПО 09.02.11 Разработка и управление программным обеспечением / Кузбасский государственный технический университет имени Т. Ф. Горбачева, Кафедра информатики и информационных систем ; составитель К. В. Ерошевич. – Кемерово, 2026. – 1 файл (935 Кб). – Текст : электронный.

Методические материалы по дисциплине Осуществление интеграции программных модулей: методические материалы для обучающихся специальности СПО 09.02.11 «Разработка и управление программным обеспечением» содержат указания для проведения лабораторных работ.

Лабораторная работа №1 «Создание клиентского приложения для работы с публичным API»

Теоретическая часть

Клиент-серверное взаимодействие и API

В современной разработке программного обеспечения широко распространена архитектура, где приложение (клиент) отправляет запросы для получения данных или выполнения действий на удаленном сервере. Для организации такого взаимодействия используются API (Application Programming Interface) – программный интерфейс приложения. API определяет правила, по которым одна программа может взаимодействовать с другой.

REST API

Одним из самых популярных подходов к построению API является REST (Representational State Transfer). RESTful API использует стандартные HTTP-методы для выполнения операций с ресурсами:

1. GET – получение данных (например, списка подразделений).
2. POST – создание нового ресурса.
3. PUT / PATCH – обновление существующего.
4. DELETE – удаление ресурса.

Взаимодействие с REST API сводится к отправке HTTP-запросов на определенные конечные точки (endpoints, например, https://portal.kuzstu.ru/api/structure) и получению ответа. Данные чаще всего передаются в формате JSON (JavaScript Object Notation) – легком и удобном для чтения текстовом формате.

Реализация в C#

В языке C# для работы с HTTP-запросами используется класс HttpClient. Он предоставляет методы для отправки запросов (GetAsync, PostAsync и т. д.) и получения ответов. Полученный JSON-ответ необходимо обработать – преобразовать (десериализовать) в объекты C#. Для этого удобно использовать библиотеку Newtonsoft.Json или встроенные средства System.Text.Json.

Задание

Цель работы: Разработать консольное приложение на языке C#, которое обращается к публичному API КузГТУ и получает список всех структурных подразделений.

Задачи:

1. Изучить структуру данных, возвращаемых API по адресу https://portal.kuzstu.ru/api/structure.
2. Создать в приложении C# классы (модели данных), соответствующие структуре JSON-ответа.
3. Написать код для отправки GET-запроса к API с использованием HttpClient.
4. Получить JSON-ответ, содержащий иерархический список подразделений (с вложенными дочерними элементами children).
5. Выполнить десериализацию JSON-ответа в созданные объекты C#.
6. Вывести на экран (в консоль) информацию обо всех полученных структурных подразделениях в удобном для чтения виде. Например, можно реализовать рекурсивный вывод с отступами, чтобы показать иерархию.

Контрольные вопросы

1. Что такое API и для чего он используется?
2. Что означает аббревиатура REST? Какие основные принципы лежат в основе RESTful API?
3. Какие основные HTTP-методы вы знаете и за что каждый из них отвечает?
4. Что такое JSON? Почему он часто используется для обмена данными в веб-приложениях?
5. Для чего в C# используется класс HttpClient? Какие основные методы этого класса вы узнали?
6. Что такое десериализация? Какую роль она играет при работе с API?
7. Как обработать возможные ошибки при сетевом взаимодействии (например, отсутствие интернета или недоступность сервера)?

Лабораторная работа №2 «Создание REST API приложения с реализацией CRUD для 3-4 сущностей»

Теоретическая часть

REST API и CRUD

REST API (Representational State Transfer) – это архитектурный стиль взаимодействия компонентов распределенного приложения в сети. В центре внимания REST находятся ресурсы, которые предоставляются по определенным URL-адресам (эндпоинтам) .

CRUD (Create, Read, Update, Delete) — акроним, обозначающий четыре базовые функции управления данными. В RESTful API им ставятся в соответствие HTTP методы:

1. CREATE (Создание) – POST (для создания новых ресурсов).
2. READ (Чтение) – GET (для получения одного или списка ресурсов).
3. UPDATE (Обновление) – PUT (полное обновление) или PATCH (частичное обновление).
4. DELETE (Удаление) – DELETE .

ASP.NET Core Web API

Это фреймворк от Microsoft для создания HTTP API, оптимизированных для таких клиентов, как браузеры и мобильные приложения. Идеальная платформа для создания RESTful приложений на C# .

Entity Framework Core (EF Core)

Объектно-реляционный преобразователь (ORM), который позволяет разработчикам .NET работать с базой данных с помощью объектов .NET. EF Core автоматизирует создание и обновление схемы базы данных на основе ваших классов C# (моделей), а также предоставляет возможности для выполнения запросов, вставки, обновления и удаления данных.

Отношения между сущностями

Для работы с несколькими сущностями важно определить связи между ними. Основные типы отношений:

1. Один-ко-многим (1:N) : Например, один "Поставщик" может иметь много "Товаров". В модели C# это выражается через свойство навигации: в классе Product будет ссылка на Supplier

(public Supplier Supplier { get; set; }), а в классе Supplier – коллекция продуктов (public ICollection<Product> Products { get; set; }).

2. Многие-ко-многим (N:N) : Например, "Заказ" может содержать много "Товаров", и каждый "Товар" может быть во многих "Заказах". Реализуется через промежуточную таблицу (например, OrderItems).

Data Transfer Objects (DTO)

Это объекты, которые используются для передачи данных между процессами/приложениями. При создании API рекомендуется не возвращать клиенту ваши внутренние модели данных (сущности БД), а преобразовывать их в DTO. Это позволяет:

1. Скрыть внутреннюю структуру БД.
2. Исключить циклические ссылки при загрузке связанных данных.
3. Отправлять только те данные, которые действительно нужны клиенту.

Задание

Цель работы: Разработать REST API приложение на ASP.NET Core, которое реализует CRUD-операции для 3–4 связанных сущностей с использованием Entity Framework Core и базы данных (например, SQLite или SQL Server).

Вариант задания "Управление складом".

Необходимо реализовать API для работы со следующими сущностями и их связями:

1. Поставщик (Supplier) – имеет название, контактные данные, адрес.
2. Товар (Product) – имеет название, описание, цену, а также ссылку на поставщика (один товар может поставляться только одним поставщиком, но у поставщика много товаров – связь "один-ко-многим").
3. Склад (Warehouse) – имеет название, адрес.
4. Запас (Stock) – связующая сущность, показывающая, какой товар и в каком количестве хранится на каком складе (связь "многие-ко-многим" между Product и Warehouse через Stock).

Задачи:

1. Создать новый проект ASP.NET Core Web API.
2. Установить необходимые NuGet пакеты: Microsoft.EntityFrameworkCore, Microsoft.EntityFrameworkCore.Sqlite (или SqlServer), Microsoft.EntityFrameworkCore.Tools.
3. Создать модели (Supplier, Product, Warehouse, Stock) и настроить связи между ними с использованием Data Annotations или Fluent API.
4. Создать класс контекста базы данных (AppDbContext), унаследовав его от DbContext, и добавить в него DbSet<T> для каждой из четырех сущностей.
5. Зарегистрировать контекст в Program.cs.
6. Создать и применить миграции для создания базы данных.
7. Создать DTO (Data Transfer Objects) для каждой сущности, чтобы избежать циклических ссылок и скрыть детали реализации.
8. Создать контроллеры (ProductsController, SuppliersController, WarehousesController) и реализовать в них методы для CRUD-операций с использованием HTTP методов GET, POST, PUT, DELETE.

Требования к реализуемым методам (на примере ProductsController) (таблица 1):

Таблица 1

Метод	HTTP	URL	Описание
GetAll	`GET`	`/api/products`	Получить список всех товаров (с информацией о поставщике)
GetById	`GET`	`/api/products/{id}`	Получить конкретный товар по ID
Create	`POST`	`/api/products`	Создать новый товар (данные в теле запроса JSON)
Update	`PUT`	`/api/products/{id}`	Полностью обновить товар по ID
Delete	`DELETE`	`/api/products/{id}`	Удалить товар по ID

Контрольные вопросы

1. Что такое REST API и какие основные принципы он использует?
2. Что означает аббревиатура CRUD? Как она сопоставляется с HTTP методами?

3. Что такое Entity Framework Core и для чего он нужен? Объясните понятие "миграции"?
4. Что такое DTO и почему важно их использовать при создании API? Как DTO помогают решить проблему циклических ссылок (object cycle detection)?
5. Какие типы связей между сущностями в БД вы знаете? Как они реализуются в C# с помощью EF Core?
6. Объясните разницу между HTTP методами PUT и PATCH.
7. Что такое внедрение зависимостей (Dependency Injection) и как оно применяется при работе с DbContext в контроллерах?

Лабораторная работа №3 «Расширение функционала REST API приложения: работа с удаленным источником данных»

Теоретическая часть

Понятие удаленного источника данных

В современной архитектуре приложений часто возникает необходимость не только предоставлять собственные данные через API, но и интегрироваться с внешними сервисами. Удаленный источник данных – это любой внешний сервис или API, расположенный на другом сервере, который предоставляет необходимую информацию (например, данные о погоде, курсах валют, географические данные и т. д.).

Шаблоны интеграции с внешними API

При интеграции с удаленными источниками данных используются следующие подходы:

1. Прямые HTTP-запросы из контроллера: Простой, но не самый гибкий подход, при котором логика обращения к внешнему API находится непосредственно в контроллере.
2. Выделенные сервисы-клиенты (Client Services): Создание отдельных классов, отвечающих за взаимодействие с конкретным внешним API. Это улучшает тестируемость и переиспользование кода.
3. Использование IHttpClientFactory: Фабрика для создания экземпляров HttpClient, которая решает проблемы управления временем жизни соединений и предотвращает истощение сокетов (socket exhaustion).

Асинхронное программирование

При работе с удаленными источниками данных критически важно использовать асинхронные методы (async/await), чтобы не блокировать поток выполнения при ожидании ответа от внешнего сервиса. Это позволяет вашему API оставаться отзывчивым и масштабируемым.

Обработка ошибок и устойчивость

Внешние сервисы могут быть недоступны или отвечать медленно. Для повышения отказоустойчивости применяются:

1. Политики повторных попыток (Retry policies): Автоматическое повторение запроса при временных сбоях.

2. Обработка таймаутов: Установка разумных пределов ожидания ответа.
3. Паттерн Circuit Breaker: Предотвращение повторных вызовов заведомо неработающего сервиса.

Задание

Цель работы: Расширить существующее REST API приложение (из Лабораторной работы №2) функциональностью для получения и интеграции данных из внешнего (удаленного) API.

Вариант задания: "Обогащение данных склада"

Необходимо добавить в API возможность обогащать информацию о товарах данными из внешнего открытого API.

Задачи:

1. Выбрать внешний открытый API для получения дополнительной информации. Рекомендуемые варианты (бесплатные, без ключа или с простой регистрацией):
   - RestCountries (https://restcountries.com/) – получение информации о стране.
   - OpenLibrary (https://openlibrary.org/developers/api) – поиск книг по ISBN.
   - JSONPlaceholder (https://jsonplaceholder.typicode.com/) – тестовый API с постами, комментариями и т. д.

   Примечание: В рамках лабораторной работы можно выбрать любой публичный API, возвращающий JSON.
2. Создать отдельный сервис-клиент (например, CountryInfoService или BookInfoService) для взаимодействия с выбранным внешним API. Сервис должен:
   - Принимать в конструкторе HttpClient (через внедрение зависимости).
   - Содержать методы для получения данных из внешнего API (например, GetCountryInfoAsync(string countryName)).
   - Выполнять десериализацию ответа в соответствующие модели C# (DTO).
   - Создать модели (DTO) только для тех полей, которые нужны из ответа внешнего API.
3. Интегрировать созданный сервис в один из существующих контроллеров (например, ProductsController):

- Добавить новый эндпоинт, который принимает идентификатор товара и название страны, а возвращает информацию о товаре, дополненную данными о стране (например, столица, регион, население).
- Добавить эндпоинт для обогащения списка товаров данными из внешнего API (например, при получении списка товаров добавить для каждого страну происхождения с дополнительной информацией о ней).

4. Зарегистрировать сервис-клиент в Program.cs с использованием AddHttpClient<T>.
5. Реализовать обработку возможных ошибок при обращении к внешнему API (таймауты, недоступность сервиса) с возвратом соответствующих HTTP-статусов (например, 503 Service Unavailable или 502 Bad Gateway).

Требования к новым методам API:

1. GET /api/products/{id}/enrich?country={countryName} – получить товар по ID, обогащенный данными о стране (название, столица, регион, население).
2. GET /api/products/enriched – получить все товары, где для каждого товара добавлена информация о стране (если страна указана в свойствах товара).

Контрольные вопросы

1. Что такое удаленный источник данных и для чего нужна интеграция с внешними API?
2. Почему для работы с HttpClient рекомендуется использовать IHttpClientFactory? Какие проблемы это решает?
3. В чем преимущество выделения логики обращения к внешнему API в отдельный сервис-клиент?
4. Как обеспечить отказоустойчивость приложения при обращении к ненадежному внешнему API?
5. Что такое политика повторных попыток (retry policy) и как ее можно реализовать в .NET?
6. Какие HTTP-статус коды следует возвращать клиенту, если внешний API недоступен или вернул ошибку?
7. Объясните разницу между синхронным и асинхронным вызовом внешнего API. Почему важно использовать async/await?

Лабораторная работа №4 «Расширение функционала REST API приложения: работа со статическими изображениями (ресурсами) – загрузка, передача, удаление»

Теоретическая часть

Статические файлы в ASP.NET Core

Статические файлы – это активы (assets), которые сервер может напрямую отправлять клиенту без дополнительной обработки. К ним относятся HTML, CSS, JavaScript, изображения и другие файлы. В ASP.NET Core такие файлы по умолчанию хранятся в папке wwwroot (web root).

Для того чтобы статические файлы были доступны клиентам, необходимо настроить статический файловый middleware с помощью метода UseStaticFiles() в Program.cs. Это позволяет обращаться к файлам по прямым URL (например, http://localhost:5000/images/photo.jpg).

Загрузка файлов через Web API

Для приема файлов от клиента в ASP.NET Core используется интерфейс IFormFile. Он представляет собой файл, отправленный с HTTP-запроса. Метод действия контроллера должен принимать параметр типа IFormFile, а сам запрос должен иметь тип multipart/form-data.

Обработка файлов

После получения файла через IFormFile можно:

1. Сохранить на диске: Скопировать поток файла в файловую систему сервера с помощью FileStream.
2. Сохранить в базе данных: Преобразовать файл в массив байтов (byte[]) и сохранить в поле таблицы.
3. Передать в облачное хранилище: Отправить файл во внешний сервис (например, Azure Blob Storage, Cloudinary).
4. При сохранении файлов важно обеспечить уникальность имен, чтобы избежать перезаписи. Обычно для этого используется Guid.NewGuid().ToString().

Безопасность и ограничения

При работе с загружаемыми файлами необходимо учитывать следующие аспекты:

1. Ограничение размера файла: Чтобы избежать перегрузки сервера, следует установить лимит на максимальный размер загружаемого файла.
2. Проверка типа файла: Необходимо проверять расширение и MIME-тип файла, чтобы предотвратить загрузку опасного контента (например, скриптов).
3. Антивирусная проверка: В производственных системах загруженные файлы должны проверяться антивирусом.
4. Настройка FormOptions: Для поддержки больших загрузок может потребоваться настройка лимитов в FormOptions.

Удаление файлов

При удалении сущности (например, товара или сотрудника) необходимо также удалять связанный файл изображения, чтобы не захламлять сервер. Удаление может выполняться из файловой системы или из облачного хранилища.

Передача файлов клиенту

Существует два основных способа передачи изображений клиенту:

1. Как статический файл: Через настроенный Static Files middleware по прямому URL.
2. Как файловый результат (FileResult): Через специальный метод контроллера, который считывает файл с диска или из БД и возвращает его с соответствующим MIME-типом. Это позволяет добавить авторизацию к доступу к файлам.

Задание

Цель работы: Расширить существующее REST API приложение (из Лабораторной работы №2 или №3) функциональностью для загрузки, хранения, получения и удаления изображений, связанных с основными сущностями.

Вариант задания: "Добавление изображений к товарам"

Необходимо добавить в API возможность загрузки и управления изображениями для товаров (Product).

Задачи:

1. Подготовка инфраструктуры:
    - Создать в проекте папку wwwroot (если ее нет).
    - Внутри wwwroot создать подпапку images (или product-images) для хранения загруженных файлов.

- Настроить статические файлы в Program.cs, добавив app.UseStaticFiles().

2. Модификация модели данных:
   - Добавить в модель Product новое свойство string ImagePath для хранения пути к файлу изображения (или имени файла).
   - Создать и применить новую миграцию для обновления базы данных.
3. Создание моделей DTO для работы с изображениями:
   - Создать DTO для загрузки изображения (например, ProductImageUploadDto), содержащее поле IFormFile Image.
   - Создать DTO для возврата данных о товаре с изображением (включая полный URL для доступа).
4. Реализация методов контроллера (таблица 2):

Таблица 2

Метод	HTTP	URL	Описание
UploadImage	`POST`	`/api/products/{id}/image`	Загрузить изображение для конкретного товара. Старое изображение (если есть) должно быть удалено
GetImage	`GET`	`/api/products/{id}/image`	Получить изображение товара (как файл)
DeleteImage	`DELETE`	`/api/products/{id}/image`	Удалить изображение товара
UpdateProductWithImage	`PUT`	`/api/products/{id}`	Обновить данные товара вместе с новым изображением (опционально)

5. Дополнительные требования:
   - При загрузке изображения генерировать уникальное имя файла с помощью Guid.

- Провести валидацию загружаего файла: проверять расширение (только .jpg, .jpeg, .png, .gif) и максимальный размер (например, не более 2–5 МБ).
- При успешной загрузке возвращать URL для доступа к изображению (например, https://localhost:5001/images/{unique-filename}.jpg).
- При удалении товара автоматически удалять связанный файл изображения.

Контрольные вопросы

1. Что такое статические файлы в ASP.NET Core и для чего нужен метод UseStaticFiles()?
2. Какое назначение имеет папка wwwroot в проекте ASP.NET Core?
3. Какой интерфейс используется для приема загружаемых файлов в Web API? Как получить файл из запроса?
4. Почему при сохранении файлов на сервере важно генерировать уникальные имена? Как это можно сделать с помощью C#?
5. Какие существуют способы валидации загружаемых файлов (тип, размер)?
6. Как организовать удаление старого файла при загрузке нового или при удалении сущности?
7. В чем разница между возвратом файла как статического ресурса и через метод контроллера, возвращающий FileResult? В каких случаях предпочтительнее каждый из подходов?

Лабораторная работа №5 «Расширение функционала REST API приложения: обработка path и query параметров»

Теоретическая часть

Маршрутизация (Routing) в ASP.NET Core

Маршрутизация – это процесс сопоставления входящего HTTP-запроса с методом действия (action method) контроллера, который может его обработать. ASP.NET Core предоставляет *два основных подхода к маршрутизации*:

1. Обычная (conventional) маршрутизация: Шаблоны маршрутов определяются централизованно (например, в Program.cs) и применяются ко многим контроллерам.
2. Маршрутизация через атрибуты (attribute routing): Шаблоны определяются непосредственно в контроллерах и методах действий с помощью атрибутов, таких как [Route], [HttpGet], [HttpPost].

Параметры маршрута (Path Parameters)

Параметры маршрута (или сегменты URL) являются частью самого URL-адреса и заключаются в фигурные скобки {} в шаблоне маршрута. Они используются для обязательной идентификации ресурса, например, его уникального идентификатора.

Пример: В шаблоне "products/{id}" параметр id будет извлечен из URL /products/5.

Можно также определять необязательные параметры, добавляя символ **?** в шаблоне: "products/{id?}". Для таких параметров необходимо предусмотреть значение по умолчанию в сигнатуре метода.

Параметры строки запроса (Query Parameters)

Параметры строки запроса следуют после символа **?** в URL и имеют формат ключ=значение. Они обычно используются для фильтрации, сортировки, пагинации или передачи дополнительных, необязательных данных.

Пример: /api/products?category=books&page=2

Привязка параметров (Parameter Binding)

Привязка параметров – это процесс преобразования данных HTTP-запроса (из маршрута, строки запроса, заголовков, тела и т. д.) в строго типизированные параметры метода действия C#. ASP.NET Core автоматически выполняет эту привязку.

Для явного указания источника привязки используются атрибуты:

1. [FromRoute] – привязать значение из параметров маршрута.
2. [FromQuery] – привязать значение из строки запроса.
3. [FromHeader] – привязать значение из HTTP-заголовка.
4. [FromBody] – привязать значение из тела запроса (например, для JSON).
5. [FromForm] – привязать значение из данных формы.

Если атрибут не указан, фреймворк пытается определить источник автоматически. Для GET запросов параметры простых типов по умолчанию привязываются из строки запроса.

Валидация параметров с помощью атрибутов

Для обеспечения целостности данных можно использовать атрибуты валидации из пространства имен System.ComponentModel.DataAnnotations:

1. [Required] – параметр обязателен.
2. [Range(min, max)] – значение должно находиться в указанном диапазоне.
3. [StringLength(max)] – строка не должна превышать заданную длину.

Встроенные ограничения маршрута (Inline Route Constraints)

Для параметров маршрута можно задать ограничения прямо в шаблоне, чтобы автоматически проверять их тип или формат:

1. :int – целое число.
2. :guid – уникальный идентификатор.
3. :datetime – дата и время.
4. :bool – логическое значение.
5. :min(значение) – не меньше указанного значения.
6. :max(значение) – не больше указанного значения.
7. :range(мин, макс) – в диапазоне.
8. :alpha — только буквы.
9. :regex(выражение) – соответствует регулярному выражению.

Пример: "products/{id:int}" гарантирует, что маршрут сработает только в том случае, если сегмент id является целым числом.

Модели для сложных параметров запроса

Когда метод действия должен принимать множество параметров строки запроса, удобно объединить их в отдельный

класс (DTO) и использовать атрибут [FromQuery] для привязки всего класса.

Задание

Цель работы: Расширить существующее REST API приложение (из предыдущих лабораторных работ) функциональностью для гибкой обработки параметров маршрута и строки запроса, реализовав фильтрацию, сортировку, пагинацию и валидацию входящих данных.

Задачи:

1. Модификация существующих эндпоинтов для использования параметров маршрута.
1. Реализация пагинации через параметры запроса.
2. Реализация фильтрации через параметры запроса.
3. Реализация сортировки через параметры запроса.
4. Создание модели для сложных параметров запроса.
5. Валидация параметров.

Контрольные вопросы

1. В чем разница между обычной (conventional) маршрутизацией и маршрутизацией через атрибуты в ASP.NET Core?
2. Для чего используются параметры маршрута (path parameters), а для чего – параметры строки запроса (query parameters)? Приведите примеры.
3. Как сделать параметр маршрута необязательным? Какие действия необходимо предпринять в сигнатуре метода для его обработки?
4. Какие атрибуты привязки параметров вы знаете ([FromRoute], [FromQuery], [FromBody] и др.)? В каких случаях каждый из них применяется?
5. Что такое встроенные ограничения маршрута (inline route constraints) и для чего они нужны? Приведите примеры.
6. Как можно принять сложный набор параметров строки запроса в методе действия, чтобы избежать длинного списка параметров?
7. Как реализовать валидацию параметров запроса и корректно обработать ошибки валидации в Web API?

Лабораторная работа №6 «Расширение функционала REST API приложения: обработка ошибок, передача сообщений об ошибке пользователю»

Теоретическая часть

Важность обработки ошибок в Web API

В хорошо спроектированном API обработка ошибок играет критическую роль. При возникновении исключений или ошибок валидации API должен возвращать клиенту информативные, структурированные ответы, соответствующие стандартам, а не просто "падать" с необработанным исключением или возвращать пустой ответ с кодом ошибки.

Страница исключений для разработчика (Developer Exception Page)

ASP.NET Core предоставляет специальную страницу для отображения подробной информации о необработанных исключениях. Она использует DeveloperExceptionPageMiddleware для перехвата исключений из конвейера HTTP и генерации детализированных ответов об ошибках.

Важно: Эта страница должна использоваться только в среде разработки и никогда – в production, так как может раскрыть конфиденциальную информацию (стек вызовов, параметры запроса, заголовки, cookies).

Программное обеспечение (ПО) промежуточного слоя для обработки исключений (Exception Handler Middleware)

В средах, отличных от разработки, следует использовать UseExceptionHandler. Это ПО промежуточного слоя:

1. Перехватывает и логирует необработанные исключения.
2. Повторно выполняет запрос по альтернативному пути (например, /error), чтобы сгенерировать дружественный ответ для клиента.

Problem Details (RFC 7807)

определенном в RFC 7807 ("Problem Details for HTTP APIs"). Этот формат включает такие поля, как type (тип ошибки), title (краткое описание), status (HTTP-статус), detail (детальное описание) и instance (указатель на конкретный экземпляр ошибки).

В ASP.NET Core метод Problem() создает ответ, соответствующий RFC 7807.

Обработка ошибок валидации

При невалидной модели MVC автоматически возвращает ответ с типом ValidationProblemDetails, используя фабрику InvalidModelStateResponseFactory. Это можно настроить для поддержки различных форматов (например, XML).

Обработка клиентских ошибок (Status Code Pages)

По умолчанию для HTTP ошибок 400–599 возвращается только код статуса и пустое тело ответа. Middleware UseStatusCodePages позволяет настроить отображение информативных страниц для этих кодов.

Различные подходы к обработке ошибок

Кастомные исключения и фильтры действий: Создание своего типа исключения (например, HttpResponseException) и фильтра, который перехватывает его и формирует нужный ответ .

IExceptionHandler (начиная с .NET 8): Новый интерфейс для централизованной обработки известных исключений. Позволяет зарегистрировать несколько обработчиков, которые вызываются в порядке регистрации.

Кастомное middleware: Создание собственного ПО промежуточного слоя для полного контроля над процессом обработки ошибок.

Задание

Цель работы: Реализовать комплексную систему обработки ошибок в REST API приложении, обеспечивающую корректное логирование, возврат стандартизированных ответов об ошибках и удобное информирование клиента.

Задачи:

1. Настройка обработки исключений в зависимости от среды.
2. Создать endpoint'ы для обработки ошибок.
3. Создать контроллер (например, ErrorController) с методами.
4. Реализовать обработку ошибок валидации.
5. Создать кастомное middleware для обработки ошибок.
6. Определить кастомные исключения.
7. Настроить обработку HTTP-статусов (404, и т. д.).

Контрольные вопросы

1. Для чего используется страница исключений для разработчика (Developer Exception Page) и почему ее нельзя включать в production?
2. Что такое RFC 7807 и какие поля включает стандартный формат Problem Details?
3. В чем разница между UseExceptionHandler, UseStatusCodePages и кастомным middleware для обработки ошибок?
4. Как обрабатываются ошибки валидации модели в ASP.NET Core Web API? Как кастомизировать ответ при невалидной модели?
5. Что такое IExceptionHandler (в .NET 8+) и чем он удобен по сравнению с традиционными подходами?
6. Почему важно логировать ошибки? Какую информацию следует включать в лог?
7. Какие HTTP-статус коды следует возвращать для различных типов ошибок (не найдено, невалидные данные, внутренняя ошибка сервера и т. д.)?

Лабораторная работа №7 «Расширение функционала REST API приложения: валидация полученных данных»

Теоретическая часть

Модель состояния (Model State) в ASP.NET Core

В ASP.NET Core валидация происходит после привязки модели и перед выполнением метода действия контроллера. Результаты валидации сохраняются в ModelStateDictionary – специальном объекте, который содержит:

1. Ошибки привязки модели (например, попытка присвоить строку числовому полю).
2. Ошибки валидации, возникающие при несоответствии данных бизнес-правилам.
3. ModelState.IsValid возвращает true, если нет ошибок ни привязки, ни валидации.

Встроенные атрибуты валидации

Пространство имен System.ComponentModel.DataAnnotations предоставляет богатый набор атрибутов для декларативной валидации (таблица 3):

Таблица 3

Атрибут	Назначение
`[Required]`	Поле не должно быть null
`[StringLength(max)]`	Ограничение длины строки
`[Range(min, max)]`	Значение в заданном диапазоне
`[EmailAddress]`	Проверка формата email
`[Phone]`	Проверка формата телефона
`[RegularExpression(pattern)]`	Проверка по регулярному выражению
`[Compare]`	Сравнение двух свойств (например, пароль и подтверждение)
`[CreditCard]`	Проверка формата кредитной карты
`[Url]`	Проверка формата URL

Сообщения об ошибках

Каждый атрибут валидации позволяет задать кастомное сообщение об ошибке через параметр ErrorMessage. В сообщении можно использовать плейсхолдеры: {0} для имени поля, {1} и {2} для других параметров атрибута.

Автоматическая валидация в Web API

Контроллеры с атрибутом [ApiController] автоматически проверяют ModelState.IsValid и при ошибках возвращают HTTP 400 с деталями в формате ProblemDetails согласно RFC 7807. Это избавляет от необходимости явно проверять ModelState в каждом методе.

Неявная валидация non-nullable типов

При включенных nullable-контекстах (<Nullable>enable</Nullable>) ненулевые ссылочные типы неявно получают атрибут [Required(AllowEmptyStrings = true)]. Это означает, что отсутствие значения для такого свойства вызовет ошибку валидации.

Кастомные атрибуты валидации

Для сложных бизнес-правил можно создавать собственные атрибуты. Для этого нужно унаследоваться от базового класса ValidationAttribute и переопределить метод IsValid .

JSON-имена свойств в ошибках

По умолчанию ошибки валидации используют C#-имена свойств. Для SPA-приложений удобнее использовать JSON-имена (camelCase). Это настраивается через SystemTextJsonValidationMetadataProvider.

Задание

Цель работы: Реализовать комплексную систему валидации в REST API приложении, используя встроенные и кастомные атрибуты, а также настроить формат возвращаемых ошибок.

Задачи:

1. Модификация DTO с использованием атрибутов валидации.
2. Настройка кастомных сообщений об ошибках.
3. Создание кастомного атрибута валидации.
4. Создание кастомного атрибута с зависимостью от другого свойства.
5. Настройка использования JSON-имен свойств в ошибках.
6. Обработка неявной валидации non-nullable типов.
7. Тестирование валидации.

Контрольные вопросы

1. Что такое ModelState и какую информацию он содержит? Как проверить, успешно ли прошла валидация?
2. Какие встроенные атрибуты валидации вы знаете? Для чего используется каждый из них?
3. Как задать кастомное сообщение об ошибке для атрибута валидации? Какие плейсхолдеры можно использовать?
4. Как создать собственный атрибут валидации? Какой метод нужно переопределить?
5. Как в кастомном атрибуте валидации получить доступ к другим свойствам валидируемого объекта?
6. Как ведут себя ненулевые ссылочные типы при включенных nullable-контекстах? Почему отсутствие значения для них вызывает ошибку валидации?
7. Как настроить использование JSON-имен свойств (camelCase) в ключах ошибок валидации?
8. Как автоматически возвращается ошибка 400 в контроллерах с атрибутом [ApiController]?

Лабораторная работа №8 «Расширение функционала REST API приложения: добавление фоновых задач»

Теоретическая часть

Назначение фоновых задач

В современных веб-приложениях часто возникают операции, которые не должны выполняться синхронно в рамках HTTP-запроса, так как это может привести к блокировке потока, увеличению времени ответа и ухудшению пользовательского опыта.

К таким операциям относятся:

1. Отправка электронных писем.
2. Генерация отчетов или PDF-файлов.
3. Обработка очередей сообщений.
4. Очистка устаревших данных.
5. Периодическое обновление кэша.
6. Взаимодействие с внешними API.

Фоновые задачи позволяют разгрузить основной поток обработки запросов, повысить отзывчивость приложения и улучшить его масштабируемость.

Размещенные службы (Hosted Services)

В ASP.NET Core фоновые задачи реализуются как размещенные службы (hosted services) – классы с логикой фоновой работы, которые управляются узлом приложения. Основой для создания таких служб служит интерфейс IHostedService, определяющий два метода:

1. StartAsync(CancellationToken) – вызывается при запуске приложения, содержит логику запуска фоновой задачи. Должен выполняться быстро, так как последующие службы ожидают его завершения.
2. StopAsync(CancellationToken) – вызывается при graceful shutdown приложения, содержит логику корректного завершения фоновой задачи. По умолчанию имеет 30-секундный таймаут.

Базовый класс BackgroundService

Начиная с .NET Core 2.1, появился удобный абстрактный базовый класс BackgroundService, реализующий IHostedService. Для создания фоновой задачи достаточно унаследоваться от этого

класса и переопределить метод ExecuteAsync(CancellationToken stoppingToken), в котором размещается основная логика.

Важно понимать, что фоновость реализуется не за счет выделения отдельного потока, а благодаря асинхронной модели TAP. Метод ExecuteAsync должен немедленно возвращать Task, а его выполнение продолжается в фоне. Бесконечные циклы внутри ExecuteAsync обязательно должны быть асинхронными (содержать await), иначе они навсегда заблокируют поток, в котором были запущены, и приложение зависнет.

Типы фоновых задач

Существует несколько распространенных типов фоновых задач:

1. Периодические задачи (Timed tasks) – выполняются через заданные интервалы времени с использованием System.Threading.Timer.
2. Задачи с областью действия (Scoped tasks) – требуют доступа к сервисам с ограниченным временем жизни (например, DbContext в Entity Framework Core). Для их использования необходимо создавать область видимости (scope) внутри фоновой службы.
3. Очереди фоновых задач (Queued tasks) – позволяют динамически добавлять задачи в очередь и выполнять их последовательно.
4. Фоновые службы воркера (Worker Service) – отдельные приложения, предназначенные только для выполнения фоновых задач, без веб-интерфейса. Создаются на основе шаблона Worker Service .

Обработка ошибок в фоновых задачах

Исключения в фоновых задачах не обрабатываются автоматически глобальным middleware, как в контроллерах. Их необходимо перехватывать внутри метода ExecuteAsync с помощью try-catch и логировать. При отмене задачи через CancellationToken следует корректно завершать выполнение, обрабатывая OperationCanceledException.

Сравнение с профессиональными инструментами

Встроенные BackgroundService подходят для простых и средних по сложности фоновых задач. Для более сложных

сценариев (персистентность, повторные попытки, мониторинг) существуют специализированные библиотеки:

1. Hangfire – добавляет хранение заданий в БД, повторные попытки, веб-панель управления.
2. Quartz.NET – мощный планировщик для enterprise-задач со сложными cron-расписаниями и поддержкой кластеризации.

Задание

Цель работы: Расширить существующее REST API приложение (из предыдущих лабораторных работ) функциональностью для выполнения фоновых задач с использованием BackgroundService, реализовав периодическую очистку данных и фоновую обработку длительных операций.

Задачи:

1. Реализация периодической фоновой задачи (Timed task).
2. Реализация фоновой задачи с областью действия (Scoped task).
3. Реализация очереди фоновых задач (опционально/повышенной сложности).
4. Регистрация фоновых служб.
5. Обработка ошибок и корректное завершение.

Контрольные вопросы

1. Для чего нужны фоновые задачи в веб-приложениях? Приведите примеры.
2. Что такое размещенные службы (hosted services) и какие интерфейсы лежат в их основе?
3. В чем разница между интерфейсом IHostedService и абстрактным классом BackgroundService? Когда удобнее использовать каждый из них?
4. Почему бесконечный цикл в ExecuteAsync обязательно должен содержать await? Что произойдет, если этого не сделать?
5. Как создать периодическую задачу, выполняющуюся каждый час?
6. Почему в фоновой задаче нельзя напрямую использовать сервисы с областью действия (scoped services), например DbContext, и как правильно организовать их использование?
7. Как обрабатывать исключения в фоновых задачах, чтобы они не приводили к остановке всего приложения?

8. Какие существуют профессиональные библиотеки для работы с фоновыми задачами и в чем их преимущества перед встроенными средствами?
9. Как корректно завершить фоновую задачу при остановке приложения? Какую роль играет CancellationToken?

Лабораторная работа №9 «Расширение функционала REST API приложения: добавление аутентификации и авторизации, создание ролевой системы»

Теоретическая часть

Аутентификация и авторизация: основные понятия

Веб-безопасность строится на двух фундаментальных процессах:

1. Аутентификация (Authentication) – процесс проверки подлинности пользователя, ответ на вопрос "Кто вы?".
2. Система проверяет учетные данные (логин/пароль, токен) и устанавливает личность пользователя.
3. Авторизация (Authorization) – процесс определения прав доступа, ответ на вопрос "Что вам разрешено делать?". После идентификации система определяет, к каким ресурсам и операциям пользователь имеет доступ.

JWT (JSON Web Token)

JWT – это открытый стандарт (RFC 7519) для безопасной передачи информации между сторонами в виде JSON-объекта. Токен состоит из трех частей, разделенных точками :

1. Header (Заголовок) – содержит метаданные: тип токена (JWT) и алгоритм подписи (например, HS256).
2. Payload (Полезная нагрузка) – содержит утверждения (claims) о пользователе (идентификатор, имя, роль, срок действия и т. д.). Эта часть только кодируется (base64), но не шифруется, поэтому не должна содержать конфиденциальных данных.
3. Signature (Подпись) – создается путем хеширования заголовка и полезной нагрузки с использованием секретного ключа. Подпись гарантирует, что токен не был изменен.

Преимущества JWT для API:

1. Stateless (Отсутствие состояния): Серверу не нужно хранить сессии, токен содержит всю необходимую информацию. Это упрощает масштабирование приложения.
2. Безопасность: Криптографическая подпись предотвращает подделку токена.
3. Гибкость: Токен может содержать любые данные (claims), необходимые для авторизации.

4. Ролевая модель доступа (RBAC – Role-Based Access Control).
5. RBAC – это подход к авторизации, при котором права доступа группируются по ролям, а пользователям назначаются эти роли. Роли представляют собой широкие категории разрешений, например, "Admin", "User", "Manager". Простота реализации и понимания делает этот подход популярным для большинства приложений.

Более гибкие подходы к авторизации

По мере усложнения приложений ролевая модель может стать недостаточно гибкой. Альтернативные подходы:

1. Claims-Based Authorization (Авторизация на основе утверждений): Использует утверждения (claims) – пары "ключ-значение", предоставляющие более детальную информацию о пользователе (например, "department": "Sales", "country": "US").
2. Policy-Based Authorization (Авторизация на основе политик): Самый мощный и гибкий подход. Политики – это многократно используемые правила, которые комбинируют роли, утверждения и даже пользовательскую логику для определения сложных требований доступа.

ASP.NET Core Identity

ASP.NET Core Identity – это встроенная система membership, которая добавляет функциональность логина, регистрации, управления пользователями, ролями, хеширования паролей и многого другого. По умолчанию Identity использует Entity Framework Core для хранения данных в базе данных.

Схема работы JWT в Web API

Клиент отправляет запрос на /api/auth/login с учетными данными.

Сервер проверяет учетные данные. Если они верны, сервер генерирует JWT-токен, содержащий claims (включая роли), и отправляет его клиенту.

Клиент сохраняет токен (обычно в localStorage или sessionStorage) и включает его в заголовок Authorization: Bearer {токен} при каждом последующем запросе к защищенным ресурсам.

Сервер принимает запрос, валидирует токен (проверяет подпись, срок действия), извлекает claims из токена и использует их для авторизации доступа.

Задание

Цель работы: Реализовать в REST API приложении (из предыдущих лабораторных работ) систему аутентификации на основе JWT и авторизации на основе ролей (RBAC) с использованием ASP.NET Core Identity.

Задачи:

1. Подготовка проекта.
2. Модификация модели данных.
3. Регистрация Identity и настройка JWT.
4. Создание моделей DTO.
5. Создание контроллера для аутентификации (AuthController).
6. Создание метода для генерации JWT.
7. Реализация ролевой авторизации в контроллерах.
8. Тестирование.

Контрольные вопросы

1. В чем принципиальная разница между аутентификацией и авторизацией?
2. Что такое JWT и из каких частей он состоит? Какая часть токена гарантирует его целостность?
3. Какие преимущества дает использование JWT для аутентификации в Web API по сравнению с классическими сессиями?
4. Что такое ASP.NET Core Identity и какие задачи он решает?
5. Как в JWT включить информацию о ролях пользователя? Как затем использовать эти роли для ограничения доступа к эндпоинтам?
6. Что произойдет, если срок действия (exp) JWT истек? Как сервер это обрабатывает?
7. Какие существуют более гибкие альтернативы ролевой авторизации (RBAC) и в каких случаях их стоит применять?

Лабораторная работа №10 «Создание клиентского приложения для работы с публичным WebSocket»

Теоретическая часть

Протокол WebSocket

WebSocket – это протокол, обеспечивающий двустороннюю (full-duplex) связь между клиентом и удаленным сервером через одно TCP-соединение. В отличие от традиционного HTTP, где клиент всегда инициирует запрос, WebSocket позволяет серверу отправлять данные клиенту без предварительного запроса, что делает его идеальным для приложений реального времени.

Сценарии использования WebSocket:

1. Чаты и мессенджеры.
2. Биржевые котировки и финансовые данные.
3. Онлайн-игры.
4. Совместная работа с документами.
5. Уведомления в реальном времени.

Класс ClientWebSocket в .NET

В C# для работы с WebSocket-клиентами используется класс ClientWebSocket из пространства имен System.Net.WebSockets. Этот класс предоставляет возможность установить подключение WebSocket через открывающее рукопожатие, которое создается и отправляется методом ConnectAsync.

Основные методы ClientWebSocket (таблица 4):

Таблица 4

Метод	Назначение
`ConnectAsync`	Устанавливает соединение с WebSocket-сервером
`SendAsync`	Отправляет данные на сервер
`ReceiveAsync`	Получает данные от сервера
`CloseAsync`	Закрывает соединение с сервером
`CloseOutputAsync`	Закрывает исходящий поток данных

Важные особенности:

1. Одновременно поддерживается только одна операция отправки и одна операция получения на каждом экземпляре ClientWebSocket.

2. Нельзя выполнять несколько отправок или несколько получений одновременно без синхронизации.
3. Все операции должны выполняться последовательно (необходимо дождаться завершения предыдущей).

WebSocket-соединения через прокси и SSL

ClientWebSocket поддерживает:

1. Защищенные соединения (wss://) через SSL/TLS.
2. HTTP-прокси.
3. Кастомные заголовки.
4. Куки.
5. Клиентские сертификаты.

Сжатие сообщений

Начиная с .NET Core 3.1, ClientWebSocket поддерживает сжатие сообщений согласно RFC 7692 (per-message deflate). Однако следует помнить, что включение сжатия делает приложение уязвимым для атак типа CRIME/BREACH, поэтому не рекомендуется сжимать данные, содержащие секреты.

Keep-Alive механизмы

Начиная с .NET 9, доступны две стратегии Keep-Alive :

1. Unsolicited PONG (по умолчанию) – клиент регулярно отправляет PONG-фреймы независимо от активности.
2. PING/PONG – клиент отправляет PING после периода неактивности и ожидает PONG; если ответ не получен в течение таймаута, соединение прерывается.

Задание

Цель работы: Разработать клиентское консольное приложение на C#, которое подключается к публичному WebSocket-серверу, отправляет запросы и обрабатывает получаемые данные в реальном времени.

Вариант задания: "Клиент для публичного WebSocket API WhiteBIT"

Биржа WhiteBIT предоставляет публичный WebSocket API для получения рыночных данных в реальном времени. Мы создадим клиент, который будет подключаться к этому API и получать обновления цен криптовалют.

Задачи:

1. Изучение документации API.
2. Создание моделей данных.
3. Реализация WebSocket-клиента.
4. Обработка Keep-Alive.
5. Подписка на обновления.
6. Обработка ошибок и переподключение.

Контрольные вопросы

1. Что такое протокол WebSocket и чем он отличается от HTTP? Какие преимущества он дает для приложений реального времени?
2. Какой класс в .NET используется для создания WebSocket-клиента? Опишите основные методы этого класса.
3. Почему важно соблюдать последовательность операций при использовании ClientWebSocket? Что произойдет при одновременной отправке нескольких сообщений?
4. Что такое JSON-RPC и как он используется в WebSocket API биржи WhiteBIT?
5. Для чего нужен механизм Keep-Alive в WebSocket? Какие стратегии Keep-Alive поддерживаются в .NET?
6. Как обрабатываются защищенные WebSocket-соединения (wss://)? Какие требования предъявляются к сертификатам?
7. Что такое сжатие сообщений в WebSocket? Какие риски безопасности связаны с его использованием?
8. Как правильно обрабатывать закрытие соединения и реализовать переподключение?

Лабораторная работа №11 «Создание серверного приложения для работы по WebSocket протоколу»

Теоретическая часть

Протокол WebSocket на стороне сервера

WebSocket – это протокол, обеспечивающий полнодуплексную (двустороннюю) связь через одно TCP-соединение. В отличие от HTTP, где клиент всегда инициирует запрос, WebSocket позволяет серверу отправлять данные клиенту без предварительного запроса, что делает его идеальным для приложений реального времени.

Процесс установки соединения (Handshake)

WebSocket-соединение начинается с HTTP-рукопожатия (handshake):

1. Клиент отправляет GET-запрос с заголовком Upgrade: websocket.
2. Сервер должен проверить наличие этого заголовка.
3. Сервер извлекает значение заголовка Sec-WebSocket-Key.
4. Сервер конкатенирует этот ключ со специальным GUID 258EAFA5-E914-47DA-95CA-C5AB0DC85B11.
5. Вычисляет SHA-1 хеш и кодирует его в Base64.
6. Возвращает ответ с заголовком Sec-WebSocket-Accept, содержащим этот хеш.

Формат WebSocket-фрейма

После установки соединения данные передаются в виде фреймов (таблица 5):

Таблица 5

Байт	Биты	Назначение
1	FIN (1 бит)	Указывает, является ли это последним фреймом сообщения
1	RSV1-3 (3 бита)	Зарезервированы для расширений (должны быть 0)
1	Opcode (4 бита)	Тип фрейма (0×1 = текст, 0×2 = бинарные, 0×8 = закрытие, 0×9 = ping, 0×A = pong)
2	MASK (1 бит)	Указывает, замаскированы ли данные (клиент → сервер: всегда 1)
2	Payload length (7 бит)	Длина данных (если 126, то следующие 2 байта; если 127, то следующие 8 байт)
3–6	Masking key (4 байта)	Ключ для декодирования данных (только если MASK = 1)
7+	Payload data	Собственно данные

Подходы к реализации WebSocket-сервера в C#

В .NET существует несколько подходов к созданию WebSocket-сервера:

1. ASP.NET Core WebSocket middleware – встроенная поддержка WebSocket в ASP.NET Core через app.UseWebSockets(). Это наиболее простой и рекомендуемый подход для веб-приложений.
2. Низкоуровневый TcpListener – реализация WebSocket протокола "с нуля" с использованием TcpListener и ручной обработкой handshake и фреймов. Дает полный контроль, но требует больше кода.
3. SignalR – высокоуровневая библиотека, абстрагирующая детали WebSocket и предоставляющая удобную модель удаленного вызова процедур.

Управление несколькими подключениями

Для работы с множеством клиентов необходимо:

1. Хранить активные подключения в потокобезопасной коллекции (например, ConcurrentDictionary<string, WebSocket>).
2. При подключении добавлять клиента в коллекцию.
3. При отключении удалять клиента.
4. Для широковещательной рассылки (broadcast) проходиться по всем активным подключениям.
5. Обработка ошибок и корректное закрытие.

WebSocket-соединения могут быть нестабильными из-за сетевых условий. Необходимо:

1. Проверять result.CloseStatus в цикле получения сообщений для обнаружения закрытия соединения.
2. Вызывать webSocket.CloseAsync() для корректного завершения handshake закрытия.
3. Удалять клиента из коллекции при закрытии.

Задание

Цель работы: Разработать серверное приложение на C# с использованием ASP.NET Core, которое реализует WebSocket-сервер для обработки подключений, приема и рассылки сообщений, а также управления подключенными клиентами.

Вариант задания: "Чат-сервер с поддержкой комнат"
Необходимо реализовать WebSocket-сервер для чата.
Задачи:

1. Создание проекта ASP.NET Core Web API.
2. Настройка WebSocket middleware.
3. Обработка handshake.
4. Разработка протокола сообщений.
5. Управление подключениями.
6. Обработка входящих сообщений.
7. Широковещательная рассылка (broadcast).
8. Обработка закрытия соединения.

Контрольные вопросы

1. Как происходит процесс handshake при установке WebSocket-соединения? Какие заголовки участвуют в этом процессе?
2. Из каких частей состоит WebSocket-фрейм? Что означают поля FIN, Opcode, MASK?
3. Почему сообщения от клиента к серверу всегда маскируются (MASK = 1)?
4. Как правильно организовать хранение и управление несколькими WebSocket-подключениями на сервере?
5. Как обнаружить, что клиент закрыл соединение? Что такое close handshake?
6. Какие существуют способы реализации WebSocket-сервера в .NET? В чем преимущества и недостатки каждого?
7. Что такое Keep-Alive в контексте WebSocket и зачем он нужен?
8. Чем WebSocket отличается от HTTP и для каких задач он лучше подходит?

Лабораторная работа №12 «Создание микросервисного приложения с взаимодействием по REST»

Теоретическая часть

Микросервисная архитектура (MSA)

Микросервисная архитектура – это подход к разработке программного обеспечения, при котором приложение строится как набор небольших, независимо развертываемых сервисов. Каждый микросервис:

1. Реализует определенную бизнес-функциональность.
2. Имеет собственную базу данных (если требуется).
3. Может быть написан на любом языке программирования.
4. Взаимодействует с другими сервисами через сетевые протоколы.

Сравнение с монолитной архитектурой (таблица 6):

Таблица 6

Критерий	Монолит	Микросервисы
Развертывание	Единое	Независимое
Масштабирование	Все приложение	Отдельные сервисы
Технологический стек	Единый	Разные технологии
Сложность разработки	Ниже для малых проектов	Выше, но лучше для больших
Отказоустойчивость	Один сбой останавливает все	Сбой изолирован

Синхронное взаимодействие между микросервисами

В микросервисной архитектуре сервисы должны обмениваться данными. Существует два основных подхода:

1. Синхронный (REST, gRPC) – сервис делает запрос и ждет ответа.
2. Асинхронный (Message brokers) – сервис отправляет сообщение и продолжает работу.

Синхронная коммуникация через REST API – наиболее популярный способ взаимодействия. Она работает как телефонный звонок: вы звоните и ждете, пока собеседник ответит.

Преимущества синхронной коммуникации:

1. Простая реализация (REST – это стандарт).

2. Прямой доступ к данным.
3. Немедленный ответ.
4. Привычность для разработчиков.

Недостатки:

1. Временная зависимость: если сервис B недоступен, сервис A "зависает".
2. Проблемы с масштабированием при высокой нагрузке.
3. Повышенная задержка при последовательных вызовах.

Паттерн "API Gateway"

API Gateway – это единая точка входа для всех клиентов (веб-приложений, мобильных приложений и т. д.), которая маршрутизирует запросы к соответствующим микросервисам.

Функции API Gateway:

1. Маршрутизация запросов.
2. Балансировка нагрузки.
3. Аутентификация и авторизация.
4. Ограничение скорости (rate limiting).
5. Сбор метрик и логирование.
6. Трансформация протоколов.

RESTful API в микросервисах

REST (Representational State Transfer) – архитектурный стиль, использующий стандартные HTTP-методы для работы с ресурсами.

Основные принципы REST API в микросервисах:

1. Ресурсы идентифицируются URI (например, /api/products)
   HTTP-методы определяют операции:
   - GET – получение ресурса или списка.
   - POST – создание нового ресурса.
   - PUT – полное обновление ресурса.
   - PATCH – частичное обновление.
   - DELETE – удаление ресурса.
2. Использование существительных в URI (не глаголов): /orders, а не /getOrders.
3. Множественное число для коллекций: /products, а не /product.
4. Использование HTTP-статусов для индикации результата.

Пример REST-эндпоинтов для сущности (таблица 7):

Таблица 7

Метод	URI	Описание
GET	`/api/items`	Список всех записей
GET	`/api/items/{uuid}`	Детали конкретной записи
POST	`/api/items`	Создание новой записи
PUT	`/api/items/{uuid}`	Обновление записи
DELETE	`/api/items/{uuid}`	Удаление записи

Связи между сущностями в разных сервисах

В микросервисной архитектуре связанные сущности могут находиться в разных сервисах. Например:

1. Сервис проектов – хранит данные о проектах.
2. Сервис задач – хранит задачи, ссылающиеся на проекты.

Для получения полной информации (например, задача + данные проекта) необходимо:

1. Получить задачу из сервиса задач.
2. По projectUuid сделать HTTP-запрос к сервису проектов.
3. Объединить данные и вернуть клиенту.

Docker и Docker Compose для микросервисов

Docker позволяет упаковать каждый микросервис в отдельный контейнер.

Docker Compose – инструмент для определения и запуска многоконтейнерных приложений. В одном файле docker-compose.yml описываются:

1. Все сервисы (микросервисы).
2. Их сетевые настройки.
3. Зависимости между сервисами.
4. Проброшенные порты.

Задание

Цель работы: разработать микросервисное приложение, состоящее из двух сервисов с полным набором CRUD-операций для связанных сущностей, реализовать синхронное взаимодействие между ними по REST, а также настроить единую точку входа (API Gateway) с помощью nginx.

Вариант задания: "Управление проектами и задачами"

Необходимо реализовать два микросервиса:

1. Сервис проектов (ProjectService) – управляет проектами.
2. Сервис задач (TaskService) – управляет задачами, связанными с проектами.

Связь: один проект может содержать множество задач (отношение «один ко многим»).

Задачи:

1. Создание микросервисов.
2. Реализация REST API.
3. Реализация синхронного обмена.
4. Настройка API Gateway на nginx.
5. Контейнеризация с Docker.
6. Тестирование.

Контрольные вопросы

1. Что такое микросервисная архитектура? Каковы ее основные принципы? В чем ее отличия от монолитной архитектуры?
2. Какие существуют способы взаимодействия между микросервисами? В чем разница между синхронным и асинхронным взаимодействием?
3. Что такое API Gateway и зачем он нужен в микросервисной архитектуре?
4. Как настроить проксирование запросов с помощью nginx? Приведите пример конфигурации.
5. Как в .NET выполнить HTTP-запрос к другому микросервису? Какие классы для этого используются?
6. Какие преимущества дает использование REST API для взаимодействия между микросервисами?
7. Как правильно проектировать REST-эндпоинты для ресурсов? Какие HTTP-методы и статус-коды следует использовать?
8. Что такое обогащение (enrichment) данных и как оно реализуется в микросервисной архитектуре?
9. Как с помощью Docker Compose организовать запуск многоконтейнерного приложения?
10. Какие проблемы могут возникнуть при синхронном взаимодействии микросервисов и как их можно решить?

Лабораторная работа №13 «Создание микросервисного приложения с взаимодействием по gRPC»

Теоретическая часть

Что такое gRPC?

gRPC – это современный, высокопроизводительный RPC (Remote Procedure Call) фреймворк с открытым исходным кодом, разработанный Google. Он позволяет клиентским приложениям напрямую вызывать методы на серверном приложении, как если бы это был локальный объект, что упрощает создание распределенных приложений и микросервисов.

Почему gRPC подходит для микросервисов?

В современной микросервисной архитектуре разные сервисы могут быть написаны на разных языках программирования. Для эффективного взаимодействия им нужен общий, быстрый и надежный протокол обмена данными. Традиционный подход с REST и JSON имеет ряд ограничений: текстовый формат данных создает избыточность, а HTTP/1.1 не позволяет эффективно использовать одно соединение.

gRVC решает эти проблемы, предлагая:

1. Высокую производительность: бинарная сериализация Protocol Buffers (protobuf) обеспечивает высокую скорость и компактность сообщений (до 8 раз быстрее JSON).
2. Строгие контракты: сервисы определяются в .proto файлах, что гарантирует типовую безопасность и предотвращает ошибки во время выполнения.
3. Поддержку потоковой передачи: gRPC поддерживает серверный, клиентский и двунаправленный стриминг.
4. Кроссплатформенность: gRPC работает с множеством языков (C#, Java, Go, Python, и др.), обеспечивая бесшовное взаимодействие между микросервисами на разных технологиях.

Protocol Buffers (protobuf)

Protocol Buffers – это язык определения интерфейса (IDL) от Google для сериализации структурированных данных . Он является основой gRPC. Вы определяете структуру данных в файле .proto, а затем компилятор protoc генерирует код на нужном языке для чтения и записи этих данных.

gRPC vs REST: Сравнение (таблица 8).

Таблица 8

Характеристика	gRPC	REST
Протокол	HTTP/2	HTTP/1.1 (обычно)
Формат данных	Protocol Buffers (бинарный)	JSON, XML (текстовый)
Производительность	Высокая	Средняя
Поддержка браузеров	Ограничена (требуется gRPC-Web)	Полная
Стриминг	Встроенный (одно соединение)	Ограничен (Server-Sent Events, WebSocket)
Генерация кода	Автоматическая из `.proto`	Ручная или через OpenAPI

Типы взаимодействия в gRPC

gRPC поддерживает четыре режима вызова:

1. Унарный RPC (Unary RPC): клиент отправляет один запрос и получает один ответ.
2. Серверный стриминг (Server streaming): клиент отправляет один запрос, а сервер возвращает поток сообщений.
3. Клиентский стриминг (Client streaming): клиент отправляет поток сообщений и получает один ответ.
4. Двунаправленный стриминг (Bidirectional streaming): и клиент, и сервер отправляют потоки сообщений независимо друг от друга.

Задание

Цель работы: разработать микросервисное приложение, состоящее из двух сервисов (сервер и клиент), которые взаимодействуют друг с другом по протоколу gRPC, и реализовать несколько типов gRPC-вызовов.

Вариант задания: "Система обработки заказов"

Необходимо реализовать gRPC-сервер (ProductService), управляющий каталогом товаров, и gRPC-клиент (OrderService), который при создании заказа запрашивает информацию о товаре у ProductService.

Задачи:

1. Создание gRPC-сервера (ProductService).
2. Реализация логики сервера.

3. Создание gRPC-клиента (OrderService).
4. Реализация различных типов вызовов в клиенте.
5. Обработка ошибок.
6. Тестирование.

Контрольные вопросы

1. Что такое gRPC и какие преимущества он дает для микросервисной архитектуры по сравнению с REST?
2. Что такое Protocol Buffers (protobuf) и какова их роль в gRPC?
3. Из каких частей состоит .proto файл? Что такое теги полей (field tags)?
4. Какие типы RPC-вызовов поддерживает gRPC? Опишите каждый из них.
5. Как в .NET создать gRPC-сервер и зарегистрировать его в конвейере обработки запросов?
6. Как создать gRPC-клиент в .NET и вызвать удаленный метод? Что такое GrpcChannel?
7. Как обрабатываются ошибки в gRPC? Что такое RpcException и какие статусы она содержит?
8. Какие существуют стратегии обеспечения безопасности gRPC-соединений (например, TLS)?

Лабораторная работа №14 «Создание микросервисного приложения с взаимодействием через брокера сообщений (consumer, producer)»

Теоретическая часть

Асинхронное взаимодействие в микросервисной архитектуре

В микросервисной архитектуре сервисы должны обмениваться данными. Существует два основных подхода к взаимодействию (таблица 9):

Таблица 9

Характеристика	Синхронное (REST/gRPC)	Асинхронное (Message Broker)
Связь	Запрос-ответ	Событийная, через очередь
Зависимость	Высокая (клиент ждет ответ)	Низкая (отправитель не ждет)
Отказоустойчивость	Низкая (сбой получателя → ошибка)	Высокая (сообщения хранятся в очереди)
Масштабирование	Сложное	Горизонтальное (несколько потребителей)
Производительность	Может снижаться при нагрузке	Стабильна за счет буферизации

Что такое брокер сообщений?

Брокер сообщений (Message Broker) – это промежуточный программный компонент, который обеспечивает обмен данными между различными приложениями или сервисами. Он выступает в роли посредника: получает сообщения от отправителей (producers) и доставляет их получателям (consumers).

Основные функции брокера:

1. Буферизация: временное хранение сообщений, если потребитель временно недоступен.
2. Маршрутизация: доставка сообщений нужным потребителям на основе правил.
3. Трансформация: изменение формата сообщений при необходимости.
4. Гарантии доставки: обеспечение надежности (сообщение будет доставлено хотя бы один раз).

RabbitMQ: популярный брокер сообщений

RabbitMQ – это один из самых популярных брокеров сообщений с открытым исходным кодом, реализующий протокол AMQP (Advanced Message Queuing Protocol).

Ключевые концепции RabbitMQ:

1. Producer (Производитель) – приложение, которое отправляет (публикует) сообщения.
2. Consumer (Потребитель) – приложение, которое получает и обрабатывает сообщения.
3. Queue (Очередь) – буфер, где хранятся сообщения. Очередь привязана к памяти и диску сервера.
4. Exchange (Обменник) – компонент, который получает сообщения от producer'a и направляет их в одну или несколько очередей на основе правил (binding).
5. Binding (Связка) – правило, связывающее exchange с очередью (часто с указанием routing key).
6. Routing Key (Ключ маршрутизации) – атрибут сообщения, который exchange использует для принятия решения о том, в какую очередь направить сообщение.

Типы Exchange:

1. Direct: сообщение доставляется в очередь, у которой routing key полностью совпадает с routing key сообщения.
2. Fanout: сообщение доставляется во все очереди, привязанные к exchange (широковещательная рассылка).
3. Topic: маршрутизация на основе шаблонов routing key (например, *.log).
4. Headers: маршрутизация на основе заголовков сообщения, а не routing key.

Паттерн "Producer-Consumer"

Это фундаментальный паттерн асинхронного обмена:

1. Producer создает сообщение и отправляет его в брокер (не зная о потребителях).
2. Брокер сохраняет сообщение в очереди.
3. Consumer (один или несколько) подписывается на очередь и получает сообщения для обработки.

Конкурирующие потребители (Competing Consumers): если несколько consumer'ов слушают одну очередь, сообщения распределяются между ними (каждое сообщение обрабатывается только

одним consumer'ом). Это позволяет горизонтально масштабировать обработку.

Библиотеки для работы с RabbitMQ в .NET

1. RabbitMQ.Client – официальная низкоуровневая библиотека.
2. MassTransit – высокоуровневая абстракция над RabbitMQ (и другими брокерами), упрощающая разработку.
3. rmq.message.management – библиотека для декларативной настройки очередей и exchange.

Задание

Цель работы: разработать микросервисное приложение, состоящее из двух сервисов, которые взаимодействуют асинхронно через брокер сообщений RabbitMQ. Один сервис выступает в роли producer'a (отправляет сообщения), другой – в роли consumer'a (получает и обрабатывает сообщения).

Вариант задания: "Система уведомлений при создании заказа"

Необходимо реализовать два микросервиса:

1. OrderService (Producer) – сервис для создания заказов. При создании заказа публикует событие "Заказ создан".
2. NotificationService (Consumer) – сервис, который получает события о созданных заказах и имитирует отправку уведомления (например, логирует в консоль).

Контрольные вопросы

1. В чем разница между синхронным (REST) и асинхронным (Message Broker) взаимодействием микросервисов? Каковы преимущества и недостатки каждого подхода?
2. Что такое брокер сообщений? Какую роль он играет в микросервисной архитектуре?
3. Опишите основные компоненты RabbitMQ: Producer, Consumer, Queue, Exchange, Binding, Routing Key.
4. Какие типы exchange существуют в RabbitMQ (Direct, Fanout, Topic) и для каких сценариев они подходят?
5. Как настроить и запустить RabbitMQ локально с помощью Docker?
6. Что такое паттерн "Конкурирующие потребители" (Competing Consumers) и как он помогает в масштабировании?

7. Какие существуют гарантии доставки сообщений (at most once, at least once, exactly once) и как они реализуются в RabbitMQ?
8. Что такое ACK (подтверждение) в RabbitMQ и почему важно его обрабатывать?

Лабораторная работа №15 «Настройка конфигурации REST API приложения (порт, хост, данные для подключения к источнику данных, приватные ключи)»

Теоретическая часть

Модель конфигурации в ASP.NET Core

Современные приложения работают в разных средах: на компьютере разработчика, в тестовом окружении, на production-сервере. В каждом окружении могут быть свои настройки: адреса баз данных, ключи API, порты и хосты. В ASP.NET Core реализована гибкая и иерархическая система конфигурации, которая позволяет загружать настройки из множества источников.

При создании приложения с помощью WebApplication.CreateBuilder(args) автоматически настраиваются следующие провайдеры конфигурации в указанном порядке:

1. appsettings.json – файл с настройками по умолчанию.
2. appsettings.{Environment}.json – файл, специфичный для текущего окружения (например, appsettings.Development.json). Переопределяет настройки из основного файла.
3. User Secrets (Секреты пользователя) – для хранения конфиденциальных данных в процессе разработки (подключается только в среде Development).
4. Переменные окружения (Environment Variables) – стандартный способ настройки в контейнерах и облачных средах.
5. Аргументы командной строки (Command-line arguments) – имеют наивысший приоритет и переопределяют все предыдущие источники.

Этот механизм позволяет реализовать принцип иерархичности и переопределения: значения из источников, расположенных ниже по списку, заменяют значения из вышестоящих источников.

Типы конфигурационных данных

В REST API можно выделить несколько категорий настроек:

1. Параметры хоста и порта: определяют, по какому адресу и порту будет доступно приложение.
2. Строки подключения к базе данных (Connection Strings): содержат адрес сервера БД, имя базы, учетные данные.
3. Приватные ключи и секреты: ключи для подписи JWT, пароли, ключи доступа к внешним API.

4. Параметры приложения: специфические для бизнес-логики настройки (например, размер страницы при пагинации, лимиты запросов).

Безопасность конфигурации: секреты

Никогда не следует хранить пароли, ключи API или строки подключения с учетными данными прямо в коде или в файлах appsettings.json, которые попадают в систему контроля версий (Git).

Для безопасного хранения таких данных в .NET предусмотрены:

1. Менеджер секретов (Secret Manager) – инструмент для разработки. Он хранит секреты в JSON-файле вне структуры проекта (в профиле пользователя), что предотвращает их случайную отправку в репозиторий.
2. Переменные окружения – предпочтительный способ передачи секретов в production-среде, особенно в контейнерных средах (Docker, Kubernetes).
3. Специализированные хранилища (Azure Key Vault, HashiCorp Vault) – для корпоративных систем с высокими требованиями к безопасности.

Конфигурация Kestrel

Kestrel – это кроссплатформенный веб-сервер ASP.NET Core. Его поведение, включая прослушиваемые адреса и порты, можно настраивать.

Существует несколько способов задать URL для приложения:

1. В launchSettings.json – используется только для локальной разработки и отладки.
2. Переменные окружения: ASPNETCORE_URLS для задания списка URL или ASPNETCORE_HTTP_PORTS / ASPNETCORE_HTTPS_PORTS для указания только портов.
3. В appsettings.json – через секцию Kestrel.
4. В коде: через UseUrls() или ConfigureKestrel().

Задание

Цель работы: Настроить конфигурацию существующего REST API приложения (из предыдущих работ) для его гибкого и безопасного развертывания в различных средах.

Задачи:

1. Анализ и реорганизация конфигурации.

2. Настройка хоста и портов.
3. Настройка подключения к базе данных.
4. Настройка JWT-аутентификации (или другого ключа).
5. Создание модели для строго типизированных настроек.
6. Использование переменных окружения для production.

Контрольные вопросы

1. Какие источники конфигурации использует ASP.NET Core по умолчанию и в каком порядке они применяются? Что произойдет, если один и тот же ключ задан в нескольких источниках?
2. Для чего нужны файлы appsettings.Development.json и appsettings.Production.json? Как приложение понимает, какой файл загрузить?
3. Что такое Secret Manager и для решения какой проблемы он предназначен? Как он защищает секреты от попадания в Git?
4. Как задать URL и порты, на которых будет работать приложение? В чем разница между настройками в launchSettings.json и переменной окружения ASPNETCORE_URLS?
5. Как передать сложную конфигурацию (например, объект JwtSettings) через переменные окружения в Docker или Kubernetes? (Вспомните про синтаксис с двойным подчеркивание `__`).
6. В чем преимущество использования строго типизированных классов настроек (Options pattern) перед прямым обращением к IConfiguration в каждом классе?
7. Какие существуют более продвинутые и безопасные способы хранения секретов, чем переменные окружения, для production-среды?

Лабораторная работа №16 «Внедрение логирования в REST API приложение»

Теоретическая часть

Роль логирования в REST API

Логирование – это процесс записи информации о работе приложения. В контексте REST API логирование выполняет несколько критически важных функций:

1. Мониторинг работоспособности: отслеживание доступности API и времени ответа.
2. Отладка: выявление причин ошибок и некорректного поведения.
3. Аудит: фиксация действий пользователей для безопасности и соответствия требованиям.
4. Анализ производительности: выявление узких мест и медленных запросов.

Система логирования в ASP.NET Core

ASP.NET Core предоставляет встроенную, высокопроизводительную систему логирования через интерфейс ILogger<T>.

Она поддерживает:

1. Структурированное логирование – логи хранятся не как простой текст, а как структурированные данные с именованными полями.
2. Множество провайдеров – логи можно отправлять в консоль, файлы, базы данных, облачные сервисы.
3. Фильтрацию по уровням – возможность включать/отключать логи в зависимости от важности.

Архитектура логирования

Система логирования в .NET построена на *трех ключевых компонентах*:

1. ILogger<TCategoryName> – интерфейс для записи логов. Категория обычно соответствует имени класса, в котором используется логгер.
2. Log Level (Уровень логирования) – определяет важность события.
3. Log Providers – компоненты, которые отправляют логи в конкретные места назначения: консоль, файл, базу данных и т. д.

Структурированное логирование и Serilog

Традиционное логирование сохраняет сообщения как простой текст, что затрудняет их анализ и поиск. Структурированное логирование сохраняет данные в формате ключ-значение, что позволяет:

1. Выполнять сложные запросы к логам.
2. Строить дашборды и графики.
3. Быстро находить все события, связанные с конкретным пользователем или запросом.

Serilog – самая популярная библиотека для структурированного логирования в .NET. Она интегрируется со встроенной системой логирования и предоставляет:

1. Синки (Sinks) – назначения для логов (консоль, файл, Seq, Elasticsearch, базы данных).
2. Обогатители (Enrichers) – добавляют контекстную информацию к каждому логу (имя компьютера, ID-потока, имя пользователя).
3. Гибкую конфигурацию – через код или JSON-файлы.

Логирование HTTP-запросов и ответов

Для REST API особенно важно логировать входящие запросы и исходящие ответы. Это можно реализовать через:

1. Middleware – компонент, перехватывающий каждый HTTP-запрос и ответ.
2. Фильтры действий – логирование на уровне контроллеров.

Correlation ID (Сквозной идентификатор)

В микросервисной архитектуре один пользовательский запрос может проходить через несколько сервисов. Correlation ID – это уникальный идентификатор, который добавляется к каждому логу на всех этапах обработки запроса. Это позволяет:

1. Отслеживать путь запроса через всю систему.
2. Собирать все логи, связанные с конкретной операцией.
3. Упрощать диагностику распределенных транзакций.

Задание

Цель работы: Внедрить комплексную систему логирования в REST API приложение с использованием встроенных средств ASP.NET Core и библиотеки Serilog, реализовать логирование

HTTP-запросов и ответов, а также настроить сквозную трассировку с Correlation ID.

Задачи:

1. Настройка встроенного логирования ASP.NET Core.
2. Интеграция Serilog.
3. Логирование HTTP-запросов и ответов.
4. Добавление контекстной информации.
5. Реализация Correlation ID.
6. Настройка логирования ошибок.
7. Тестирование.

Контрольные вопросы

1. Какие уровни логирования существуют в ASP.NET Core и для каких ситуаций каждый из них предназначен?
2. Что такое структурированное логирование и чем оно лучше обычного текстового?
3. В чем преимущества использования Serilog перед встроенными провайдерами логирования?
4. Что такое "синк" (sink) в терминологии Serilog? Приведите примеры популярных синков.
5. Зачем нужен Correlation ID и как он помогает при отладке распределенных систем?
6. Как можно логировать HTTP-запросы и ответы без нарушения безопасности (чтобы не логировать пароли и токены)?
7. Какие существуют способы добавления контекстной информации в логи?
8. Почему важно использовать асинхронные синки в production-окружении?

Лабораторная работа №17 «Упаковка REST API приложения в контейнер и доставка на другое устройство»

Теоретическая часть

Что такое контейнеризация?

Контейнеризация – это метод виртуализации на уровне операционной системы, который позволяет упаковать приложение вместе со всеми его зависимостями в изолированную среду – контейнер. В отличие от виртуальных машин, контейнеры используют ядро хостовой ОС, что делает их легковесными и быстродействующими.

Преимущества контейнеризации:

1. Переносимость: контейнер работает одинаково на любом устройстве с Docker.
2. Изоляция: приложения не конфликтуют между собой.
3. Масштабируемость: легко запустить множество экземпляров.
4. Воспроизводимость: точное воспроизведение окружения разработки на production.
5. Эффективность: минимальные накладные расходы по сравнению с виртуальными машинами.

Docker – платформа для контейнеризации

Docker – это самая популярная платформа для создания, доставки и запуска контейнерных приложений.

Основные компоненты Docker:

1. Docker Engine – среда выполнения контейнеров.
2. Docker Image (Образ) – шаблон для создания контейнеров, включающий приложение и все зависимости.
3. Docker Container (Контейнер) – запущенный экземпляр образа.
4. Docker Hub / Registry – реестр для хранения и распространения образов.
5. Dockerfile – текстовый файл с инструкциями для сборки образа.

Dockerfile – инструкция по сборке

Dockerfile содержит последовательность команд для создания образа. Основные инструкции (таблица 10):

Таблица 10

Инструкция	Назначение	Пример
`FROM`	Базовый образ	`FROM mcr.microsoft.com/dotnet/aspnet:8.0`
`WORKDIR`	Рабочая директория	`WORKDIR /app`
`COPY`	Копирование файлов	`COPY . .`
`RUN`	Выполнение команды при сборке	`RUN dotnet restore`
`EXPOSE`	Указание порта	`EXPOSE 80`
`ENV`	Переменные окружения	`ENV ASPNETCORE_ENVIRONMENT=Production`
`ENTRYPOINT`	Команда при запуске	`ENTRYPOINT ["dotnet", "MyApi.dll"]`

Многоступенчатая сборка (Multi-stage builds)

Многоступенчатая сборка позволяет создавать маленькие и эффективные образы, разделяя процесс на этапы сборки и этап выполнения. Это уменьшает размер финального образа, так как в него попадают только необходимые артефакты.

.dockerignore – исключение файлов

Файл .dockerignore работает аналогично .gitignore и позволяет исключить ненужные файлы из контекста сборки, ускоряя процесс и уменьшая размер образа .

Реестры контейнеров (Container Registries)

Для распространения образов используются реестры:

1. Docker Hub – публичный реестр по умолчанию.
2. GitHub Container Registry (GHCR) – интеграция с GitHub.
3. Azure Container Registry (ACR) – облачный реестр от Microsoft.
4. Amazon ECR – от Amazon Web Services.
5. Приватные реестры – можно развернуть собственный.

Сети и volumes в Docker

Для взаимодействия контейнеров и сохранения данных используются:

1. Docker Networks – виртуальные сети для связи контейнеров.
2. Docker Volumes – постоянное хранилище данных, не зависящее от жизненного цикла контейнера.

Docker Compose

Docker Compose – инструмент для определения и запуска многоконтейнерных приложений. В файле docker-compose.yml описываются:

1. Сервисы (контейнеры)
2. Сети
3. Volumes
4. Зависимости между сервисами

Задание

Цель работы: Упаковать существующее REST API приложение (из предыдущих работ) в Docker-контейнер, оптимизировать образ с помощью многоступенчатой сборки, загрузить его в реестр контейнеров и развернуть на другом устройстве.

Задачи:

1. Подготовка приложения к контейнеризации.
2. Создание Dockerfile.
3. Создание .dockerignore.
4. Локальная сборка и тестирование.
5. Оптимизация образа.
6. Работа с переменными окружения.
7. Загрузка в реестр контейнеров.
8. Доставка на другое устройство.
9. Docker Compose (опционально).

Контрольные вопросы

1. Что такое контейнеризация и чем она отличается от виртуализации? Каковы преимущества контейнеров?
2. Что такое Docker-образ и Docker-контейнер? В чем разница между ними?
3. Для чего нужен Dockerfile? Опишите основные инструкции Dockerfile.
4. Что такое многоступенчатая сборка и зачем она нужна при создании образов для .NET приложений?
5. Как оптимизировать размер Docker-образа для .NET приложения?
6. Для чего нужен файл .dockerignore и что в него обычно включают?
7. Как передать конфигурацию (например, строку подключения к БД) в контейнер во время запуска?
8. Что такое Docker Hub и как загрузить туда образ?
9. Как запустить контейнер с пробросом порта и переменными окружения?

10. Для чего используется Docker Compose и какие задачи он решает?

Лабораторная работа №18 «Добавление SSL сертификата в приложение»

Теоретическая часть

HTTPS и SSL/TLS

HTTPS (Hypertext Transfer Protocol Secure) – это защищенная версия HTTP, использующая шифрование для обеспечения конфиденциальности и целостности данных при передаче между клиентом и сервером.

SSL (Secure Sockets Layer) и его преемник **TLS (Transport Layer Security)** – криптографические протоколы, обеспечивающие защищенную связь поверх TCP. Современные системы используют TLS, хотя термин "SSL" все еще широко распространен.

Зачем нужно HTTPS в REST API?

1. Шифрование данных: защита конфиденциальной информации (пароли, токены, персональные данные).
2. Аутентификация сервера: клиент может быть уверен, что общается именно с вашим сервером.
3. Целостность данных: гарантия, что данные не были изменены при передаче.
4. Требования безопасности: многие API-клиенты (браузеры, мобильные приложения) требуют HTTPS.
5. Соответствие стандартам: GDPR, PCI DSS и другие регуляторные требования.

Таблица 11

Типы SSL-сертификатов

Тип	Описание	Применение
Самоподписанный (Self-signed)	Создан самостоятельно, не доверяется браузерами	Разработка, тестирование
Подписанный CA (Certificate Authority)	Выпущен доверенным центром сертификации	Production
Let's Encrypt	Бесплатные автоматически выдаваемые сертификаты	Production, автоматическое обновление
Wildcard	Действует для домена и всех поддоменов (*.example.com)	Много поддоменов

Как работает HTTPS?

1. Рукопожатие (Handshake).
2. Обмен ключами.
3. Защищенная передача данных.

SSL в ASP.NET Core

ASP.NET Core имеет встроенную поддержку HTTPS. По умолчанию новые проекты настроены на использование HTTPS с сертификатом разработки.

Ключевые компоненты:

1. HTTPS Redirection Middleware (UseHttpsRedirection) – автоматически перенаправляет HTTP-запросы на HTTPS.
2. HSTS Middleware (UseHsts) – добавляет заголовок Strict-Transport-Security.
3. Kestrel – веб-сервер ASP.NET Core, который можно настроить для использования SSL-сертификатов.

Сертификат разработки (Dev Certificate)

.NET Core автоматически создает и доверяет самоподписанному сертификату для разработки. Этот сертификат используется при запуске проекта через Visual Studio или dotnet run.

HSTS (HTTP Strict Transport Security)

HSTS – это механизм, который заставляет браузеры всегда использовать HTTPS при обращении к сайту. Сервер отправляет заголовок Strict-Transport-Security, указывающий браузеру запоминать это требование на определенный срок.

Важно: Включать HSTS в development-окружении не рекомендуется, так как это может заблокировать доступ к приложению при отсутствии HTTPS.

Настройка SSL в Docker-контейнерах

При контейнеризации приложения возникают дополнительные сложности:

1. Сертификат нужно добавить в образ или подключить как volume.
2. Порты 80 и 443 должны быть проброшены.
3. Для production обычно используют обратный прокси (nginx) с терминированием SSL.

Задание

Цель работы: Настроить HTTPS в REST API приложении, используя сертификат разработки для локального тестирования, и затем настроить production-окружение с использованием самоподписанного сертификата или Let's Encrypt.

Задачи:

1. Настройка HTTPS в development-среде.
2. Создание самоподписанного сертификата для production.
3. Настройка Kestrel для использования сертификата.
4. Интеграция с контейнеризацией (из работы №17).
5. Настройка HSTS.
6. Использование Let's Encrypt (опционально).
7. Тестирование.

Контрольные вопросы

1. Чем HTTPS отличается от HTTP и почему важно использовать HTTPS в API?
2. Что такое SSL/TLS и как работает рукопожатие (handshake)?
3. Какие типы SSL-сертификатов существуют и для каких целей они используются?
4. Что такое самоподписанный сертификат и почему браузеры не доверяют ему по умолчанию?
5. Как создать и доверить сертификат разработки в .NET?
6. Что такое HSTS и зачем он нужен? Почему его не рекомендуется включать в development?
7. Как настроить Kestrel для использования SSL-сертификата из файла?
8. Как безопасно хранить пароль от сертификата при развертывании в Docker?
9. Что такое Let's Encrypt и в чем его преимущества?
10. Какие порты используются для HTTP и HTTPS по умолчанию?

Лабораторная работа №19 «Настройка конфигурации безопасности приложения»

Теоретическая часть

Многоуровневая безопасность REST API

Безопасность REST API – это не отдельная функция, а комплекс мер, реализуемых на различных уровнях приложения.

Современный подход к безопасности API включает:

1. Транспортная безопасность – шифрование данных при передаче (HTTPS/TLS).
2. Аутентификация – проверка подлинности клиента (кто?).
3. Авторизация – проверка прав доступа (что можно делать?).
4. Валидация входных данных – защита от инъекций и некорректных данных.
5. Rate Limiting – ограничение частоты запросов.
6. Аудит и логирование – фиксация событий безопасности.
7. CORS – контроль доступа из браузеров.
8. Защита от распространенных атак – XSS, CSRF, DDoS.

Принципы безопасного API

OWASP (Open Web Application Security Project) определяет ключевые *принципы безопасности API*:

1. Минимальные привилегии (Least Privilege) – каждый пользователь/сервис имеет ровно те права, которые необходимы.
2. Глубокая эшелонированная защита (Defense in Depth) – несколько уровней защиты.
3. Безопасность по умолчанию (Secure by Default) – безопасные настройки по умолчанию.
4. Никогда не доверяй пользовательскому вводу (Never Trust User Input) – всегда валидируй и санируй.
5. Безопасное хранение секретов – никаких паролей и ключей в коде.

Rate Limiting (Ограничение частоты запросов)

Rate Limiting защищает API от злоупотреблений и DoS-атак. Основные стратегии:

1. Fixed Window – фиксированное окно времени (например, 100 запросов в минуту).

2. Sliding Window – скользящее окно (более равномерное распределение).
3. Token Bucket – накопление токенов для запросов.
4. Concurrency Limiting – ограничение одновременных запросов

CORS (Cross-Origin Resource Sharing)

CORS – механизм, позволяющий веб-страницам запрашивать ресурсы с другого домена. Безопасная настройка CORS критически важна для API, которые используются в браузерах.

Безопасность аутентификации

Для JWT-аутентификации важны следующие аспекты:

1. Короткое время жизни токена (access token) — обычно 15-30 минут.
2. Refresh tokens – для обновления access token без повторного ввода пароля.
3. Безопасное хранение токенов на клиенте (HttpOnly cookies, не localStorage).
4. Ротация refresh tokens – новый refresh token при каждом обновлении.
5. Отзыв токенов (token revocation) – для компрометированных токенов.

Защита конфиденциальных данных

1. Шифрование – пароли хранятся в захэшированном виде (bcrypt, Argon2).
2. Маскирование – в логах и ответах API не должно быть паролей, токенов, CVV.
3. Токенизация – замена конфиденциальных данных на токены.

Задание

Цель работы: Реализовать комплексную систему безопасности для REST API приложения, включающую защиту от основных веб-уязвимостей, ограничение частоты запросов, настройку CORS, безопасное логирование и защиту конфиденциальных данных.

Задачи:

1. Настройка HTTPS и HSTS.
2. Настройка CORS.
3. Защита аутентификации.
4. Rate Limiting.
5. Валидация входных данных.

6. Безопасное логирование.
7. Защита от XSS и CSRF.
8. Тестирование безопасности.

Контрольные вопросы

1. Какие уровни безопасности существуют в REST API и какие задачи решает каждый из них?
2. Что такое CORS и почему его нужно настраивать для API, используемых в браузерах?
3. Какие существуют стратегии Rate Limiting и как они защищают от DDoS-атак?
4. Чем refresh token отличается от access token и зачем нужны оба?
5. Как защитить API от SQL-инъекций при использовании Dapper (не ORM)?
6. Что такое Content-Security-Policy и от каких атак он защищает?
7. Почему нельзя хранить секреты в коде и как правильно их передавать в production?
8. Какие данные нельзя логировать и как обеспечить их маскировку?
9. Что такое OWASP Top 10 и почему важно его знать разработчику API?
10. Как настроить безопасное хранение паролей в базе данных?

Лабораторная работа №20 «Реализация кэширования данных в REST API приложении»

Теоретическая часть

Что такое кэширование?

Кэширование – это процесс сохранения часто запрашиваемых данных во временном хранилище (кэше) с высокой скоростью доступа, чтобы при повторных запросах можно было быстро получить данные без повторного выполнения дорогостоящих операций.

В контексте REST API кэширование позволяет:

1. Снизить нагрузку на сервер и базу данных.
2. Уменьшить время ответа (латентность) для клиентов.
3. Сэкономить ресурсы (вычислительные, сетевые).
4. Повысить пропускную способность системы.

Таблица 12

Уровни кэширования

Уровень	Где хранится	Скорость	Срок жизни
In-Memory Cache	Память сервера	Очень высокая	Сессия/перезапуск
Distributed Cache	Redis, Memcached	Высокая	Конфигурируемый
Response Cache	Клиент/прокси	Средняя	HTTP-заголовки
Database Cache	СУБД	Средняя	Зависит от БД

HTTP-кэширование

HTTP имеет встроенные механизмы кэширования через заголовки:

1. Cache-Control – директивы для кэширования.
2. ETag – уникальный идентификатор версии ресурса.
3. Last-Modified – дата последнего изменения.

In-Memory Caching в ASP.NET Core

ASP.NET Core предоставляет встроенный механизм кэширования в памяти через интерфейс IMemoryCache. Кэш хранится в памяти процесса и теряется при перезапуске приложения.

Особенности:

1. Очень быстрый доступ.
2. Простая настройка.
3. Ограничен памятью сервера.

4. Не работает в ферме серверов (нераспределенный).

Distributed Caching (Распределенное кэширование)

Для масштабируемых приложений, работающих на нескольких серверах, необходимо распределенное кэширование.

Стратегии инвалидации кэша

Кэш должен обновляться при изменении данных. Основные стратегии:

1. Time-based (TTL) – данные удаляются через заданное время.
2. Explicit invalidation – при изменении данных удаляем связанный кэш.
3. Write-through – обновление кэша при записи в БД.
4. Read-through – автоматическое заполнение при чтении.

Паттерны кэширования

Cache-Aside (Lazy Loading) – самый распространенный паттерн:

1. Проверяем наличие данных в кэше.
2. Если есть – возвращаем.
3. Если нет – загружаем из БД, сохраняем в кэш, возвращаем.

Задание

Цель работы: Реализовать многоуровневое кэширование в REST API приложении с использованием in-memory кэша для часто запрашиваемых данных и распределенного кэша (Redis) для масштабирования, а также настроить HTTP-кэширование для публичных эндпоинтов.

Задачи:

1. Настройка In-Memory Caching.
2. Настройка Response Caching (HTTP-кэширование).
3. Настройка Redis для распределенного кэширования.
4. Инвалидация кэша.
5. Кэширование сложных запросов.
6. Мониторинг и тестирование.

Контрольные вопросы

1. Что такое кэширование и какие проблемы оно решает в REST API?
2. В чем разница между in-memory и distributed caching? Когда что использовать?

3. Какие HTTP-заголовки используются для управления кэшированием? Что означает Cache-Control: no-cache?
4. Что такое ETag и как он помогает в условных запросах?
5. Как работает паттерн Cache-Aside (Lazy Loading)?
6. Какие стратегии инвалидации кэша существуют и в чем их особенности?
7. Почему Redis популярен для распределенного кэширования?
8. Как правильно построить ключ для кэша, учитывающий параметры запроса?
9. Какие данные нельзя кэшировать по соображениям безопасности?
10. Как измерить эффективность кэширования (hit ratio)?

Лабораторная работа №21 «Оптимизация производительности REST API через профилирование»

Теоретическая часть

Что такое профилирование?

Профилирование (Profiling) – это процесс динамического анализа приложения для измерения его производительности, использования памяти, времени выполнения методов и других характеристик. Это диагностический инструмент, который позволяет разработчику понять, какие части кода работают медленно, потребляют много ресурсов или создают узкие места (bottlenecks).

Зачем нужно профилирование REST API?

REST API, работающие под высокой нагрузкой, должны быть оптимизированы. Профилирование помогает:

1. Выявить узкие места – найти методы, которые выполняются дольше всего.
2. Оптимизировать запросы к БД – обнаружить N+1 проблему, медленные запросы.
3. Уменьшить потребление памяти – найти утечки, неэффективные структуры данных.
4. Улучшить время ответа – сократить latency для конечных пользователей.
5. Планировать масштабирование – понять, какие ресурсы нужны под нагрузкой.

Таблица 13

Типы профилирования

Тип	Что измеряет	Инструменты
CPU Profiling	Время выполнения методов	dotTrace, PerfView, Visual Studio Profiler
Memory Profiling	Использование heap, утечки	dotMemory, JetBrains Memory Profiler
Database Profiling	SQL-запросы, время выполнения	SQL Server Profiler, MiniProfiler, EF Core Logging
Async Profiling	Асинхронные операции, deadlocks	Visual Studio, Concurrency Visualizer
Network Profiling	HTTP-запросы, размер ответов	Fiddler, Postman, Browser DevTools

Метрики производительности API

Ключевые метрики для REST API:

1. Latency (Задержка) – время от запроса до ответа (p95, p99).
2. Throughput (Пропускная способность) – запросов в секунду (RPS).
3. Error Rate – процент ошибочных ответов.
4. CPU/Memory Usage – потребление ресурсов сервером.
5. Database Query Time – время выполнения запросов к БД.
6. GC Pause Time – время остановок сборщика мусора.

Паттерны и антипаттерны производительности

Паттерны:

1. Lazy Loading – загрузка данных только когда они нужны.
2. Caching – кэширование часто запрашиваемых данных.
3. Pagination – постраничная выдача данных.
4. Projection – выборка только нужных полей.

Антипаттерны:

1. N+1 Queries – выполнение N дополнительных запросов для коллекции.
2. Cartesian Explosion – разрастание результатов из-за JOIN.
3. Sync-over-Async – блокировка асинхронного кода.
4. String concat in loops – конкатенация строк в циклах.

Задание

Цель работы: Провести комплексное профилирование REST API приложения, выявить узкие места производительности, применить оптимизации и измерить их эффективность.

Задачи:

1. Подготовка тестового окружения.
2. Базовое профилирование.
3. Профилирование запросов к БД.
4. CPU профилирование.
5. Memory профилирование.
6. Асинхронное программирование.
7. Нагрузочное тестирование.

Контрольные вопросы

1. Что такое профилирование и чем оно отличается от простого логирования?

2. Какие типы профилирования существуют и для каких задач они используются?
3. Что такое проблема N+1 запросов в Entity Framework Core и как ее обнаружить?
4. Как с помощью MiniProfiler проанализировать производительность ASP.NET Core приложения?
5. Какие метрики важны для оценки производительности REST API?
6. В чем разница между профилированием в development и production?
7. Какие инструменты для профилирования .NET приложений вы знаете?
8. Что такое async-over-sync и sync-over-async антипаттерны?
9. Как сборщик мусора (GC) влияет на производительность API?
10. Какие стратегии оптимизации запросов к БД существуют?

Список литературы

1. Рихтер, Д. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C# / Д. Рихтер. – 4-е изд. – Санкт-Петербург : Питер, 2017. – 896 с. — (Серия «Мастер-класс»).
2. Троелсен, Э. Язык программирования C# 7 и платформы .NET и .NET Core / Э. Троелсен, Ф. Джепикс. – 8-е изд. – Санкт-Петербург : Диалектика, 2018. – 1328 с.
3. Фримен, А. ASP.NET Core MVC 2 с примерами на C# для профессионалов / А. Фримен. – Москва : Вильямс, 2018. – 1024 с.
4. Локхарт, Дж. Современный C#: 100 советов по программированию / Дж. Локхарт. – Москва : ДМК Пресс, 2021. – 260 с.
5. Эванс, Б. Java: оптимизация программ. Практические методы повышения производительности приложений / Б. Эванс, Д. Грофф. – Москва : Диалектика, 2020. – 448 с.
6. Ньюмен, С. Создание микросервисов / С. Ньюмен. – Санкт-Петербург : Питер, 2016. – 704 с. – (Серия «Для профессионалов»).
7. Ричардсон, К. Микросервисы. Паттерны разработки и рефакторинга / К. Ричардсон. – Санкт-Петербург : Питер, 2019. – 544 с. – (Серия «Библиотека программиста»).
8. Фаулер, М. Шаблоны корпоративных приложений / М. Фаулер. – Москва : Вильямс, 2016. – 544 с.
9. Эванс, Э. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем / Э. Эванс. – Москва : Вильямс, 2011. – 448 с.
10. Скит, Д. C# для профессионалов. Тонкости программирования / Д. Скит. – 3-е изд. – Москва : Вильямс, 2019. – 608 с.
11. Бейли, Д. Docker для разработчиков / Д. Бейли. – Санкт-Петербург : Питер, 2021. – 352 с.
12. Мартин, Р. Чистая архитектура. Искусство разработки программного обеспечения / Р. Мартин. – Санкт-Петербург : Питер, 2018. – 352 с. – (Серия «Библиотека программиста»).

13. Фаулер, М. Рефакторинг. Улучшение существующего кода / М. Фаулер. – Москва : Вильямс, 2019. – 448 с.
14. Макконнелл, С. Совершенный код. Мастер-класс / С. Макконнелл. – Москва : Русская редакция, 2017. – 896 с.
15. Microsoft Docs. Документация по .NET- .NET / Microsoft Lean. – URL: https://docs.microsoft.com/ru-ru/dotnet/ (дата обращения: 01.06.2026).

Содержание