

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

РАЗРАБОТКА КОДА ИНФОРМАЦИОННЫХ СИСТЕМ

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

Разработка кода информационных систем : 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 (900 Кб). – Текст : электронный.

Методические материалы по дисциплине «Разработка кода информационных систем» для обучающихся специальности СПО 09.02.11 «Разработка и управление программным обеспечением» содержат указания для проведения лабораторных работ.

© Кузбасский государственный
технический университет
имени Т. Ф. Горбачева, 2026

© Ерошевич К. В., составление, 2026

Лабораторная работа №1

«Модуль обработки строк с регулярными выражениями»

Цель работы: Изучить принципы работы с текстовой информацией и освоить технологию использования регулярных выражений в среде разработки на языке C# для создания модулей валидации и поиска данных.

Теоретическая часть

Ключевые понятия:

1. **Регулярное выражение (Regular Expression):** Это формальный язык поиска и осуществления манипуляций с подстроками в тексте, основанный на использовании метасимволов. По сути, это шаблон, состоящий из символов и метасимволов, который задает правило поиска.
2. **Пространство имен System.Text.RegularExpressions:** В C# вся работа с регулярными выражениями сосредоточена в этом пространстве имен. Ключевым классом является `Regex`.
3. **Метасимволы:** Символы, имеющие специальное значение в шаблоне (например, `^`, `$`, `*`, `+`, `?`, `\d`, `\w`, `\s`).

Основные классы и их методы

Класс `Regex`

Представляет собой неизменяемое (immutable) скомпилированное регулярное выражение.

Компиляция шаблона: При создании экземпляра класса `Regex` в конструктор передается строковый шаблон. Возможно также указание опций (`RegexOptions`), таких как игнорирование регистра (`IgnoreCase`) или многострочный режим (`Multiline`).

Статические методы: Класс предоставляет статические методы, которые позволяют применить регулярное выражение к строке без явного создания экземпляра класса (например, `Regex.IsMatch(input, pattern)`). Это удобно для разовых операций, но при многократном использовании лучше создавать объект `Regex` для повышения производительности за счет кэширования.

Объект Match

Представляет собой результат одного совпадения, найденного во входной строке. Содержит свойства Value (сама найденная подстрока), Index (позиция начала совпадения), Length (длина совпадения), а также коллекцию групп Groups.

Группировка ()

Круглые скобки в шаблоне выполняют две функции:

1. Определяют порядок применения операторов (как в арифметических выражениях).
2. Создают захватываемую группу. Часть строки, которая совпала с подвыражением внутри скобок, доступна отдельно через свойство Groups объекта Match. Это позволяет извлекать из строки не всё совпадение целиком, а его составные части (например, из email можно отдельно извлечь имя пользователя и домен).

Задание

Общая постановка задачи:

Разработать консольное приложение (модуль), демонстрирующее работу с регулярными выражениями для решения трех типовых задач обработки текста.

Варианты заданий (выбрать один из вариантов или выполнить все три):

Вариант А (Валидатор): Создать модуль проверки корректности вводимых пользователем данных.

1. Разработать шаблон для проверки номера телефона в формате: +X (XXX) XXX-XXXX, где X – цифра.
2. Разработать шаблон для проверки адреса электронной почты (упрощенный вариант: латинские буквы, цифры, точки, подчеркивания, затем @, затем домен из латинских букв и точек).

Входные данные: набор тестовых строк (вводимых пользователем или загружаемых из файла).

Выходные данные: для каждой строки вывести вердикт: «Корректный телефон», «Некорректный телефон» или аналогично для email.

Вариант Б (Поиск и извлечение данных): Создать модуль анализа текстового лога или HTML-страницы.

Исходные данные: многострочный текст (например, содержимое небольшого HTML-файла или лог-файла сервера), содержащий даты в формате ДД.ММ.ГГГГ и числа (целые или десятичные).

Задача: Используя регулярные выражения, найти и извлечь в отдельные коллекции все даты и все числа. Вывести их на экран с указанием типа.

Вариант В (Форматирование и замена): Создать модуль для приведения текста к заданному стандарту.

Исходные данные: произвольный текст на русском или английском языке, в котором нарушены правила расстановки пробелов (лишние пробелы, отсутствие пробелов после знаков препинания).

Задача: Используя регулярные выражения и метод Replace, выполнить автоматическое форматирование текста:

1. Удалить лишние пробелы (оставить только один пробел между словами).
2. Удалить пробелы перед знаками препинания (., ,, !, ?, :, ;).
3. Гарантировать наличие одного пробела после каждого знака препинания.
4. Выходные данные: исходный и отформатированный текст.

Контрольные вопросы

1. Что такое регулярное выражение? Для решения каких задач оно применяется в информационных системах (приведите 2–3 примера)?
2. Какое пространство имен и какой основной класс в C# отвечают за работу с регулярными выражениями?

3. Объясните разницу между использованием статических методов класса `Regex` и созданием экземпляра класса `Regex`. В каком случае какой подход предпочтительнее?
4. Что такое метасимволы? Опишите назначение следующих метасимволов: `^`, `$`, `\d`, `\w`, `\s`, `*`, `+`, `?`.
5. Для чего используются квадратные скобки `[ ]` в шаблоне регулярного выражения? Приведите пример.
6. Что такое «захватываемая группа» и как она обозначается? Как получить доступ к значению группы в `C#` после успешного поиска?
7. Опишите, что будет делать следующий шаблон (без привязки к коду): `@"^\d{3}-\d{2}-\d{4}$"`.
8. С помощью какого метода класса `Regex` можно не только найти, но и заменить найденный фрагмент текста?
9. В чем разница между методами `Match` и `Matches`?
10. Какие сложности или ограничения могут возникнуть при использовании регулярных выражений для обработки сложных или вложенных структур (например, HTML или арифметических выражений)?

Лабораторная работа №2

«Обработчик исключений и структура данных»

Цель работы: Изучить механизм обработки исключительных ситуаций (exception handling) в C# и освоить принципы работы с обобщенными коллекциями (generics) на примере одной из структур данных для организации надежного хранения и обработки информации.

Теоретическая часть

Ключевые понятия:

Исключения (Exceptions)

Определение: Исключение – это событие, возникающее во время выполнения программы, которое нарушает нормальный ход выполнения инструкций. Это может быть деление на ноль, обращение к несуществующему элементу массива, ошибка ввода-вывода и т. д.

Механизм обработки: В C# реализован структурированный механизм обработки исключений, основанный на блоках try, catch и finally. Этот механизм позволяет отделить код обработки ошибок от основного кода программы.

Структуры данных и коллекции - обобщения (Generics): Механизм, позволяющий создавать классы, структуры, интерфейсы и методы, тип данных которых определяется при их создании. Это обеспечивает типобезопасность и производительность, избавляя от необходимости упаковки/распаковки (boxing/unboxing), как в необобщенных коллекциях (ArrayList, Hashtable).

Интеграция исключений и структур данных: при работе с коллекциями необходимо предусматривать обработку исключительных ситуаций.

Задание

Общая постановка задачи:

Разработать консольное приложение, которое реализует работу с одной из стандартных структур данных (коллекций), инкапсу-

лируя логику работы в отдельном классе. Программа должна демонстрировать корректную обработку исключительных ситуаций, которые могут возникнуть при взаимодействии с этой структурой (попытка доступа к несуществующему элементу, добавление дубликата, работа с пустой коллекцией и т. д.). Все действия пользователя и возникающие ошибки должны логироваться (записываться в файл).

Варианты заданий (выберите один):

Вариант А (Библиотечный каталог на Dictionary):

1. Реализуйте класс LibraryCatalog, который внутри использует Dictionary<string, string> (ключ – уникальный инвентарный номер книги, значение – название книги).
2. Реализуйте методы:
 - AddBook(string id, string title) – добавление книги. Должен генерировать исключение, если книга с таким ID уже существует.
 - RemoveBook(string id) – удаление книги. Должен генерировать исключение, если книга с таким ID не найдена.
 - FindBook(string id) – поиск книги. Должен генерировать исключение, если ID не найден.
 - PrintAllBooks() – вывод всего каталога. Должен корректно обрабатывать ситуацию пустого каталога (не генерировать ошибку, а выводить сообщение).
3. В основной программе создайте меню для вызова этих методов и обработайте все возможные исключения с выводом понятных пользователю сообщений.

Вариант Б (Журнал событий на List):

1. Реализуйте класс EventLog, который внутри использует List<string> для хранения записей событий.
2. Реализуйте методы:
 - AddEvent(string message) – добавление события.
 - GetEvent(int index) – получение события по индексу. Должен генерировать исключение при выходе индекса за границы.

- `GetLastEvent()` – получение последнего события. Должен генерировать исключение, если список пуст.
- `ClearLog()` – очистка журнала.
- `GetStats()` – возвращает общее количество событий.

3. Реализуйте статический метод `WriteExceptionToFile` (`Exception ex`), который при возникновении любой ошибки записывает в текстовый файл `log.txt` время ошибки, тип исключения и сообщение.

Вариант В (Очередь задач на Queue):

1. Реализуйте класс `TaskQueue`, который внутри использует `Queue<string>` для хранения задач.
2. Реализуйте методы:
 - `EnqueueTask(string task)` – постановка задачи в очередь.
 - `DequeueTask()` – извлечение задачи из очереди. Должен генерировать исключение `InvalidOperationException` с понятным текстом, если очередь пуста.
 - `PeekNextTask()` – просмотр следующей задачи без извлечения. Аналогичная обработка пустой очереди.
 - `GetQueueLength()` – получение длины очереди.
3. В основной программе реализуйте «безопасное» извлечение задач, перехватывая исключение `InvalidOperationException` и предлагая пользователю добавить задачу перед извлечением.

Требования к реализации модуля:

1. Работа с коллекцией должна быть инкапсулирована в отдельном классе (не в `Main`).
2. Методы класса должны генерировать исключения (`throw`) при возникновении ошибочных ситуаций, специфичных для логики работы (например, "Ключ уже существует").
3. В методе `Main` или в точке взаимодействия с пользователем должны использоваться блоки `try-catch` для перехвата этих исключений и вывода информативных сообщений.
4. Для одного из блоков `catch` должен быть реализован `finally`, который, например, уведомляет о завершении операции или закрывает ресурс (если используется файловый ввод/вывод).

5. В программе должны обрабатываться как пользовательские исключения (брошенные вами), так и системные (например, `FormatException` при вводе числа).

Контрольные вопросы

1. Что такое исключение? Чем исключительная ситуация отличается от обычной ошибки в логике программы?
2. Опишите назначение и порядок выполнения блоков `try`, `catch`, `finally`.
3. Может ли в конструкции `try-catch` отсутствовать блок `finally`? Может ли отсутствовать блок `catch`? Если да, то в каких случаях?
4. Что произойдет, если в блоке `try` возникнет исключение, тип которого не указан ни в одном из последующих блоков `catch`?
5. Для чего используется ключевое слово `throw`?
6. Что такое обобщения (generics) в C#? Какое преимущество дает использование `List<T>` перед необобщенным `ArrayList`?
7. Опишите разницу между коллекциями `List<T>` и `Dictionary<TKey, TValue>`. В каких случаях предпочтительнее использовать словарь?
8. Чем отличается стек (`Stack<T>`) от очереди (`Queue<T>`)? Приведите примеры использования каждой структуры в реальных информационных системах.
9. Какое исключение возникнет при попытке обращения к элементу `List<T>` по индексу, который больше количества элементов? А при попытке вызвать `Pop()` на пустом стеке?
10. Как бы вы реализовали собственный класс исключения (например, `BookNotFoundException`)? От какого класса он должен наследоваться и зачем это может понадобиться?

Лабораторная работа №3

«Файловый менеджер с сериализацией»

Цель работы: Изучить возможности платформы .NET по работе с файловой системой (пространство имен System.IO) и освоить механизмы сериализации/десериализации объектов для обеспечения долговременного хранения данных информационной системы.

Теоретическая часть

Ключевые понятия:

Работа с файловой системой (System.IO):

1. Класс File: Предоставляет статические методы для создания, копирования, удаления, перемещения и открытия файлов. Удобен для однократных операций с файлами.
2. Класс FileInfo: Предоставляет методы экземпляра для тех же операций. Требуется создания объекта и больше подходит для многократных операций с одним и тем же файлом, так как не требует проверки прав безопасности при каждом вызове.
3. Класс Directory: Статические методы для работы с каталогами (создание, удаление, перемещение, получение списка подкаталогов и файлов).
4. Класс DirectoryInfo: Методы экземпляра для работы с каталогами.
5. Класс Path: Статические методы для работы с путями к файлам и каталогам (объединение путей, получение расширения, имени файла и т. д.). Позволяет создавать пути, корректные для разных операционных систем.
6. Класс StreamReader/StreamWriter: Позволяют читать и записывать текстовую информацию в поток (файл). Работают на более высоком уровне абстракции.
7. Класс FileStream: Предоставляет низкоуровневый доступ к файлу (побайтовое чтение/запись). Используется как основа для других потоков.

Сериализация (Serialization):

1. Определение: Процесс преобразования объекта (или графа связанных объектов) в последовательность байт для передачи по сети или сохранения на диске.
2. Десериализация (Deserialization): Обратный процесс – восстановление объекта из последовательности байт.
3. Назначение: Сохранение состояния программы между запусками (persistence), глубокое копирование объектов, передача данных между процессами/сервисами.

Форматы и механизмы сериализации в C#:

1. Атрибут [Serializable]: Помечает класс, объекты которого могут быть сериализованы. Это требование для многих старых механизмов сериализации (BinaryFormatter – не рекомендуется использовать в новых проектах из-за проблем безопасности).
2. XML-сериализация (System.Xml.Serialization).
3. JSON-сериализация (System.Text.Json).
4. Атрибут [NonSerialized] (для BinaryFormatter) / [JsonIgnore] (для System.Text.Json): Позволяет исключить определенные поля из процесса сериализации.

Задание**Общая постановка задачи:**

Разработать консольное приложение "Файловый менеджер", которое позволяет пользователю выполнять базовые операции с файлами и каталогами, а также предоставляет функционал для сохранения и загрузки структурированных данных (например, списка файлов с их атрибутами или пользовательских заметок) в файл с использованием сериализации.

Варианты заданий (выберите один):**Вариант А** (Файловый менеджер с историей операций):

1. Разработайте класс FileOperation с полями: OperationType (Copy, Delete, Create), FilePath, Timestamp.
2. Реализуйте базовый файловый менеджер с функциями: просмотр содержимого текущего каталога (с использованием

DirectoryInfo), создание текстового файла, копирование файла, удаление файла.

3. Все успешные операции пользователя должны записываться в список `List<FileOperation>`.
4. При запуске программы должна выполняться десериализация (из JSON-файла `history.json`) предыдущей истории операций, если она существует.
5. При выходе из программы (по команде пользователя) должна выполняться сериализация текущей истории в JSON-файл.
6. Добавьте команду для просмотра всей истории операций.

Вариант Б (Менеджер заметок с тегами):

1. Разработайте класс `Note` с полями: `Title` (заголовок), `Content` (текст заметки), `CreationDate`, `Tags` (список строк, `List<string>`).
2. Реализуйте консольный интерфейс для: создания новой заметки (ввод заголовка, текста, тегов через запятую); просмотра списка всех заметок (только заголовки и даты); просмотра полного содержимого конкретной заметки по заголовку или индексу; удаления заметки; поиска заметок по тегу.
3. Все заметки должны храниться в коллекции `List<Note>` в памяти.
4. При запуске программа должна загружать заметки из XML-файла `notes.xml` (используя `XmlSerializer`).
5. При завершении программы (или после каждой операции, по желанию) – сохранять актуальный список в тот же XML-файл.

Вариант В (Информатор о дисках и каталогах с сохранением состояния):

1. Реализуйте класс `DirectorySnapshot`, который хранит информацию о состоянии каталога: полный путь и список объектов `FileInfo` (или их упрощенных DTO-моделей с именем, размером, датой изменения).
2. Создайте утилиту, которая: отображает список всех логических дисков на машине, позволяет пользователю перейти в указанный каталог, показывает содержимое текущего каталога (файлы и подпапки) с их атрибутами

(размер для файлов, дата изменения), позволяет пользователю сделать "снимок" (snapshot) текущего каталога (создать объект `DirectorySnapshot` и добавить его в общий список `List<DirectorySnapshot>`).

3. Реализуйте возможность просмотра всех сохраненных снимков.
4. Сохраняйте список всех снимков в JSON-файл при выходе и загружайте при старте программы.

Требования к реализации модуля:

1. Активно использовать классы из пространства имен `System.IO` (`File`, `Directory`, `Path`, `FileInfo`, `DirectoryInfo`).
2. Использовать механизм сериализации (JSON или XML) для сохранения и загрузки состояния программы.
3. Для JSON-сериализации использовать `System.Text.Json`. Для XML – `XmlSerializer`.
4. При работе с файловой системой предусмотреть обработку исключений (`DirectoryNotFoundException`, `FileNotFoundException`, `UnauthorizedAccessException`, `IOException`).
5. Организовать структуру программы с четким разделением на логику работы с данными и пользовательский интерфейс.
6. Предусмотреть проверку существования файла с данными при загрузке, чтобы не возникало исключения при первом запуске программы.

Контрольные вопросы

1. В чем разница между классами `File` и `FileInfo`? В каком случае вы бы использовали каждый из них?
2. Как получить список всех текстовых файлов (с расширением `.txt`) в определенном каталоге с учетом подкаталогов?
3. Что такое сериализация и десериализация? Зачем они нужны в информационных системах?
4. Какие форматы сериализации поддерживаются в C#? Перечислите их достоинства и недостатки.

5. Для чего нужен атрибут [Serializable]? Обязателен ли он для XmlSerializer или JsonSerializer?
6. Как исключить определенное поле класса из сериализации при использовании System.Text.Json?
7. В чем разница между StreamReader/StreamWriter и FileStream? Что такое поток (stream)?
8. Какие исключения могут возникнуть при попытке удалить файл, который используется другим процессом? Как это можно обработать?
9. Что произойдет при попытке десериализации из пустого или несуществующего файла?
10. Как бы вы реализовали резервное копирование файла перед его перезаписью при сохранении данных?

Лабораторная работа №4

«Многомодульное консольное приложение»

Цель работы: Изучить принципы модульного проектирования программного обеспечения, освоить подходы к разделению приложения на логические слои (уровни абстракции) и научиться организовывать взаимодействие между модулями через четко определенные интерфейсы.

Теоретическая часть

Ключевые понятия:

Модульность (Modularity):

Определение: свойство системы, при котором она разделена на внутренне связанные, но слабо зависящие друг от друга части – модули. Каждый модуль отвечает за строго определенную функциональность.

Преимущества:

1. Удобство разработки: Над разными модулями могут работать разные разработчики.
2. Тестирование: Модули можно тестировать изолированно (юнит-тестирование).
3. Повторное использование: Модуль можно использовать в разных проектах.
4. Поддерживаемость: Изменения в одном модуле с меньшей вероятностью ломают другие.

Архитектурные слои (Layers):

Одним из распространенных способов модульности является разделение на горизонтальные слои. Для информационной системы типичны следующие слои:

1. Слой представления (Presentation Layer): Отвечает за взаимодействие с пользователем. В консольном приложении это может быть класс, обрабатывающий ввод с клавиатуры и вывод на экран (меню, приглашения, форматированный вывод). Его задача – получить команду от пользователя и

отобразить результат, не вникая в детали вычислений или работы с данными.

2. Слой бизнес-логики (Business Logic Layer/Domain Layer): Содержит правила предметной области (core logic). Именно здесь выполняются основные вычисления, проверки (валидация), реализуются алгоритмы и сценарии использования (use cases). Этот слой не должен знать о том, откуда пришли данные (от пользователя, из файла, из сети) и куда они будут сохранены.
3. Слой доступа к данным (Data Access Layer/Persistence Layer): Отвечает за хранение и извлечение данных. Он предоставляет бизнес-логике простой интерфейс для работы с данными, скрывая детали реализации: будь то файловая система, база данных или внешнее API. В контексте предыдущих работ, сюда относится код сериализации и работы с System.IO.

Интерфейсы и принцип инверсии зависимостей (Dependency Inversion Principle):

1. Чтобы слои оставались независимыми, они не должны напрямую ссылаться друг на друга. Вместо этого слои должны зависеть от абстракций (интерфейсов).
2. Интерфейс (Interface) в C# – это контракт, который определяет набор методов и свойств, но не реализует их. Класс, реализующий интерфейс, обязуется предоставить реализацию всех его членов.

Разделение на проекты (Assemblies):

В рамках одной среды разработки (решения Visual Studio) логические слои часто физически разделяют на отдельные проекты (библиотеки классов). Это дает следующие преимущества:

1. Явное указание зависимостей между модулями (на уровне ссылок на проекты).
2. Возможность использовать модули в других решениях.
3. Четкое разграничение областей ответственности.

Задание

Общая постановка задачи:

Реорганизовать код одного из предыдущих заданий (лабораторные работы №2 или №3) или разработать новое простое приложение, применив принципы модульной архитектуры. Необходимо разделить приложение на три логических слоя, выделить их в отдельные классы (или проекты) и организовать взаимодействие через интерфейсы.

Варианты заданий (выберите один):

Вариант А (Рефакторинг "Менеджера заметок"):

Возьмите за основу "Менеджер заметок с тегами" из лабораторной работы №3 (вариант Б).

Создайте структуру решения (Solution), состоящую из трех проектов:

1. Notes.Models (Библиотека классов): Содержит только модель данных – класс Note. Ни от чего не зависит.
2. Notes.Data (Библиотека классов): Отвечает за хранение. Содержит интерфейс INoteRepository с методами IEnumerable<Note> LoadAll(), void SaveAll(IEnumerable<Note> notes). Также содержит конкретную реализацию JsonNoteRepository, который работает с файлами (использует код сериализации из ЛР3).
3. Notes.App (Консольное приложение): Содержит слой представления и бизнес-логику. Получает ссылки на проекты Notes.Models и Notes.Data. Взаимодействует с хранилищем только через интерфейс INoteRepository.

Продемонстрируйте, что, заменив в точке входа (Main) JsonNoteRepository на гипотетическую другую реализацию (например, MockNoteRepository, которая хранит данные в памяти), приложение продолжит работать без изменения кода бизнес-логики.

Вариант Б (Калькулятор с историей и архитектурой):

Разработайте простое приложение-калькулятор (сложение, вычитание, умножение, деление) с сохранением истории вычислений.

Примените модульную архитектуру:

1. Calculator.Core (Библиотека классов): Содержит интерфейс ICalculator с методами операций. Содержит класс SimpleCalculator, реализующий эти методы (бизнес-логика). Также содержит модель CalculationRecord (Операнд1, Операнд2, Операция, Результат, Дата).
2. Calculator.Storage (Библиотека классов): Содержит интерфейс IHistoryStorage с методами AddRecord, IEnumerable<CalculationRecord> GetAllRecords. Содержит реализацию FileHistoryStorage, который сохраняет историю в JSON/XML файл (доступ к данным).
3. Calculator.UI (Консольное приложение): Содержит меню, ввод/вывод. Получает ссылки на два предыдущих проекта. Создает объекты SimpleCalculator и FileHistoryStorage, связывает их и запускает цикл обработки команд пользователя.

При каждой операции калькулятор должен через интерфейс IHistoryStorage сохранять запись о вычислении. Пользователь должен иметь возможность просмотреть всю историю.

Вариант В (Система управления задачами с несколькими хранилищами):

Реализуйте простой TODO-менеджер (список задач: описание, статус выполнения, дата создания).

Создайте модули:

1. Todo.Models: Класс TaskItem.
2. Todo.Data: Интерфейс ITaskRepository (добавить, удалить, обновить статус, получить все). Реализуйте две версии репозитория:
3. MemoryTaskRepository – хранит задачи в List<T> в оперативной памяти (теряет данные после выхода).
4. FileTaskRepository – хранит задачи в JSON-файле.
5. Todo.App: Слой представления (консольное меню). При запуске программа должна спрашивать пользователя, какой тип хранилища использовать (в памяти или в файле). В зависимости от ответа, создается соответствующий

репозиторий (через интерфейс), и вся дальнейшая работа идет через него.

Требования к реализации модуля:

1. Обязательное использование интерфейсов для определения контрактов между слоями (особенно между бизнес-логикой/представлением и слоем данных).
2. Четкое разделение ответственности: классы, работающие с пользователем, не должны содержать кода работы с файлами, и наоборот.
3. Решение должно быть организовано либо как несколько проектов в одном решении, либо как несколько папок/пространств имен с четкой структурой.
4. В основной программе (точке входа) должна выполняться композиция приложения – создание экземпляров конкретных классов и связывание их между собой (это место называют Composition Root).
5. Приложение должно сохранять работоспособность и обрабатывать ошибки ввода-вывода и пользовательского ввода (как в ЛР2 и ЛР3).

Контрольные вопросы

1. Что такое модульность? Почему она важна при разработке больших информационных систем?
2. Опишите классическую трехуровневую архитектуру (Presentation, Business Logic, Data Access). Какую функцию выполняет каждый слой?
3. Почему слой бизнес-логики не должен напрямую зависеть от слоя доступа к данным?
4. Какую роль в организации слабой связанности (loose coupling) играют интерфейсы? Приведите пример из вашей работы.
5. Что такое принцип инверсии зависимостей (DIP – Dependency Inversion Principle)? Как он был применен в лабораторной работе?

6. В чем разница между агрегацией и композицией при связывании модулей? (в контексте того, кто создает объекты зависимостей).
7. Где в вашем приложении находится Composition Root (корень композиции)? Почему важно выносить создание зависимостей именно туда?
8. Какие преимущества дает физическое разделение на отдельные проекты (библиотеки классов) по сравнению с простым разделением на папки в рамках одного проекта?
9. Как бы изменилась архитектура приложения, если бы потребовалось добавить новый способ вывода данных (например, не на консоль, а в графический интерфейс или Telegram-бота)?
10. Что такое юнит-тестирование и как модульная архитектура облегчает написание юнит-тестов? Какой слой вы бы тестировали в первую очередь и почему?

Лабораторная работа №5

«TCP-сервер для обработки JSON запросов»

Цель работы: Изучить основы сетевого программирования в среде .NET, освоить работу с сокетами (TCP) для создания серверных приложений, а также научиться организовывать взаимодействие клиента и сервера на основе обмена сообщениями в формате JSON.

Теоретическая часть

Ключевые понятия:

Сетевое программирование и сокеты:

1. Сокет (Socket): Программный интерфейс, обеспечивающий обмен данными между процессами. В контексте TCP/IP сокет идентифицируется IP-адресом и номером порта.
2. TCP (Transmission Control Protocol): Протокол транспортного уровня, обеспечивающий надежное, упорядоченное соединение между клиентом и сервером. Гарантирует доставку данных, контролирует целостность потока и автоматически пересылает потерянные пакеты.
3. Клиент-серверная модель

Классы .NET для сетевого взаимодействия (System.Net.Sockets):

1. TcpListener: Высокоуровневая обертка над сокетом для серверной стороны. Позволяет запустить прослушивание порта (Start()), принять входящее соединение (AcceptTcpClient() – блокирующий вызов) и получить объект TcpClient для обмена данными с подключившимся клиентом.
2. TcpClient: Предоставляет клиентское подключение для обмена данными по TCP. Используется как на стороне клиента (для подключения к серверу), так и на стороне сервера (для работы с каждым подключившимся клиентом). Предоставляет метод GetStream() для получения сетевого потока.
3. NetworkStream: Поток данных, через который осуществляется чтение и запись данных при TCP-соединении.

Асинхронность:

1. В серверных приложениях критически важно не блокировать выполнение программы на время ожидания подключения или чтения данных.
2. Асинхронные методы (`async/await`): Позволяют запускать длительные операции (ввод-вывод) без блокировки основного потока выполнения.
3. Ключевые асинхронные методы: `AcceptTcpClientAsync()`, `ReadAsync()`, `WriteAsync()`. Они позволяют серверу ожидать подключения или данные от клиента, одновременно (в одном потоке) обрабатывая другие задачи или принимая новых клиентов.

Протокол на основе JSON:

Поскольку TCP является потоковым протоколом, он не имеет встроенного понятия "сообщения" или "пакета". Данные отправляются и читаются как поток байтов. Чтобы понимать, где заканчивается один запрос и начинается другой, необходимо определить протокол прикладного уровня.

Способы разделения сообщений:

1. Длина сообщения: Сначала отправляется целое число (4 байта), указывающее длину последующего JSON-сообщения. Сервер читает эти 4 байта, узнает длину, а затем считывает ровно столько байт, сколько указано.
2. Символ-разделитель: Например, символ перевода строки `\n`. Сервер читает данные до тех пор, пока не встретит этот символ.

Структура JSON-запроса: Для унификации взаимодействия клиент и сервер договариваются о формате JSON.

Задание**Общая постановка задачи:**

Разработать два приложения: TCP-сервер и TCP-клиент. Сервер должен прослушивать заданный порт, принимать подключения от клиентов, получать от них JSON-запросы, парсить их, выполнять соответствующую команду (бизнес-логику) и отправлять

обратно JSON-ответ. Клиент должен подключаться к серверу, отправлять команды в формате JSON и выводить полученные ответы.

Варианты заданий (выберите один):

Вариант А (Математический сервер):

Сервер: Реализует обработку математических команд.

Поддерживаемые команды в JSON:

1. {"command": "ADD", "a": 10, "b": 5} → возвращает сумму.
2. {"command": "SUB", "a": 10, "b": 5} → возвращает разность.
3. {"command": "MUL", "a": 10, "b": 5} → возвращает произведение.
4. {"command": "DIV", "a": 10, "b": 5} → возвращает частное (обработка деления на ноль с возвратом ошибки в JSON).

Клиент: Консольное приложение с меню. Позволяет пользователю выбрать операцию, ввести два числа, сформировать JSON-запрос, отправить его на сервер, получить и отобразить ответ (результат или сообщение об ошибке).

Требование: Сервер должен быть асинхронным и поддерживать одновременное обслуживание нескольких клиентов (например, через Task.Run).

Вариант Б (Сервер заметок – многопользовательский):

Сервер: Хранит заметки в памяти (или с использованием репозитория из ЛР3/ЛР4). Понимает команды:

1. {"command": "CREATE", "username": "alice", "title": "Заголовок", "content": "Текст"} – создает заметку для пользователя.
2. {"command": "GET_ALL", "username": "alice"} – возвращает список всех заметок пользователя.
3. {"command": "DELETE", "username": "alice", "title": "Заголовок"} – удаляет заметку.

Клиент: При запуске запрашивает имя пользователя. Затем показывает меню (создать, показать все, удалить, выйти). Формирует JSON-запросы и отправляет на сервер.

Требование: JSON-ответы должны содержать поле status ("success" или "error") и либо data (список заметок), либо message (подтверждение/ошибка).

Вариант В (Эмулятор банкомата):

Сервер: Хранит информацию о нескольких счетах (номер счета, баланс, PIN-код – для простоты можно в памяти). Реализует команды:

1. {"command": "AUTH", "account": "12345", "pin": "1234"} – проверка PIN. Возвращает токен сессии (например, GUID).
2. {"command": "BALANCE", "session": "guid-токен"} – запрос баланса (требуется валидная сессия).
3. {"command": "WITHDRAW", "session": "...", "amount": 100} – снятие средств (проверка наличия средств, возврат ошибки при недостатке).
4. {"command": "DEPOSIT", "session": "...", "amount": 100} – пополнение счета.

Клиент: Позволяет пользователю ввести номер счета и PIN для аутентификации. После успеха показывает меню операций. Сессионный токен хранится в клиенте и добавляется в каждый последующий запрос.

Требования к реализации модуля:

1. Сервер должен использовать TcpListener и работать асинхронно.
2. Для разделения сообщений в TCP-потоке использовать префикс с длиной сообщения (первыми 4 байтами передавать длину последующего JSON).
3. Для парсинга JSON использовать System.Text.Json.
4. Предусмотреть корректное завершение работы сервера (остановка прослушивания, закрытие подключений).
5. Обрабатывать возможные исключения: обрыв соединения, некорректный JSON, неизвестная команда.
6. Клиент должен иметь понятный интерфейс и корректно отображать как успешные результаты, так и ошибки, пришедшие от сервера.

Контрольные вопросы

1. В чем разница между протоколами TCP и UDP? Почему для данного задания был выбран TCP?
2. Опишите назначение классов TcpListener и TcpClient. В чем их отличие от низкоуровневого класса Socket?
3. Что такое блокирующий и неблокирующий (асинхронный) ввод-вывод? Почему асинхронный подход критически важен для серверных приложений?
4. Какую проблему решает предварение каждого JSON-сообщения его длиной (length-prefixing)?
5. Каким образом сервер может обрабатывать несколько клиентов одновременно? Опишите возможные подходы (многопоточность, асинхронность).
6. Какие шаги необходимо выполнить клиенту, чтобы установить соединение с сервером и начать обмен данными?
7. Какие исключения могут возникнуть при работе с сетью? Как их корректно обработать, чтобы сервер продолжил работу?
8. Для чего нужна сериализация и десериализация JSON в контексте данной работы? Какие классы из System.Text.Json для этого используются?
9. В чем преимущество текстовых протоколов (например, на основе JSON) перед бинарными? В чем их недостатки?
10. Предложите схему аутентификации для защиты сервера от неавторизованных запросов (например, как в варианте В).

Лабораторная работа №6

«REST API для управления задачами»

Цель работы: Изучить принципы построения RESTful-сервисов, освоить фреймворк ASP.NET Core для создания HTTP-серверов и научиться реализовывать полноценное API с поддержкой основных операций CRUD (Create, Read, Update, Delete).

Теоретическая часть

Ключевые понятия

REST (Representational State Transfer):

Определение: Архитектурный стиль взаимодействия компонентов распределенного приложения. В контексте веба – это набор принципов организации взаимодействия клиента и сервера через HTTP.

Ключевые принципы REST:

1. Клиент-серверная архитектура: Разделение ответственности.
2. Отсутствие состояния (Stateless): Сервер не хранит информацию о состоянии клиента между запросами. Каждый запрос содержит всю необходимую информацию для его обработки.
3. Кэширование: Ответы должны явно указывать, можно ли их кэшировать.
4. Единообразие интерфейса (Uniform Interface):
Основополагающий принцип. Ресурсы идентифицируются в запросах (обычно через URL), а их представление отделено от внутренней реализации. Манипуляции над ресурсами выполняются через стандартные HTTP-методы.

Ресурсы и HTTP-методы

В REST все является ресурсами (задачи, пользователи, заказы). Каждому ресурсу соответствует уникальный URI (например, /api/tasks, /api/tasks/123).

Для работы с ресурсами используются стандартные HTTP-методы (глаголы):

1. GET – получение ресурса или списка ресурсов (идемпотентный, безопасный).
2. POST – создание нового ресурса (не идемпотентный).
3. PUT – полное обновление существующего ресурса (идемпотентный).
4. PATCH – частичное обновление ресурса.
5. DELETE – удаление ресурса (идемпотентный).

ASP.NET Core Web API

Контроллеры (Controllers): Классы, обрабатывающие входящие HTTP-запросы. Наследуются от ControllerBase. Методы контроллера (actions) сопоставляются с конкретными HTTP-методами и маршрутами.

Атрибуты маршрутизации:

1. [Route("api/[controller]")] – задает базовый маршрут для контроллера.
2. [HttpGet], [HttpPost("{id}"), [HttpPut("{id}"), [HttpDelete("{id}")] – привязывают метод к HTTP-методу и шаблону URL.

Модели (Models) и DTO (Data Transfer Objects):

1. Модели: Представляют сущности предметной области.
2. DTO: Объекты передачи данных, используемые для обмена между клиентом и сервером. Позволяют скрыть внутреннюю структуру модели и отправлять только необходимые поля.

Автоматическое связывание (Model Binding): ASP.NET Core автоматически извлекает данные из тела запроса (JSON), строки запроса, маршрута или заголовков и преобразует их в параметры метода контроллера (с помощью атрибутов [FromBody], [FromRoute], [FromQuery]).

Коды состояния HTTP

Важной частью REST API являются коды состояния, которые сервер возвращает вместе с ответом:

1. 200 OK – успешный запрос.
2. 201 Created – ресурс успешно создан (обычно в ответ на POST).

3. 204 No Content – успешный запрос без тела ответа (часто для DELETE, PUT).
4. 400 Bad Request – неверный запрос (ошибка валидации).
5. 404 Not Found – ресурс не найден.
6. 500 Internal Server Error – внутренняя ошибка сервера.

Задание

Общая постановка задачи:

Разработать Web API на базе ASP.NET Core, реализующее полноценное RESTful-сервис для управления задачами (Task Manager). API должно поддерживать все основные CRUD-операции и возвращать корректные HTTP-статусы.

Спецификация API.

Сущность TaskItem (задача):

1. Id (int, уникальный идентификатор).
2. Title (string, обязательное, максимум 100 символов).
3. Description (string, опционально).
4. IsCompleted (bool, статус выполнения).
5. CreatedAt (DateTime, дата создания, устанавливается автоматически).

Требуемые эндпоинты:

1. GET /api/tasks – получить список всех задач.
2. GET /api/tasks/{id} – получить задачу по ID. Возвращать 404, если задача не найдена.
3. POST /api/tasks – создать новую задачу. Принимать DTO без Id и CreatedAt (они генерируются на сервере). Возвращать 201 Created с созданным объектом и Location-заголовком.
4. PUT /api/tasks/{id} – полностью обновить задачу. Принимать обновленный объект. Возвращать 204 No Content при успехе, 400 при ошибках валидации, 404 если задача не найдена.
5. DELETE /api/tasks/{id} – удалить задачу. Возвращать 204 No Content при успехе, 404 если задача не найдена.
6. PATCH /api/tasks/{id}/complete – отметить задачу как выполненную (специальный эндпоинт для изменения статуса).

Варианты расширения (дополнительные задания):

1. Вариант А (С фильтрацией): Добавить возможность фильтрации списка задач по статусу (?isCompleted=true) и поиска по заголовку (?search=куп).
2. Вариант Б (С валидацией): Реализовать полноценную валидацию входных данных с использованием атрибутов ([Required], [MaxLength]) и возвращать детализированные ошибки валидации в теле ответа (формат Problem Details).
3. Вариант В (С хранением в БД): Вместо хранения в памяти подключить реальную базу данных (например, SQLite) с использованием Entity Framework Core для сохранения задач между перезапусками.

Требования к реализации:

1. Создать новый проект ASP.NET Core Web API (пустой шаблон или шаблон с контроллерами).
2. Реализовать контроллер TasksController.
3. Использовать отдельные классы для моделей и DTO (например, TaskCreateDto, TaskUpdateDto, TaskResponseDto).
4. Данные хранить в памяти (статический список List<TaskItem>), если не выбран вариант В.
5. Обрабатывать ситуации, когда запрашиваемый ресурс не найден (return NotFound()).
6. Для POST эндпоинта возвращать корректный 201 Created с заголовком Location, указывающим на URL созданного ресурса.
7. Протестировать API с помощью инструментов типа Swagger (встроенный), Postman или curl.

Контрольные вопросы

1. Что такое REST? Перечислите основные принципы RESTful-архитектуры.
2. Чем REST отличается от просто HTTP? Что такое "ресурс" в контексте REST?
3. Для чего используются различные HTTP-методы (GET, POST, PUT, DELETE, PATCH)? В чем разница между PUT и PATCH?

4. Что такое идемпотентность? Какие HTTP-методы являются идемпотентными и почему?
5. Что такое DTO (Data Transfer Object) и зачем его использовать в API, если у нас уже есть модель данных?
6. Какие коды состояния HTTP нужно возвращать при успешном создании ресурса, при успешном удалении, при попытке получить несуществующий ресурс?
7. Как в ASP.NET Core происходит связывание данных из JSON-запроса с параметрами метода контроллера?
8. Как работает маршрутизация в ASP.NET Core? Что означают атрибуты [Route], [HttpGet("{id}")]?
9. Какие существуют способы передачи параметров в Web API (через URL, строку запроса, тело запроса, заголовки)?
10. Как можно документировать Web API? Что такое Swagger/OpenAPI и для чего он используется?

Лабораторная работа №7

«HTTP-клиент с обработкой ошибок»

Цель работы: Изучить основы клиент-серверного взаимодействия по протоколу HTTP, освоить класс HttpClient для отправки запросов и обработки ответов, а также научиться корректно обрабатывать различные типы ошибок при работе с удаленными сервисами.

Теоретическая часть

Ключевые понятия:

HTTP-клиент в .NET:

Класс HttpClient: Основной класс для отправки HTTP-запросов и получения HTTP-ответов от ресурса, идентифицируемого URI.

Важные особенности использования:

1. Единый экземпляр: HttpClient предназначен для создания один раз и использования на протяжении всей жизни приложения. При частом создании и удалении HttpClient происходит исчерпание сокетов (порт-эксаустен). Для этого рекомендуется использовать IHttpClientFactory.
2. Асинхронность: Все методы HttpClient (GetAsync, PostAsync, SendAsync) являются асинхронными и должны использоваться с await.

Основные методы:

1. GetAsync – отправка GET-запроса.
2. PostAsync – отправка POST-запроса (обычно с телом).
3. PutAsync – отправка PUT-запроса.
4. DeleteAsync – отправка DELETE-запроса.
5. SendAsync – наиболее гибкий метод, позволяющий настроить все аспекты запроса через HttpRequestMessage.

Работа с HTTP-сообщениями:

1. HttpRequestMessage: Представляет собой HTTP-запрос. Позволяет задать метод, URI, заголовки, тело (содержимое).

2. `HttpResponseMessage`: Представляет собой HTTP-ответ. Содержит код состояния (`StatusCode`), заголовки, тело (`Content`) и флаг успешности (`IsSuccessStatusCode` – true для кодов 2xx).
3. `StringContent/ByteArrayContent/StreamContent`: Классы для представления тела запроса/ответа в различных форматах. Для JSON чаще всего используется `StringContent` с указанием `MediaTypeHeaderValue` "application/json".

Сериализация/Десериализация JSON:

1. Для отправки данных на сервер объект C# необходимо сериализовать в JSON-строку (с помощью `JsonSerializer.Serialize`).
2. Для обработки ответа от сервера JSON-строку необходимо десериализовать обратно в объект C# (с помощью `JsonSerializer.Deserialize<T>`).

Обработка ошибок:

1. Сетевые ошибки: Исключения типа `HttpRequestException` (нет соединения, DNS-ошибка, таймаут). Их нужно перехватывать и обрабатывать отдельно.
2. HTTP-статусы ошибок (4xx, 5xx): `HttpClient` не генерирует исключение при получении таких статусов (кроме случаев, когда вызван метод `EnsureSuccessStatusCode()`). Разработчик должен сам проверять свойство `IsSuccessStatusCode` или `StatusCode` и реагировать соответствующим образом.
3. Логирование: Важно логировать как успешные запросы, так и ошибки для отладки и мониторинга.

Задание

Общая постановка задачи:

Разработать консольное приложение (или приложение с простым UI), которое выступает в роли HTTP-клиента для взаимодействия с REST API, созданным в лабораторной работе №6 (или с любым тестовым публичным API). Клиент должен предоставлять пользовательский интерфейс для выполнения операций с задачами и демонстрировать корректную обработку всех возможных ошибок.

Базовый сценарий (для API из ЛР№6):

Создайте клиент, который подключается к вашему API (например, <http://localhost:5000/api/tasks>) и позволяет:

1. Просматривать список всех задач.
2. Просматривать детали конкретной задачи по ID.
3. Создавать новую задачу (ввод заголовка и описания).
4. Отмечать задачу как выполненную.
5. Удалять задачу.

Требования к реализации клиента:

1. Использовать HttpClient (правильно, один экземпляр на всё время работы программы или через IHttpClientFactory).
2. Для всех операций с API использовать асинхронные методы (async/await).
3. Реализовать класс-сервис (например, TasksApiClient), который инкапсулирует логику взаимодействия с API (формирование URL, сериализацию, базовую обработку ответов). Основная программа должна работать через этот сервис.
4. Корректно обрабатывать как исключения, так и неуспешные HTTP-статусы.
5. Предоставить пользователю понятное меню и информативные сообщения о результатах операций.

Контрольные вопросы

1. Для чего предназначен класс HttpClient? Какие проблемы могут возникнуть при неправильном использовании (частом создании новых экземпляров)?
2. В чем разница между исключениями (например, HttpRequestException) и неуспешными HTTP-статусами (например, 404 Not Found)? Как их различать в коде?
3. Что такое HttpResponseMessage и какую информацию из него можно получить?
4. Как отправить на сервер данные в формате JSON с помощью HttpClient?
5. Как прочитать и десериализовать JSON-ответ от сервера?

6. Что делает метод `EnsureSuccessStatusCode()`? Когда его использовать безопасно, а когда лучше проверять статус вручную?
7. Какие HTTP-статусы относятся к успешным (2xx), клиентским ошибкам (4xx), серверным ошибкам (5xx)? Приведите примеры.
8. Как установить таймаут для HTTP-запроса? Зачем это нужно?
9. Что такое политика повторных попыток (`retry policy`)? Для каких типов ошибок ее целесообразно применять?

Лабораторная работа №8

«Асинхронный обработчик логов»

Цель работы: Углубить понимание асинхронного программирования в C#, изучить паттерны Producer-Consumer и работу с потоками данных, а также освоить создание эффективных обработчиков событий для работы с большими объемами информации.

Теоретическая часть

Ключевые понятия:

Асинхронное программирование (продвинутый уровень):

1. Контекст синхронизации (SynchronizationContext): Механизм, который позволяет продолжить выполнение асинхронного кода в нужном контексте (например, в UI-потоке). В консольных приложениях контекст по умолчанию – пул потоков.
2. Конфигурация ожидания (ConfigureAwait): Метод ConfigureAwait(false) говорит, что после await не нужно возвращаться в исходный контекст синхронизации. Это повышает производительность и предотвращает взаимоблокировки в библиотечном коде.

Паттерн Producer-Consumer (Производитель-Потребитель):

1. Описание: Классический паттерн для организации многопоточной обработки данных. Один или несколько потоков (производители) генерируют данные и помещают их в общую очередь. Другие потоки (потребители) извлекают данные из очереди и обрабатывают их.
2. Преимущества: Разделение генерации и обработки данных, возможность балансировки нагрузки, сглаживание пиковых нагрузок.
3. Реализация в C#: Требуется потокобезопасная коллекция.
Потокобезопасные коллекции (System.Collections.Concurrent):
 1. ConcurrentQueue<T>: Потокобезопасная очередь, работающая по принципу FIFO. Идеально подходит для паттерна Producer-

Consumer. Методы: Enqueue (добавление), TryDequeue (попытка извлечения).

2. BlockingCollection<T>: Обертка над потокобезопасной коллекцией (по умолчанию над ConcurrentQueue<T>), которая предоставляет механизм блокировки. Если коллекция пуста, потребитель, вызывающий метод Take(), будет заблокирован до появления нового элемента. Если коллекция достигла максимального размера, производитель блокируется. Реализует паттерн "ограниченный буфер".
3. ConcurrentDictionary<TKey, TValue>: Потокобезопасная версия словаря.

Каналы (System.Threading.Channels):

1. Определение: Современный и более производительный способ реализации паттерна Producer-Consumer, появившийся в .NET Core.
2. Класс Channel<T>: Представляет собой очередь, которая может быть ограниченной (Bounded) или неограниченной (Unbounded).
3. Объекты ChannelReader<T> и ChannelWriter<T>: Позволяют производителям и потребителям безопасно работать с каналом. Поддерживают асинхронные операции (WaitToReadAsync, ReadAsync, WriteAsync), что делает их идеальными для использования в связке с async/await.
4. Завершение канала: Производитель может сигнализировать о завершении работы, вызвав Writer.Complete(), после чего потребители узнают об этом через Reader.Completion.

Задание

Общая постановка задачи:

Разработать приложение "Асинхронный обработчик логов", которое имитирует поступление большого количества логов (например, от нескольких источников) и обрабатывает их в реальном времени. Приложение должно демонстрировать преимущества асинхронной обработки и паттерна Producer-Consumer.

Базовый сценарий:

1. Несколько "производителей" (имитация сервисов или модулей) генерируют логи с разной интенсивностью (например, с случайными интервалами).
2. Логи помещаются в общую очередь (буфер).
3. Несколько "потребителей" (обработчиков) параллельно извлекают логи из очереди и "обрабатывают" их (например, записывают в файл, анализируют, выводят на экран с цветовой кодировкой по уровню важности).

Требования к реализации:

1. Обязательное использование асинхронных методов (async/await) и задач (Task).
2. Потокбезопасная очередь или канал для обмена данными между производителями и потребителями.
3. Демонстрация параллельной работы: производители и потребители должны работать одновременно.
4. Корректная обработка завершения работы: программа не должна "виснуть" после окончания генерации логов.
5. Логирование (в консоль или файл) ключевых событий: старт/стоп производителей, начало/конец обработки.
6. Обработка возможных исключений в задачах (например, через Task.WhenAll с try-catch).

Контрольные вопросы

1. В чем суть паттерна Producer-Consumer? Для решения каких задач он применяется?
2. Чем ConcurrentQueue<T> отличается от обычной Queue<T>? Почему нельзя использовать обычную очередь в многопоточной среде?
3. Что такое BlockingCollection<T> и в каких сценариях ее использование оправдано? Что такое "ограниченный буфер"?
4. Что такое System.Threading.Channels? В чем его преимущества перед BlockingCollection<T>?

5. Как корректно завершить работу потребителей, когда производители закончили работу? Что такое "ядовитая пилюля" (poison pill)?
6. Что произойдет, если производители будут генерировать данные быстрее, чем потребители успевают их обрабатывать, при использовании ограниченного буфера?
7. Что такое `ConfigureAwait(false)` и когда его нужно использовать?
8. Чем асинхронное программирование (`async/await`) отличается от многопоточности? Как они связаны в контексте данной работы?

Лабораторная работа №9

«Файловый обмен через сокеты»

Цель работы: Изучить особенности передачи больших объемов данных (файлов) по сети, освоить протокол прикладного уровня для управления передачей файлов и научиться решать проблемы фрагментации данных и контроля целостности.

Теоретическая часть

Ключевые понятия:

Особенности передачи файлов по TCP:

1. Фрагментация: TCP – это потоковый протокол. Данные разбиваются на пакеты на уровне IP, и нет гарантии, что один Send вызовет одно Receive. Отправленные данные могут быть склеены (несколько отправок в одном приеме) или разбиты (одна отправка считается за несколько приемов).
2. Решение: Необходимо реализовать протокол прикладного уровня, который позволяет точно определить границы передаваемых данных.

Протокол передачи файла:

Для надежной передачи файла необходимо договориться о структуре сообщений. Обычно передача состоит из двух этапов: передача метаданных (информации о файле) и передача самого содержимого.

Вариант А (Заголовок фиксированной длины + данные):

1. Клиент отправляет серверу заголовок фиксированного размера (например, 8 байт: 4 байта на длину имени файла + 4 байта на размер файла).
2. Сервер читает ровно 8 байт, извлекает длину имени и размер.
3. Клиент отправляет имя файла (переменной длины).
4. Сервер читает имя (зная его длину).
5. Клиент отправляет содержимое файла.
6. Сервер читает содержимое, зная размер файла.

Вариант Б (JSON-метаинформация + данные):

1. Клиент отправляет JSON-объект с метаданными (имя, размер, возможно хеш).
2. Так как JSON – строка переменной длины, нужно сначала отправить его длину (например, первые 4 байта).
3. Сервер читает длину, затем читает JSON, десериализует его.
4. Затем сервер читает содержимое файла (ровно столько байт, сколько указано в размере).

Контроль целостности:

1. При передаче по сети данные могут быть повреждены из-за ошибок на линии.
2. Хеш-суммы (Hash): Для проверки целостности переданного файла можно вычислить его хеш (например, MD5, SHA256) на стороне отправителя и включить его в метаданные. Получатель после получения файла вычисляет его хеш и сравнивает. Если хеши не совпадают – передача была с ошибкой.
3. Класс `System.Security.Cryptography.HashAlgorithm` предоставляет методы для вычисления хешей.

Прогресс передачи:

1. При передаче больших файлов важно информировать пользователя о прогрессе.
2. В цикле чтения из файлового потока и записи в сетевой поток можно подсчитывать количество переданных байт и вычислять процент выполнения.

Управление подключениями:

1. Файловый сервер должен корректно обрабатывать множество одновременных подключений (асинхронно).
2. Важно освобождать ресурсы (`NetworkStream`, `FileStream`, `TcpClient`) после завершения передачи, используя блоки `using` или `Dispose`.

Задание**Общая постановка задачи:**

Разработать приложения клиента и сервера для передачи файлов по протоколу TCP. Сервер должен принимать файлы от

клиентов и сохранять их на диск, а также уметь отдавать список доступных файлов и отправлять их клиентам по запросу. Клиент должен иметь удобный интерфейс для выбора файла для отправки, скачивания файла с сервера и просмотра списка файлов на сервере.

Базовый функционал:

Сервер:

1. Асинхронно принимает подключения.
2. Обрабатывает команды клиента (текстовые команды, либо через JSON).
3. Поддерживаемые команды:
 - LIST – возвращает список файлов в специальной директории сервера.
 - UPLOAD [filename] [filesize] – принимает файл от клиента и сохраняет его.
 - DOWNLOAD [filename] – отправляет запрошенный файл клиенту.
4. Хранит файлы в отдельной папке (например, ServerStorage).

Клиент:

1. Подключается к серверу.
2. Консольное меню:
 - Просмотреть список файлов на сервере.
 - Отправить файл на сервер (пользователь указывает локальный путь).
 - Скачать файл с сервера (пользователь выбирает из списка).
 - Выход.
3. При отправке/скачивании отображает прогресс операции в процентах.

Требования к реализации:

1. Сервер должен быть асинхронным и обрабатывать несколько клиентов одновременно.
2. Использовать протокол прикладного уровня с заголовками (JSON с длиной или бинарный заголовок).
3. Чтение и запись файлов должны производиться буферизированно (не загружать весь файл в память), особенно для

больших файлов. Использовать буферы (например, 8192 байт) и циклы.

4. Отображать прогресс передачи на клиенте.
5. Обрабатывать возможные ошибки: обрыв соединения, отсутствие файла, отказ в доступе, недостаточно места на диске.
6. Предусмотреть корректное завершение работы сервера (остановка прослушивания, завершение текущих передач).

Контрольные вопросы

1. Какие проблемы возникают при передаче файлов по TCP, и как протокол прикладного уровня помогает их решить?
2. Почему нельзя просто отправить файл, вызвав один раз Send для всего массива байт?
3. Опишите структуру вашего протокола передачи файла. Как клиент и сервер понимают, где заканчиваются метаданные и начинаются данные?
4. Для чего нужна проверка целостности файла (хеширование)? Какие алгоритмы хеширования вы знаете?
5. Как организовать отображение прогресса передачи файла на клиенте?
6. Что такое фрагментация TCP и как она влияет на чтение данных из сокета?
7. Как сервер может одновременно обслуживать несколько клиентов, передающих файлы?
8. Какие исключения могут возникнуть при работе с файловой системой в процессе приема файла? Как их обработать, чтобы не "уронить" сервер?
9. В чем преимущество буферизированного чтения/записи (чтения небольшими блоками) при работе с большими файлами?
10. Как можно реализовать докачку файла, если соединение оборвалось в середине передачи?

Лабораторная работа №10

«Форма ввода с валидацией»

Цель работы: Изучить принципы построения графических пользовательских интерфейсов (GUI) на платформе .NET (Windows Forms или WPF), освоить механизмы валидации пользовательского ввода и научиться организовывать взаимодействие между элементами формы и объектной моделью приложения.

Теоретическая часть

Ключевые понятия:

Технологии создания GUI в .NET:

1. Windows Forms (WinForms): Классическая технология для создания настольных приложений под Windows. Основана на классах из пространства имен System.Windows.Forms. Проста в освоении, использует визуальный конструктор форм, где элементы интерфейса (кнопки, текстовые поля) перетаскиваются мышью.
2. WPF (Windows Presentation Foundation): Более современная и гибкая технология, использующая язык XAML для декларативного описания интерфейса. Поддерживает мощные механизмы связывания данных (Data Binding) и стилизации. Требуется более глубокого изучения, но предоставляет большие возможности для создания сложных интерфейсов.

Элементы управления (Controls):

1. TextBox – для ввода текста.
2. Button – для инициирования действий.
3. Label – для вывода статического текста.
4. ComboBox/ListBox – для выбора из списка.
5. DateTimePicker – для ввода даты.
6. DataGridView/DataGrid – для отображения табличных данных.

Событийно-ориентированная модель:

1. GUI-приложения работают по принципу обработки событий. Пользователь взаимодействует с элементами (кликает, вводит

текст), что генерирует события (например, Click, TextChanged).

2. Разработчик создает методы-обработчики (event handlers), которые привязываются к этим событиям и содержат логику, выполняемую в ответ на действие пользователя.

Валидация данных (Data Validation):

Задача: Проверка корректности введенных пользователем данных перед их дальнейшей обработкой (например, перед сохранением в базу данных).

Виды валидации:

1. Валидация на стороне клиента: Выполняется в интерфейсе до отправки данных. Обеспечивает мгновенную обратную связь и снижает нагрузку на сервер (в десктопных приложениях – просто улучшает UX).
2. Валидация на стороне модели: Выполняется бизнес-логикой. Обязательна даже при наличии клиентской валидации для защиты целостности данных.

Подходы к валидации в Windows Forms:

1. Событие Validating: Возникает, когда элемент управления теряет фокус (свойство CausesValidation = true). Позволяет реализовать "отложенную" валидацию. Если данные не прошли проверку, можно отменить потерю фокуса и установить фокус обратно на некорректное поле.
2. Ручная валидация при нажатии кнопки "Сохранить": В обработчике кнопки проверяются значения всех полей, и при ошибках выводится сообщение. Это проще, но менее дружелюбно к пользователю.

Подходы к валидации в WPF:

1. Встроенная валидация через IDataErrorInfo или INotifyDataErrorInfo: Реализуется в классах моделей. WPF автоматически отображает ошибки валидации в интерфейсе (например, красную рамку вокруг поля), если привязать данные через Binding с опцией ValidatesOnDataErrors = True.
2. Исключения в сеттерах свойств: При попытке установить некорректное значение в модели можно генерировать

исключение, которое также может быть отображено в интерфейсе.

Связывание данных (Data Binding):

Механизм, связывающий свойство элемента управления со свойством объекта данных. При изменении значения в интерфейсе автоматически обновляется объект, и наоборот. Это основа паттерна MVVM в WPF и активно используется в WinForms.

Задание

Общая постановка задачи:

Разработать графическое приложение (на выбор: Windows Forms или WPF), представляющее собой форму для ввода и редактирования информации о некоторой сущности (например, о сотруднике, книге, заказе). Форма должна содержать поля для ввода различных типов данных и обеспечивать их валидацию перед сохранением. Сохраненные данные должны отображаться в списке или таблице на той же форме.

Базовая реализация (для любой технологии):

Сущность: Person (Человек)

1. Id (int, уникальный идентификатор, генерируется автоматически).
2. FullName (string, обязательно, не менее 3 символов).
3. Age (int, от 1 до 120).
4. Email (string, обязателен, должен содержать '@' и '.').
5. Phone (string, опционально, но если введен – должен соответствовать маске номера).
6. RegistrationDate (DateTime, по умолчанию текущая дата).

Функциональность формы:

1. Поля ввода для всех свойств сущности (текстовые поля для имени и email, числовое поле для возраста, поле с маской для телефона, календарь для даты).
2. Кнопка "Добавить"/"Сохранить".
3. Список (ListBox) или таблица (DataGridView/DataGrid) для отображения всех добавленных людей.

4. При выборе человека из списка/таблицы, его данные должны подгружаться в поля для редактирования (с возможностью изменения и сохранения по кнопке "Обновить").
5. Кнопка "Удалить" для удаления выбранной записи.

Требования к валидации:

1. Валидация должна происходить либо при потере фокуса полем (рекомендуется), либо при нажатии кнопки "Сохранить".
2. Некорректные поля должны визуально выделяться (например, красная рамка, подсветка фона) и/или рядом должно появляться сообщение об ошибке.
3. Пока есть ошибки валидации на форме, кнопка "Сохранить" должна быть неактивна (Disabled).
4. При попытке сохранить данные с ошибками, должно появляться всплывающее окно с перечнем ошибок.

Требования к реализации:

1. Приложение должно иметь интуитивно понятный интерфейс.
2. Данные должны храниться в оперативной памяти (коллекция) в течение сеанса работы.
3. Валидация должна быть реализована в соответствии с выбранным вариантом.
4. Кнопка сохранения должна быть заблокирована при наличии ошибок валидации.
5. Приложение должно корректно обрабатывать ввод некорректных данных (буквы в поле возраста и т. д.) без вылетов.

Контрольные вопросы

1. В чем разница между Windows Forms и WPF? Какие преимущества и недостатки у каждой технологии?
2. Что такое событийно-ориентированное программирование? Приведите пример обработки события нажатия на кнопку.
3. Для чего нужна валидация данных на уровне интерфейса, если бизнес-логика все равно будет проверять данные?
4. Опишите компонент ErrorProvider в Windows Forms. Как он используется?

5. Что такое интерфейс `INotifyDataErrorInfo` и для чего он нужен в WPF?
6. В чем разница между валидацией через события `Validating` и ручной проверкой при нажатии на кнопку?
7. Что такое связывание данных (`Data Binding`)? Какие преимущества оно дает при разработке GUI?
8. Что такое паттерн MVVM? Из каких компонентов он состоит и какова роль каждого?
9. Как заблокировать кнопку "Сохранить", пока на форме есть ошибки валидации?
10. Как можно организовать хранение данных между сеансами работы приложения (например, чтобы добавленные люди не пропадали после закрытия программы)?

Лабораторная работа №11

«Таблица с данными из базы данных»

Цель работы: Изучить основы взаимодействия приложений .NET с реляционными базами данных (БД), освоить технологии доступа к данным (ADO.NET и Entity Framework Core) и научиться отображать и редактировать данные из базы в графическом интерфейсе пользователя.

Теоретическая часть

Ключевые понятия:

Технологии доступа к данным в .NET:

ADO.NET: Низкоуровневый набор классов для прямого взаимодействия с БД. Предоставляет максимальный контроль и производительность, но требует написания большого количества кода (соединения, команды, ридеры).

SqlConnection – подключение к БД.

SqlCommand – выполнение SQL-запроса.

SqlDataReader – потоковое чтение результата запроса (только вперед).

DataSet/DataTable – работа с данными в оперативной памяти как с таблицами (отсоединенный режим).

Entity Framework Core (EF Core): Современная объектно-реляционная карта (ORM), которая позволяет работать с базой данных через объекты C#. Автоматически генерирует SQL-запросы, отслеживает изменения и упрощает разработку.

Подходы:

1. Database-First: База данных уже существует, по ней генерируются классы моделей.
2. Code-First: Сначала пишутся классы C# (модели), а по ним создается (или синхронизируется) база данных с помощью миграций.

Основные компоненты EF Core:

1. DbContext: Класс, представляющий сеанс работы с базой данных. Содержит DbSet<T> для каждой таблицы и управляет подключением, отслеживанием изменений, сохранением.
2. DbSet<T>: Представляет коллекцию сущностей определенного типа, которые хранятся в таблице. Позволяет выполнять LINQ-запросы к этой таблице.
3. Миграции (Migrations): Механизм версионирования схемы базы данных. Позволяют создавать и обновлять структуру БД по мере изменения классов моделей, сохраняя уже имеющиеся данные.

LINQ to Entities:

1. Интегрированный язык запросов (LINQ) позволяет писать запросы к базе данных на C#, используя знакомый синтаксис. EF Core транслирует эти запросы в SQL.
2. Пример использования LINQ to Entities:
`var adults = context.Persons.Where(p => p.Age >= 18).ToList();`

Подключение GUI к данным через BindingSource/DataContext:

1. В Windows Forms удобно использовать компонент BindingSource как посредник между коллекцией данных (например, List<T> или DataTable) и элементами управления на форме (DataGridView, текстовые поля). Он упрощает навигацию, сортировку и фильтрацию.
2. В WPF привязка напрямую осуществляется к свойствам ViewModel, которые, в свою очередь, могут получать данные из сервисов, работающих с EF Core.

CRUD операции (Create, Read, Update, Delete):

Стандартный набор операций с данными, который должно поддерживать любое приложение, работающее с БД.

Задание

Общая постановка задачи:

Разработать графическое приложение, подключенное к базе данных SQL Server (или SQLite), которое отображает данные из

таблицы в табличном виде (DataGridView/DataGrid) и позволяет выполнять над ними CRUD-операции через удобный интерфейс.

База данных:

Создайте базу данных (например, CompanyDB) с таблицей Employees, имеющей следующую структуру:

1. Id (INT, Primary Key, Identity/AutoIncrement)
2. FirstName (NVARCHAR(50), NOT NULL)
3. LastName (NVARCHAR(50), NOT NULL)
4. Position (NVARCHAR(100))
5. Salary (DECIMAL(18, 2))
6. HireDate (DATE)

Базовая функциональность:

1. При запуске приложения данные из таблицы Employees загружаются и отображаются в таблице на форме.
2. Пользователь должен иметь возможность:
3. Добавить новую запись (через отдельное окно или поля под таблицей).
4. Редактировать существующую запись (либо прямо в ячейках таблицы, либо в отдельной форме с подгрузкой данных).
5. Удалить выбранную запись (с подтверждением).
6. Все изменения (добавление, редактирование, удаление) должны немедленно сохраняться в базе данных и отражаться в таблице.

Общие требования к реализации:

1. Приложение должно корректно подключаться к базе данных (строка подключения может быть задана в коде или в конфигурационном файле).
2. Таблица, размещенная на форме, должна отображать актуальные данные из БД.
3. Операции изменения данных должны быть защищены от ошибок (обработка исключений БД).
4. Интерфейс должен быть интуитивно понятным: кнопки добавления/удаления, подтверждение удаления.
5. При использовании EF Core обязательно применение миграций для создания/обновления схемы БД.

Контрольные вопросы

1. Что такое ADO.NET? Какие основные классы используются для работы с БД?
2. Что такое ORM? Какие преимущества дает использование Entity Framework Core по сравнению с прямым использованием ADO.NET?
3. Объясните разницу между подходами Database-First и Code-First в EF Core.
4. Что такое DbContext и DbSet<T>? Какую роль они играют?
5. Что такое миграции в EF Core и для чего они нужны?
6. Как выполнить простой SQL-запрос через SqlCommand в ADO.NET?
7. Как в ADO.NET организовать пакетное обновление изменений из DataTable обратно в БД?
8. Что такое BindingSource в Windows Forms и для чего он используется?
9. Как в WPF организовать уведомление интерфейса об изменении коллекции (например, при добавлении нового элемента)?
10. Какие меры предосторожности необходимо принимать для защиты от SQL-инъекций при работе с ADO.NET? Как EF Core помогает в этом?

Лабораторная работа №12 **«Интеграция GUI с бизнес-логикой»**

Цель работы: Изучить принципы интеграции графического пользовательского интерфейса с бизнес-логикой приложения, освоить паттерны проектирования для разделения ответственности (MVVM, MVP, Presentation Model) и научиться создавать масштабируемые и тестируемые приложения с четкой архитектурой.

Теоретическая часть

Ключевые понятия:

Архитектурные паттерны для GUI:

1. Проблема: Интерфейс содержит логику отображения (как показать данные), бизнес-логику (что делать с данными) и логику доступа к данным. Смешивание всего в коде формы (code-behind) приводит к спагетти-коду, который сложно поддерживать и тестировать.
2. Решение: Архитектурные паттерны, разделяющие приложение на слои с четкими обязанностями.

Паттерн Model-View-ViewModel (MVVM):

1. Стандарт для WPF и XAML-платформ. Максимально использует возможности связывания данных.
2. Model (Модель): Содержит данные и бизнес-логику (сущности, сервисы, репозитории). Не знает о существовании View и ViewModel.
3. View (Представление): Визуальный слой (XAML). Отображает данные и отправляет команды пользователя. Связан с ViewModel через DataContext.
4. ViewModel: Сердце паттерна. Содержит состояние и команды для View. Преобразует данные из Model в форму, удобную для отображения (например, через реализацию INotifyPropertyChanged). Не знает о конкретных элементах управления, только о данных и командах.
5. Преимущества: Тестируемость, разделение ответственности, возможность параллельной работы дизайнера и разработчика.

Паттерн Model-View-Presenter (MVP):

1. Часто используется в Windows Forms (хотя применим и везде).
2. Model: Данные и бизнес-логика.
3. View: Пассивный интерфейс, который только отображает данные и передает события пользователя Presenter'у. Реализует интерфейс IView.
4. Presenter: Содержит логику представления. Получает события от View, взаимодействует с Model и обновляет View через его интерфейс. Presenter знает о View (через интерфейс), View знает о Presenter.

Связывание данных (Data Binding) и уведомления об изменениях:

1. INotifyPropertyChanged: Интерфейс, который требует реализации события PropertyChanged. Когда свойство в ViewModel/Model меняется, объект генерирует это событие, и привязанный элемент управления автоматически обновляет отображение.
2. ObservableCollection<T>: Специальная коллекция, которая генерирует события при добавлении, удалении или изменении элементов. Идеальна для привязки к спискам (ListBox, DataGrid).

Команды (Commands):

1. Вместо обработки событий клика в WPF используются команды (ICommand). Команда инкапсулирует действие (логику) и условие, может ли оно быть выполнено.
2. Позволяет автоматически управлять доступностью элементов интерфейса (например, кнопка неактивна, если команда не может выполняться).

Внедрение зависимостей (Dependency Injection):

1. Механизм, при котором зависимости класса (например, репозиторий, сервис) передаются извне, а не создаются внутри.
2. Облегчает тестирование (можно подставить mock-объекты) и делает архитектуру гибкой.

Задание

Общая постановка задачи:

Модернизировать приложение из лабораторной работы №10 или №11, применив один из архитектурных паттернов (MVVM или MVP). Приложение должно быть четко разделено на слои: Model (бизнес-логика и данные), View (интерфейс) и ViewModel/Presenter (связующее звено). Добавить тестирование ключевых сценариев.

Базовый сценарий:

Возьмите за основу приложение "Управление сотрудниками" из ЛР №11 (или форму ввода из ЛР №10). Оно должно:

1. Отображать список сотрудников из базы данных (или коллекции в памяти).
2. Позволять добавлять, редактировать и удалять сотрудников.
3. Иметь валидацию вводимых данных.

Задача по рефакторингу/разработке:

Разделить приложение на слои в соответствии с выбранным паттерном. Весь код, не связанный напрямую с отображением, должен быть вынесен из файлов формы (code-behind) в отдельные классы.

Требования к реализации:

1. Четкое разделение на слои согласно выбранному паттерну.
2. View (форма/окно) не должно содержать логики обработки данных (только вызовы методов Presenter'a/ViewModel).
3. Валидация данных должна быть реализована на уровне Model или ViewModel.
4. Все зависимости должны передаваться через конструктор (Dependency Injection).
5. Приложение должно сохранять всю функциональность исходного варианта.

Контрольные вопросы

1. В чем заключается основная проблема смешивания UI-логики и бизнес-логики в одном классе (code-behind)?
2. Опишите паттерн MVVM. Какова роль каждого компонента (Model, View, ViewModel)?

3. Опишите паттерн MVP. Чем он отличается от MVVM?
4. Что такое INotifyPropertyChanged и для чего он нужен?
5. Что такое ObservableCollection<T> и чем она отличается от обычного List<T>?
6. Что такое команды (ICommand) и какие преимущества они дают по сравнению с обработкой событий?
7. Как организовать валидацию данных в ViewModel при использовании MVVM?
8. Что такое Dependency Injection и как он помогает в тестировании?
9. Как протестировать Presenter (в MVP) или ViewModel (в MVVM) без создания реального окна?
10. Какие преимущества дает использование MVVM Toolkit (или аналогичных библиотек) по сравнению с ручной реализацией?

Лабораторная работа №13

«Система авторизации»

Цель работы: Изучить принципы построения систем аутентификации и авторизации в приложениях, освоить механизмы безопасного хранения паролей, управления доступом на основе ролей (RBAC) и интеграции системы безопасности в графическое приложение.

Теоретическая часть

Ключевые понятия:

Аутентификация (Authentication):

Процесс проверки подлинности пользователя – доказательство того, что пользователь является тем, за кого себя выдает.

Обычно выполняется через предъявление учетных данных: логин/пароль, биометрия, токен и т. д.

Авторизация (Authorization):

Процесс проверки прав доступа – определение того, какие ресурсы и операции доступны аутентифицированному пользователю.

Ролевая модель доступа (RBAC - Role-Based Access Control):

Распространенная модель авторизации, где права доступа группируются в роли, а роли назначаются пользователям.

Пользователь -> Роль(и) -> Права (разрешения на операции).

Хеширование (Hashing): Применение односторонней функции к паролю. По хешу невозможно восстановить исходный пароль. При аутентификации хеш введенного пароля сравнивается с сохраненным хешем.

Соль (Salt): Случайная строка, добавляемая к паролю перед хешированием. Используется для защиты от радужных таблиц (rainbow tables) – предвычисленных словарей хешей. Для каждого пользователя генерируется своя уникальная соль.

Классы .NET: System.Security.Cryptography содержит алгоритмы хеширования: SHA256, SHA512, PBKDF2 (Rfc2898DeriveBytes – рекомендуется).

Управление сессией:

После успешной аутентификации приложение должно "запомнить" пользователя на время его работы.

В десктопных приложениях проще всего хранить объект текущего пользователя в статическом классе Session или передавать его между формами.

Принцип наименьших привилегий:

Пользователь должен иметь только те права, которые необходимы ему для выполнения его работы, и не больше.

Безопасность на уровне интерфейса:

Элементы управления (кнопки, пункты меню) должны автоматически становиться недоступными (Disabled) или скрытыми, если у текущего пользователя нет прав на соответствующее действие.

Это улучшает пользовательский опыт, но не является заменой авторизации на уровне бизнес-логики (проверка прав должна дублироваться в коде).

Задание**Общая постановка задачи:**

Дополнить приложение из лабораторной работы №12 системой аутентификации и авторизации. Приложение должно иметь окно входа, хранить пользователей с захешированными паролями, поддерживать роли (например, "Admin" и "User") и ограничивать доступ к функциональности в зависимости от роли пользователя.

База данных пользователей:

Создайте таблицу Users (или добавьте в существующую БД) со следующими полями:

1. Id (INT, Primary Key)
2. Username (NVARCHAR(50), уникальный)
3. PasswordHash (NVARCHAR(256)) – хеш пароля с солью
4. Salt (NVARCHAR(128)) – соль для хеширования
5. Role (NVARCHAR(50)) – роль пользователя (например, "Admin", "User")

Базовая функциональность:

1. При запуске приложения появляется окно входа (Login Form) с полями "Имя пользователя" и "Пароль".
2. После успешного входа открывается главное окно приложения (из ЛР №12), и в нем отображается имя текущего пользователя и его роль.
3. Если пользователь не прошел аутентификацию (нажал "Отмена" или ввел неверные данные), приложение завершается.
4. В главном окне некоторые функции должны быть доступны только администраторам (например, кнопка "Управление пользователями", "Удалить сотрудника" или "Редактировать зарплату").
5. Должна быть возможность выйти из системы (Logout), после чего приложение возвращается к окну входа (без завершения программы).

Требования к реализации:

1. Пароли должны храниться в БД только в виде хеша с солью (не в открытом виде).
2. При аутентификации сравниваются хеши, сам пароль нигде не сохраняется.
3. Проверка прав доступа должна выполняться на двух уровнях: уровень интерфейса, уровень бизнес-логики.
4. Сессия пользователя должна храниться в потокобезопасном статическом классе или передаваться через Dependency Injection.
5. Окно входа должно быть модальным и открываться перед главным окном.
6. Должна быть предусмотрена обработка ошибок входа (неверный логин/пароль) с выводом сообщения пользователю.

Контрольные вопросы

1. Чем отличается аутентификация от авторизации? Приведите примеры.
2. Что такое хеширование паролей? Зачем нужна "соль"?

3. Почему нельзя хранить пароли в открытом виде или использовать обратимые алгоритмы шифрования?
4. Что такое ролевая модель доступа (RBAC)? Какие преимущества она дает?
5. Как в приложении организовать хранение информации о текущем вошедшем пользователе?
6. Почему недостаточно просто скрыть кнопку на форме для защиты функциональности? Какую дополнительную проверку нужно добавить?
7. Как можно реализовать автоматическую неактивность кнопок в WPF в зависимости от роли пользователя?
8. Какие алгоритмы хеширования считаются безопасными для хранения паролей? Почему MD5 или простой SHA1 не рекомендуются?
9. Что такое "принцип наименьших привилегий" и как он применяется при проектировании систем?
10. Как можно защитить приложение от атак перебора паролей (brute-force)? Предложите простые механизмы защиты.

Лабораторная работа №14

«Кеширующий клиент»

Цель работы: Изучить принципы кэширования данных для оптимизации производительности приложений, освоить различные стратегии кэширования (in-memory, distributed) и реализовать клиент с интеллектуальным управлением кэшем для работы с удаленным источником данных.

Теоретическая часть

Ключевые понятия:

Кэширование (Caching):

Определение: Процесс сохранения часто используемых данных в высокоскоростном хранилище (кэше) для ускорения последующего доступа к ним.

Цель: Снижение задержек, уменьшение нагрузки на основной источник данных (базу данных, внешний API), повышение пропускной способности системы.

Виды кэширования по месту хранения:

1. In-Memory Cache (Кэш в памяти): Данные хранятся в оперативной памяти самого приложения. Самый быстрый, но ограничен объемом памяти и "живет", пока работает приложение (или пока не истек срок). Пример: MemoryCache в .NET.
2. Distributed Cache (Распределенный кэш): Данные хранятся во внешнем сервисе (Redis, Memcached), общем для нескольких экземпляров приложения. Позволяет масштабировать приложение горизонтально. Пример: IDistributedCache в .NET.
3. HTTP-кэширование: Кэширование на уровне HTTP-протокола (браузеры, прокси-серверы).

Стратегии (политики) кэширования:

Срок жизни (TTL – Time-To-Live): Данные хранятся в кэше в течение заданного времени, после чего удаляются. Самый простой и распространенный подход.

Вытеснение (Eviction): Когда кэш переполняется, данные удаляются по определенному алгоритму:

1. LRU (Least Recently Used): Удаляются данные, к которым дольше всего не обращались.
2. LFU (Least Frequently Used): Удаляются данные, которые используются реже всего.
3. FIFO (First In First Out): Удаляются самые старые данные.

Скользящее окно (Sliding Expiration): Срок жизни данных продлевается каждый раз, когда к ним обращаются.

Инвалидация (Invalidation): Принудительное удаление или обновление данных в кэше при изменении исходных данных (например, при обновлении записи в БД). Самая сложная, но и самая точная стратегия.

Кэширование в .NET:

Класс `MemoryCache` (пространство имен `System.Runtime.Caching/Microsoft.Extensions.Caching.Memory`).

Предназначен для in-memory-кэширования данных.

Позволяет задавать приоритеты элементов, сроки их жизни, определять зависимости от других элементов или файлов.

В современных приложениях рекомендуется использовать `IMemoryCache` из `Microsoft.Extensions.Caching.Memory` (часть `Microsoft.Extensions`).

Паттерн "Cache-Aside" (сквозное чтение):

1. Приложение пытается получить данные из кэша.
2. Если данные есть (cache hit) – возвращает их.
3. Если данных нет (cache miss) – загружает их из основного источника (БД, API), сохраняет в кэш и возвращает.
4. "Read-Through" / "Write-Through": Кэш сам отвечает за загрузку и запись данных в источник (более сложные паттерны).

Проблемы кэширования:

1. Устаревание данных (Stale Data): Данные в кэше могут не соответствовать актуальному состоянию источника.
2. "Cache Stampede" (Шквал запросов к кэшу): При опустошении кэша (например, по истечению срока хранения данных) множество одновременных запросов могут обратиться к основному источнику данных, создавая пиковую нагрузку на

него. Решение: применение блокировки или механизма "только один запрос обновляет кэш".

Задание

Общая постановка задачи:

Разработать клиентское приложение (консольное или с GUI), которое обращается к удаленному источнику данных (например, к API из лабораторной работы №6 или к базе данных) и реализует интеллектуальное кэширование полученных данных для оптимизации производительности. Приложение должно демонстрировать различные стратегии кэширования и показывать статистику попаданий в кэш (cache hit/miss).

Базовый сценарий:

Создайте приложение, которое позволяет получать информацию о сущностях (например, о сотрудниках, задачах, продуктах) по их идентификатору. Приложение должно сначала проверять наличие данных в кэше и только при отсутствии обращаться к "медленному" источнику (имитация задержки через Thread.Sleep или реальный API).

Требования к реализации:

1. Использовать Microsoft.Extensions.Caching.Memory.IMemoryCache (установить NuGet-пакет Microsoft.Extensions.Caching.Memory).
2. Четко разделить сервис доступа к данным (медленный) и сервис с кэшированием (декоратор).
3. Реализовать как минимум одну стратегию истечения срока жизни.
4. Пользователь должен видеть разницу во времени ответа при обращении к кэшу и к источнику.
5. Корректно обрабатывать ситуации, когда данные не найдены в источнике (не кэшировать null без необходимости).

Контрольные вопросы

1. Что такое кэширование и для каких целей оно применяется в информационных системах?

2. В чем разница между in-memory и distributed кэшем? В каких случаях какой тип следует выбирать?
3. Опишите паттерн Cache-Aside (сквозное чтение). Каковы его преимущества и недостатки?
4. Что такое TTL (Time-To-Live) и зачем он нужен?
5. Чем отличается абсолютный срок жизни (Absolute Expiration) от скользящего окна (Sliding Expiration)?
6. Какие проблемы могут возникнуть при использовании кэша (устаревание данных, cache stampede)?
7. Что такое "cache stampede" (шквал запросов к кэшу) и как с ним бороться?
8. Как можно организовать инвалидацию кэша при изменении данных в источнике?
9. Что такое hit rate (процент попаданий) и как он характеризует эффективность кэша?
10. Какие классы в .NET предоставляют функциональность для in-memory кэширования?

Лабораторная работа №15

«Шаблон Singleton для конфигурации»

Цель работы: Изучить порождающий паттерн проектирования Singleton (Одиночка), освоить различные способы его потокобезопасной реализации в C#, а также применить паттерн для создания глобального объекта конфигурации приложения, обеспечивающего единую точку доступа к настройкам.

Теоретическая часть

Ключевые понятия:

Паттерн Singleton (Одиночка):

Определение: Порождающий паттерн проектирования, гарантирующий, что у класса есть только один экземпляр, и предоставляющий глобальную точку доступа к этому экземпляру.

Назначение:

1. Контроль доступа к общим ресурсам (например, файл конфигурации, подключение к базе данных, логгер).
2. Снижение потребления памяти за счет исключения создания множества одинаковых объектов.
3. Обеспечение согласованного состояния (все части программы работают с одним и тем же экземпляром).

Реализация Singleton в C#:

Проблема: В многопоточной среде два потока могут одновременно создать два экземпляра класса. Это нарушает принцип одиночки.

Конфигурация приложения:

1. Источники конфигурации: JSON-файлы (appsettings.json), XML-файлы, переменные окружения, аргументы командной строки, база данных.
2. Жизненный цикл конфигурации: Обычно загружается при запуске приложения и редко меняется во время работы (или меняется, но требует уведомления компонентов).
3. Паттерн "Options" в .NET Core: В современных приложениях рекомендуется использовать встроенный механизм

IOptions<T> с внедрением зависимостей, но Singleton по-прежнему актуален для простых сценариев или обратной совместимости.

Интеграция Singleton с конфигурацией:

1. Класс-одиночка AppConfig или ConfigurationManager загружает настройки из файла при первом обращении.
2. Предоставляет свойства для доступа к отдельным настройкам (строка подключения, таймауты, пути к файлам).
3. Может поддерживать перезагрузку конфигурации (например, следить за изменением файла).

Критика Singleton:

1. Нарушает принцип единственной ответственности (класс отвечает и за свою основную логику, и за контроль количества экземпляров).
2. Создает скрытые зависимости (глобальное состояние), что затрудняет тестирование (юнит-тесты становятся зависимыми от глобального состояния).
3. В современных приложениях предпочтительнее использовать Dependency Injection с регистрацией синглтон-сервиса (services.AddSingleton<T>()), что дает те же гарантии, но с лучшей тестируемостью.

Задание

Общая постановка задачи:

Разработать класс конфигурации приложения, реализованный как потокобезопасный Singleton. Класс должен загружать настройки из внешнего источника (JSON-файла, XML-файла или переменных окружения) и предоставлять доступ к ним из любой точки приложения. Продемонстрировать использование конфигурации в нескольких модулях программы, убедившись, что все они работают с одним экземпляром.

Базовая структура конфигурации:

Создайте файл конфигурации (например, appsettings.json).

Общие требования к реализации:

1. Класс-одиночка должен быть потокобезопасным.

2. Конструктор класса должен быть приватным.
3. Загрузка данных из файла должна происходить только один раз (при первом обращении к экземпляру).
4. Обрабатывать возможные ошибки: отсутствие файла, некорректный JSON (в таких случаях либо выбрасывать исключение, либо использовать значения по умолчанию).
5. Продемонстрировать использование конфигурации в нескольких частях программы.

Контрольные вопросы

1. Что такое паттерн Singleton? Для решения каких задач он применяется?
2. Какие проблемы могут возникнуть при реализации Singleton в многопоточной среде?
3. Опишите различные способы реализации потокобезопасного Singleton в C#. Какой способ считается наиболее предпочтительным и почему?
4. Для чего нужен Lazy<T> и как он помогает в реализации Singleton?
5. Какие недостатки имеет паттерн Singleton с точки зрения тестирования и поддержки кода?
6. Чем отличается использование паттерна Singleton от регистрации сервиса как синглтона в контейнере Dependency Injection?
7. Как можно организовать перезагрузку конфигурации без перезапуска приложения? Какие при этом возникают сложности?
8. Где в реальных информационных системах может применяться Singleton-конфигурация?
9. Что такое "глобальное состояние" и почему оно считается вредным?
10. Как бы вы организовали хранение конфигурации, если бы разные модули приложения требовали разные "срезы" настроек?

Лабораторная работа №16

«Фабрика подключений к базам данных»

Цель работы: Изучить порождающий паттерн проектирования Factory Method (Фабричный метод) и его применение для создания гибкой и расширяемой системы подключения к различным типам баз данных, обеспечивающей легкую смену систем управления базами данных (СУБД) без изменения основного кода приложения.

Теоретическая часть

Ключевые понятия:

Паттерн Factory Method (Фабричный метод):

Определение: Порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

Назначение: Паттерн используется, когда классу заранее неизвестно, объекты каких подклассов ему нужно создавать, или когда класс спроектирован так, чтобы создание объектов делегировалось подклассам.

Структура:

1. **Creator (Создатель):** Абстрактный класс или интерфейс, объявляющий фабричный метод, возвращающий объект типа Product.
2. **ConcreteCreator (Конкретный создатель):** Переопределяет фабричный метод для создания конкретного продукта.
3. **Product (Продукт):** Общий интерфейс для объектов, создаваемых фабричным методом.
4. **ConcreteProduct (Конкретный продукт):** Реализует интерфейс Product.

Применение в контексте подключения к БД:

Приложение может работать с разными СУБД (SQL Server, PostgreSQL, SQLite, MySQL) в разных средах (разработка, тестирование, продакшн).

Использование фабрики позволяет:

1. Инкапсулировать логику создания подключения и команд.
2. Изолировать код приложения от конкретных реализаций ADO.NET (например, SqlConnection vs NpgsqlConnection).
3. Легко добавлять поддержку новых БД без изменения существующего кода, работающего через интерфейсы.
4. Абстрактная фабрика (Abstract Factory) для семейства объектов:
5. Для работы с БД требуется не только соединение (IDbConnection), но и команда (IDbCommand), адаптер (IDbDataAdapter), параметры (IDbDataParameter).
6. Abstract Factory предоставляет интерфейс для создания целого семейства связанных или зависимых объектов. В данном случае фабрика будет создавать все необходимые объекты для работы с конкретной БД.

Интерфейсы System.Data.Common:

1. Пространство имен System.Data.Common содержит абстрактные классы (DbConnection, DbCommand, DbDataAdapter), которые являются базовыми для всех провайдеров ADO.NET.
2. Класс DbProviderFactory – это реализация абстрактной фабрики от Microsoft, которая умеет создавать все объекты для конкретного провайдера (например, SqlClientFactory, NpgsqlFactory). Можно использовать его как основу.

Преимущества подхода:

1. Гибкость: Смена БД требует изменения только в одном месте – в коде, выбирающем конкретную фабрику.
2. Тестируемость: Можно создавать mock-фабрики для модульного тестирования.
3. Соответствие принципам SOLID: Принцип открытости/закрытости (Open/Closed) – классы открыты для расширения (новыми БД), но закрыты для модификации. Принцип инверсии зависимостей (Dependency Inversion) – код зависит от абстракций (IDbConnection), а не от конкретных реализаций.

Задание

Общая постановка задачи:

Разработать систему фабрик для создания подключений к различным базам данных. Реализовать абстрактную фабрику `IDatabaseFactory`, конкретные фабрики для двух разных СУБД (например, SQL Server и SQLite), а также класс-менеджер, который на основе конфигурации предоставляет нужную фабрику. Продемонстрировать работу приложения, выполняющего одинаковые операции с разными БД без изменения основного кода.

Базовая структура:

Создайте интерфейс `IDatabaseFactory` с методами:

1. `IDbConnection CreateConnection()`
2. `IDbCommand CreateCommand()`
3. `IDbDataAdapter CreateDataAdapter()`
4. `IDbDataParameter CreateParameter()`

Реализуйте конкретные фабрики:

1. `SqlServerFactory` – использует классы из `System.Data.SqlClient`.
2. `SQLiteFactory` – использует классы из `Microsoft.Data.Sqlite` (потребуется установить NuGet-пакет).

Создайте класс `DatabaseFactoryManager`, который:

1. Читает тип БД из конфигурационного файла (или аргументов командной строки).
2. Возвращает соответствующую фабрику (экземпляр `IDatabaseFactory`).

Демонстрационное приложение:

Создайте консольное приложение, которое:

1. Получает от пользователя тип БД (или читает из конфигурационного файла (конфига)).
2. Получает соответствующую фабрику от менеджера.
3. Выполняет простые операции с БД через интерфейсы (создание таблицы, вставка тестовых данных, чтение и вывод результатов), используя только фабрику и стандартные интерфейсы `System.Data`.

Требования к реализации:

1. Все операции с БД должны выполняться через интерфейсы (IDbConnection, IDbCommand), а не через конкретные классы.
2. Фабрика должна создавать полностью настроенные объекты (например, команда уже должна быть ассоциирована с соединением? Это вопрос дизайна – можно оставить это на усмотрение вызывающего кода).
3. Строка подключения должна передаваться в метод CreateConnection() как параметр или храниться внутри фабрики (и браться из конфига).
4. Обрабатывать возможные ошибки подключения к БД.
5. Для SQLite файл базы данных должен создаваться автоматически, если его нет.

Контрольные вопросы

1. Что такое паттерн Factory Method? Для решения каких задач он применяется?
2. Чем Factory Method отличается от Abstract Factory? Какой паттерн лучше подходит для создания семейства связанных объектов (соединение, команда, адаптер)?
3. Какие преимущества дает использование интерфейсов IDbConnection, IDbCommand вместо конкретных классов (SqlConnection, SqlCommand)?
4. Что такое DbProviderFactory в .NET и как его можно использовать?
5. Как изменится архитектура приложения, если потребуется добавить поддержку новой СУБД (например, PostgreSQL)?
6. Где лучше всего хранить тип используемой БД и строку подключения (в коде, в конфигурационном файле, в переменных окружения)? Почему?
7. Как можно протестировать класс, использующий фабрику подключений, без обращения к реальной базе данных?
8. Какие принципы SOLID соблюдаются при использовании фабричного метода?

9. В чем разница между фабрикой и простым классом-хелпером со статическими методами?
10. Как можно организовать пул соединений при использовании фабричного подхода?

Лабораторная работа №17 **«Модуль локализации»**

Цель работы: Изучить принципы интернационализации (i18n) и локализации (l10n) программного обеспечения, освоить встроенные механизмы .NET для работы с ресурсами и культурами, а также реализовать модуль, обеспечивающий динамическое переключение языка интерфейса в приложении.

Теоретическая часть

Ключевые понятия:

Интернационализация (i18n) и локализация (l10n):

1. Интернационализация: Процесс проектирования и подготовки приложения таким образом, чтобы оно могло быть адаптировано к различным языкам и регионам без изменений в коде.
2. Локализация: Процесс адаптации интернационализованного приложения к конкретному языку и региону путем добавления переведенных текстов, форматирования дат, чисел, валют и учета культурных особенностей.

Культура (Culture) в .NET:

1. System.Globalization.CultureInfo: Класс, представляющий информацию о конкретной культуре (язык, страна, календарь, формат чисел, дат и т. д.).

Типы культур:

2. Нейтральная культура (Neutral): Только язык, без региона (например, "ru", "en", "de").
3. Специфическая культура (Specific): Язык и регион (например, "ru-RU" – русский (Россия), "en-US" – английский (США), "en-GB" – английский (Великобритания)).

Текущая культура:

1. CultureInfo.CurrentCulture – используется для форматирования чисел, дат, валют.
2. CultureInfo.CurrentUICulture – используется для загрузки локализованных ресурсов (переводов интерфейса).

Ресурсные файлы (.resx):

Механизм хранения строк: В .NET для хранения локализованных строк используются XML-файлы с расширением .resx.

Структура: Файл содержит пары "ключ-значение". Ключ – идентификатор строки (например, "HelloMessage"), значение – текст на конкретном языке.

Именованное: Для каждого языка создается отдельный ресурсный файл с суффиксом языка:

1. Resources.resx – ресурсы по умолчанию (обычно английский).
2. Resources.ru.resx – русский перевод.
3. Resources.es.resx – испанский перевод.

Генерация кода: Visual Studio автоматически генерирует класс-обертку (например, Resources), предоставляющий строго типизированный доступ к строкам через статические свойства.

Локализация в Windows Forms:

Свойство Localizable формы (в конструкторе) нужно установить в true.

При изменении свойства Language формы в конструкторе и вводе текстов для элементов управления, Visual Studio создает отдельные .resx-файлы для каждой культуры.

Во время выполнения приложение автоматически загружает тексты из ресурсов, соответствующих CurrentUICulture.

Локализация в WPF:

Используется связывание данных с ресурсами: {x:Static properties:Resources.HelloMessage}.

Для динамического переключения языка без перезапуска требуется реализовать механизм уведомления об изменении культуры (например, через событие или специальный класс-провайдер).

Динамическая смена языка:

Позволяет пользователю изменить язык интерфейса во время работы приложения без перезапуска.

Требует, чтобы все текстовые элементы интерфейса были привязаны к ресурсам и обновлялись при изменении

CurrentUICulture. В Windows Forms это сложнее, в WPF – реализуется через обновление всех привязок.

Форматирование данных:

Даты, числа и валюта должны форматироваться согласно выбранной культуре с использованием методов ToString() с учетом CultureInfo.CurrentCulture.

Задание

Общая постановка задачи:

Модернизировать одно из ранее разработанных приложений (например, из ЛР №10, №11 или №12), добавив в него поддержку многоязычности. Приложение должно поддерживать как минимум два языка (например, русский и английский) с возможностью динамического переключения между ними без перезапуска. Все строки интерфейса, сообщения об ошибках, заголовки колонок и т.д. должны быть вынесены в ресурсные файлы.

Базовая структура:

Создайте ресурсные файлы:

1. Resources.resx – язык по умолчанию (например, английский).
2. Resources.ru.resx – русский перевод.
3. (Опционально) другие языки.

Заполните ресурсы ключами: MainFormTitle, AddButtonText, DeleteButtonText, ConfirmDeleteMessage, ErrorCaption, NameLabel, AgeLabel и т. д. – в зависимости от вашего приложения.

Замените все жёстко заданные (hardcoded) строки в коде и XAML на обращения к ресурсам.

Добавьте возможность переключения языка (например, через меню "Настройки" -> "Язык" или комбобокс на главной форме).

Требования к реализации:

1. Никакие строки, видимые пользователю, не должны быть жёстко заданы в коде (кроме, возможно, имен ключей ресурсов).
2. Ресурсные файлы должны быть корректно организованы (минимум 2 языка).

3. Переключение языка должно быть доступно пользователю через интерфейс.
4. При переключении языка интерфейс должен обновляться (для WPF – динамически, для WinForms – допустимо с перезагрузкой формы).
5. Учитывать особенности форматирования данных.

Контрольные вопросы

1. Чем отличается интернационализация (i18n) от локализации (l10n)?
2. Что такое CultureInfo? В чем разница между CurrentCulture и CurrentUICulture?
3. Для чего используются ресурсные файлы (.resx) в .NET?
4. Как Visual Studio помогает в создании локализованных версий форм Windows Forms?
5. Как организовать локализацию в WPF-приложении? Какие подходы существуют?
6. Как можно реализовать динамическое переключение языка без перезапуска приложения?
7. Почему важно форматировать даты и числа с учетом культуры пользователя?
8. Какие сложности могут возникнуть при локализации приложения с учетом не только языка, но и региональных особенностей (например, валюты, единицы измерения)?
9. Как организовать локализацию сообщений об ошибках, которые могут содержать вставки (форматированные строки)?
10. Как можно протестировать, правильно ли приложение отображается на всех поддерживаемых языках?

Лабораторная работа №18

«Шифрование данных»

Цель работы: Изучить основы криптографии, освоить симметричные и асимметричные алгоритмы шифрования, доступные в .NET, и научиться применять их для защиты конфиденциальных данных в информационных системах.

Теоретическая часть

Ключевые понятия:

Основы криптографии:

1. Шифрование (Encryption): Процесс преобразования открытых данных (plaintext) в нечитаемый вид (ciphertext) с использованием ключа и алгоритма.
2. Дешифрование (Decryption): Обратный процесс восстановления исходных данных из зашифрованных.
3. Ключ (Key): Секретная информация, используемая алгоритмом для шифрования и дешифрования.
4. Симметричное шифрование: Класс криптографических методов, в которых один и тот же ключ применяется как для шифрования, так и для дешифрования данных.

Алгоритмы симметричного шифрования в .NET:

1. AES (Advanced Encryption Standard): Современный стандарт, рекомендуется для большинства задач. Класс Aes (Aes.Create()).
2. DES / 3DES: Устаревшие, не рекомендуются к использованию.

Режимы шифрования (Cipher Modes):

1. CBC (Cipher Block Chaining): Требуется вектор инициализации (IV) для обеспечения уникальности шифротекста даже для одинаковых блоков.
2. ECB (Electronic Codebook): Небезопасен (одинаковые блоки дают одинаковый шифротекст), не рекомендуется.
3. GCM (Galois/Counter Mode): Обеспечивает одновременно шифрование и аутентификацию (проверку целостности).

Асимметричное шифрование:

Использует пару ключей: открытый (public key) для шифрования и закрытый (private key) для дешифрования.

Преимущества: Нет проблемы передачи ключа (открытый ключ можно передавать открыто).

Недостатки: Низкая скорость, не подходит для больших объемов данных.

Алгоритмы в .NET:

RSA: Наиболее распространенный алгоритм. Класс RSA (RSA.Create()).

Применение: Обычно используется для шифрования симметричных ключей или цифровых подписей, а не самих данных.

Гибридная схема: Комбинация симметричного и асимметричного шифрования:

1. Генерируется случайный симметричный ключ (сеансовый ключ).
2. Этим ключом шифруются данные (быстро и эффективно для больших объемов).
3. Сам симметричный ключ шифруется открытым ключом получателя (асимметрично).
4. Зашифрованные данные и зашифрованный ключ отправляются получателю.
5. Получатель расшифровывает симметричный ключ своим закрытым ключом, а затем расшифровывает данные.
6. Используется в протоколах TLS/HTTPS.

Хеширование (краткое напоминание):

Односторонняя функция. Для любого входного сообщения произвольной длины вычисляет строку фиксированной длины (хеш или дайджест). Не является шифрованием, но часто используется вместе с ним для проверки целостности.

*Алгоритмы: SHA256, SHA512.***Реализация криптографии в .NET:**

1. Пространство имен System.Security.Cryptography.
2. Все алгоритмы реализуют паттерн "фабрика" – создание через статические методы Create().

3. Для работы с потоками используются классы CryptoStream.

Задание

Общая постановка задачи:

Разработать приложение (консольное или с GUI), демонстрирующее различные методы шифрования данных. Приложение должно позволять пользователю вводить текст или выбирать файл, выбирать алгоритм шифрования, вводить ключ (или генерировать его), выполнять шифрование и дешифрование, а также сохранять/загружать зашифрованные данные.

Базовая функциональность:

1. Возможность ввода текста для шифрования (или выбора файла).
2. Выбор алгоритма: AES (симметричный) и RSA (асимметричный).
3. Для AES: возможность ввода пароля (из которого будет генерироваться ключ) или генерация случайного ключа.
4. Для RSA: генерация пары ключей (открытый/закрытый) с возможностью сохранения в файл.
5. Отображение зашифрованных данных (в Base64 для читаемости).
6. Возможность дешифрования и проверки соответствия исходным данным.

Требования к реализации:

1. Не использовать устаревшие алгоритмы (DES, MD5).
2. Для симметричного шифрования использовать AES с ключом не менее 256 бит.
3. Для генерации ключа из пароля использовать PBKDF2 (Rfc2898DeriveBytes) с достаточным числом итераций (не менее 10000).
4. Корректно обрабатывать исключения криптографических операций.
5. Для RSA использовать рекомендуемый размер ключа (не менее 2048 бит).

6. При работе с файлами использовать потоки (FileStream, CryptoStream).

Контрольные вопросы

1. В чем разница между симметричным и асимметричным шифрованием? Каковы области применения каждого?
2. Почему AES считается предпочтительным симметричным алгоритмом? Какие размеры ключей поддерживаются?
3. Что такое вектор инициализации (IV) и зачем он нужен в режиме CBC?
4. Почему режим ECB считается небезопасным?
5. Что такое гибридная криптосистема? В каких ситуациях она применяется?
6. Что такое PBKDF2 и зачем он нужен при использовании пароля для шифрования?
7. Чем хеширование отличается от шифрования? Можно ли использовать хеш для восстановления исходных данных?
8. Какие классы в .NET предоставляют функциональность для работы с AES и RSA?
9. Что такое CryptoStream и как он используется для потокового шифрования?
10. Какие меры предосторожности необходимо соблюдать при хранении и передаче ключей шифрования?

Лабораторная работа №19

«Цифровая подпись и HTTPS»

Цель работы: Изучить механизмы обеспечения подлинности и целостности данных с помощью электронной цифровой подписи (ЭЦП), освоить создание и проверку подписей в .NET, а также понять принципы работы протокола HTTPS и его роль в защите клиент-серверного взаимодействия.

Теоретическая часть

Ключевые понятия:

Цифровая подпись (Digital Signature):

Определение: Криптографический механизм, позволяющий доказать подлинность (авторство) и целостность электронного документа.

Свойства:

1. Аутентичность: Подтверждает, что документ подписан именно владельцем закрытого ключа.
2. Целостность: Любое изменение документа после подписания делает подпись недействительной.
3. Неотказуемость (Non-repudiation): Отправитель не может отказаться от факта подписания документа.

Принцип работы:

1. Отправитель вычисляет хеш документа.
2. Хеш шифруется закрытым ключом отправителя – это и есть цифровая подпись.
3. Отправитель отправляет документ и подпись получателю.
4. Получатель вычисляет хеш полученного документа.
5. Расшифровывает подпись открытым ключом отправителя, получая исходный хеш.
6. Сравнивает хеши. Если совпадают – подпись верна.

Алгоритмы цифровой подписи в .NET:

1. RSA: Может использоваться как для шифрования, так и для подписи.

2. DSA (Digital Signature Algorithm): Специализированный алгоритм для подписей.
3. ECDSA (Elliptic Curve DSA): Современный алгоритм на эллиптических кривых, обеспечивающий высокую безопасность при меньшей длине ключа.

Классы: RSACryptoServiceProvider, DSACryptoServiceProvider, ECDsa (рекомендуется).

Сертификаты X.509

Сертификат X.509: Цифровой документ, который связывает открытый ключ с владельцем и подтвержден подписью удостоверяющего центра (CA, Certificate Authority).

Содержит информацию о владельце, открытый ключ, срок действия, имя издателя (CA).

В .NET представлен классом X509Certificate2.

HTTPS (HTTP over TLS): Защищенная версия HTTP, использующая протоколы TLS (Transport Layer Security) или SSL.

Обеспечивает:

1. Шифрование: Все данные между клиентом и сервером шифруются (гибридная схема).
2. Аутентификацию сервера: Клиент проверяет, что сервер – это тот, за кого себя выдает (через сертификат).
3. Аутентификацию клиента (опционально): Сервер может также запросить сертификат у клиента.
4. Целостность данных: Защита от модификации данных в пути.

HTTPS в ASP.NET Core:

1. Для разработки можно использовать самоподписанные сертификаты (Visual Studio автоматически создает их для localhost).
2. Middleware UseHttpsRedirection перенаправляет HTTP-запросы на HTTPS.
3. Атрибут [Authorize] может работать в связке с сертификатами для аутентификации клиентов.

Клиентские сертификаты:

Позволяют серверу идентифицировать клиента не по логину/пароллю, а по сертификату, установленному на машине клиента.

Используется в системах с высокими требованиями к безопасности (например, в банковских системах для доступа сотрудников).

Задание**Общая постановка задачи:**

Разработать приложение, демонстрирующее создание и проверку цифровой подписи для документов. Также создать простое клиент-серверное приложение (на базе ASP.NET Core и консольного клиента), работающее по HTTPS, и опционально добавить аутентификацию с использованием клиентских сертификатов.

Часть 1. Цифровая подпись:

Создайте консольное приложение, которое позволяет:

1. Генерировать пару ключей (RSA или ECDSA) и сохранять их в файлы (закрытый ключ – защищенно, открытый – открыто).
2. Подписывать файл (или текст) с использованием закрытого ключа. Подпись сохранять в отдельный файл (.sig).
3. Проверять подпись для файла с использованием открытого ключа.
4. Продемонстрировать, что изменение даже одного байта в файле делает подпись недействительной.

Часть 2. HTTPS сервер и клиент:

Создайте простое ASP.NET Core Web API (как в ЛР №6) и консольный клиент к нему (как в ЛР №7), но обеспечьте их работу по HTTPS.

Требования к реализации:

1. Для цифровой подписи использовать классы из System.Security.Cryptography (RSA или ECDsa).
2. Закрытые ключи должны храниться в защищенном виде (хотя бы не в открытом тексте в коде). Для простоты можно

использовать экспорт в XML с последующим шифрованием файла.

3. Для работы с сертификатами использовать X509Certificate2.
4. В части HTTPS настроить проект ASP.NET Core на использование HTTPS (по умолчанию так и есть).
5. Обрабатывать ошибки, связанные с недействительными сертификатами и подписями.

Контрольные вопросы

1. Что такое цифровая подпись? Какие свойства документа она обеспечивает?
2. В чем разница между шифрованием документа и его подписанием?
3. Опишите процесс создания и проверки цифровой подписи.
4. Какие алгоритмы цифровой подписи поддерживаются в .NET? Чем RSA отличается от ECDSA?
5. Что такое сертификат X.509? Какую роль в нем играет удостоверяющий центр (CA)?
6. Что такое HTTPS и какие задачи безопасности он решает?
7. Как работает аутентификация сервера в HTTPS? Как клиент проверяет, что сервер подлинный?
8. Что такое клиентские сертификаты и для чего они используются?
9. Как в ASP.NET Core настроить требование клиентского сертификата?
10. Чем самоподписанный сертификат отличается от сертификата, выданного CA? В каких случаях допустимо использовать самоподписанные сертификаты?

Лабораторная работа №20

«Защита от SQL-инъекций»

Цель работы: Изучить уязвимость SQL-инъекций, освоить методы защиты от них при работе с базами данных в .NET-приложениях и научиться применять параметризованные запросы и ORM-средства для обеспечения безопасности.

Теоретическая часть

Ключевые понятия:

SQL-инъекция (SQL Injection):

Определение: Один из самых распространенных и опасных видов атак на веб-приложения и любые системы, взаимодействующие с базами данных. Атака заключается во внедрении (инъекции) произвольного SQL-кода в запрос через пользовательский ввод.

Причина: Неправильная обработка пользовательского ввода, когда данные напрямую конкатенируются (склеиваются) со строкой SQL-запроса.

Последствия:

1. Несанкционированное чтение данных (кража информации).
2. Модификация или удаление данных.
3. Выполнение команд администратора (например, удаление таблиц).
4. Полная компрометация сервера БД.

Методы защиты:

1. Параметризованные запросы (Parameterized Queries): Основной и самый надежный способ защиты. Данные передаются отдельно от кода SQL, СУБД воспринимает их только как значения, а не как исполняемый код.
2. Хранимые процедуры (Stored Procedures): Если вызывать хранимые процедуры и передавать параметры, это также защищает от инъекций (при условии, что внутри процедуры нет динамической сборки SQL).

3. Экранирование (Escaping): Менее надежный метод, заключающийся в экранировании спецсимволов. Легко ошибиться и не учесть все нюансы конкретной СУБД.
4. ORM (Object-Relational Mapping): Современные ORM, такие как Entity Framework Core, практически исключают возможность SQL-инъекций, если разработчик использует LINQ-запросы, а не сырой SQL.

Принцип наименьших привилегий:

Учетная запись (учетка), под которой приложение подключается к БД, должна иметь только необходимые права (например, только SELECT для чтения, отдельная учетка для записи). Это минимизирует ущерб в случае успешной атаки.

Валидация входных данных:

Хотя параметризация решает проблему инъекций, валидация (проверка типа, длины, формата) все равно необходима для бизнес-логики и предотвращения других атак (XSS).

Задание

Общая постановка задачи:

Разработать приложение (консольное или с GUI), которое демонстрирует уязвимость SQL-инъекций и затем реализует защищенные методы работы с БД. Приложение должно позволять пользователю вводить поисковые запросы и другие данные, и показывать разницу между небезопасным и безопасным подходами.

Базовая структура:

1. Создайте базу данных (SQL Server или SQLite) с таблицей Users (Id, UserName, Password, Email, Role).
2. Заполните таблицу тестовыми данными (5–10 пользователей).
3. Реализуйте два режима работы: "Небезопасный" и "Безопасный".

Требования к реализации:

1. В небезопасном режиме чётко покажите, как работает инъекция.
2. В безопасном режиме используйте параметризованные запросы или LINQ.

3. Для демонстрации деструктивных операций (DROP TABLE) используйте либо вывод сгенерированного SQL, либо тестовую копию таблицы/базы.
4. Обязательно обрабатывайте исключения, связанные с SQL.
5. В коде должны быть комментарии, поясняющие, где уязвимость и где защита.

Контрольные вопросы

1. Что такое SQL-инъекция? Опишите механизм атаки.
2. Какие последствия может иметь успешная SQL-инъекция?
3. Почему конкатенация строк при построении SQL-запросов опасна?
4. Как параметризованные запросы защищают от SQL-инъекций?
5. В чем разница между SqlCommand с параметрами и прямой вставкой значений в строку запроса?
6. Как Entity Framework Core защищает от SQL-инъекций при использовании LINQ?
7. Чем отличаются методы FromSqlRaw и FromSqlInterpolated в EF Core?
8. Достаточно ли только экранирования кавычек для защиты от SQL-инъекций? Почему?
9. Какие дополнительные меры безопасности (кроме параметризации) можно применить для защиты БД?
10. Какую роль играет принцип наименьших привилегий при подключении к БД в контексте защиты от инъекций?

Лабораторная работа №21

«OAuth2-клиент»

Цель работы: Изучить протокол авторизации OAuth 2.0, освоить основные потоки (grant types) авторизации, научиться интегрировать .NET-приложения с внешними поставщиками идентификации (Google, GitHub, VK) для реализации входа через сторонние сервисы.

Теоретическая часть

Ключевые понятия:

OAuth 2.0:

Определение: Промышленный стандарт протокола авторизации, который позволяет приложениям получать ограниченный доступ к ресурсам пользователя на других сервисах без передачи логина и пароля пользователя этому приложению.

Основные роли:

1. Resource Owner (Владелец ресурса): Пользователь, владеющий данными.
2. Client (Клиент): Наше приложение, которое хочет получить доступ к данным пользователя.
3. Authorization Server (Сервер авторизации): Сервер, который аутентифицирует пользователя и выдает токены доступа (например, Google Accounts, GitHub OAuth).
4. Resource Server (Сервер ресурсов): Сервер, предоставляющий API с данными пользователя (например, Google API, GitHub API). Часто объединен с сервером авторизации.

Потоки OAuth 2.0 (Grant Types):

1. Authorization Code Flow (Код авторизации): Самый безопасный поток для веб-приложений и мобильных приложений. Клиент получает временный код, который затем обменивает на токен доступа на своем серверном бэкенде.
2. Implicit Flow (Неявный поток): Устаревший, не рекомендуется. Использовался для SPA.

3. Resource Owner Password Credentials Flow: Пользователь передает логин/пароль клиенту, клиент отправляет их на сервер авторизации. Рекомендуется только для доверенных приложений.
4. Client Credentials Flow: Для доступа приложения к своим собственным ресурсам (без пользователя).

Токены:

1. Access Token (Токен доступа): Ключ, который клиент предъявляет серверу ресурсов для доступа к данным. Имеет ограниченный срок жизни.
2. Refresh Token (Токен обновления): Опциональный токен, позволяющий получить новый access token без повторной аутентификации пользователя. Имеет длительный срок жизни.

OpenID Connect (OIDC):

Надстройка над OAuth 2.0 для аутентификации (вход в систему). Добавляет ID Token (JWT - JSON Web Token), который содержит информацию о пользователе.

OAuth 2.0 в .NET:

1. ASP.NET Core Identity: Встроенная поддержка аутентификации через внешних провайдеров (AddAuthentication().AddGoogle().AddFacebook()...).
2. Библиотеки-клиенты: Для ручной реализации можно использовать HttpClient для отправки запросов к API провайдера.
3. JWT (JSON Web Tokens): Стандарт для представления утверждений (claims) между двумя сторонами. Часто используется как формат токенов доступа.

Этапы Authorization Code Flow:

1. Клиент перенаправляет пользователя на сервер авторизации с указанием client_id, redirect_uri, scope.
2. Пользователь аутентифицируется на сервере авторизации и дает разрешение клиенту.
3. Сервер авторизации перенаправляет пользователя обратно на redirect_uri с временным кодом.

4. Клиент отправляет код, `client_id` и `client_secret` на сервер авторизации для получения `access_token` (и `refresh_token`).

5. Клиент использует `access_token` для доступа к API сервера ресурсов.

Задание

Общая постановка задачи:

Разработать веб-приложение (ASP.NET Core MVC или Razor Pages) или десктопное приложение, которое реализует вход через внешнего OAuth2-провайдера (например, Google, GitHub, Yandex). После успешного входа приложение должно отображать информацию о пользователе (имя, email, аватар), полученную через API провайдера.

Базовая структура (для веб-приложения):

1. Создайте новое ASP.NET Core веб-приложение (с аутентификацией).
2. Настройте аутентификацию через выбранного внешнего провайдера.
3. Добавьте кнопку "Войти через [Провайдер]".
4. После входа отобразите на странице профиля информацию, полученную от провайдера.

Технические требования:

1. Client ID и Client Secret не должны храниться в коде. Используйте менеджер секретов (Secret Manager) или переменные окружения.
2. Корректно обрабатывайте ошибки авторизации (пользователь отказал, истек токен и т.д.).
3. Если используется Authorization Code Flow, убедитесь, что `redirect_uri` точно совпадает с зарегистрированным.
4. Для веб-приложений используйте встроенные механизмы ASP.NET Core.

Контрольные вопросы

1. Что такое OAuth 2.0? Для решения каких задач он предназначен?

2. Назовите основные роли в протоколе OAuth 2.0.
3. Чем отличается аутентификация от авторизации? Как OAuth 2.0 связан с этими понятиями?
4. Что такое Authorization Code Flow? Опишите его шаги.
5. Зачем нужен временный код (authorization code), почему нельзя сразу выдать токен?
6. Что такое access token и refresh token? Какова их функция?
7. Что такое OpenID Connect (OIDC) и чем он отличается от "чистого" OAuth 2.0?

Лабораторная работа №22

«Чат на сокетах»

Цель работы: Разработать полноценное клиент-серверное приложение для обмена сообщениями в реальном времени, используя сокеты TCP. Закрепить навыки асинхронного программирования, работы с сетью и многопоточности, а также реализовать базовые функции чата (личные сообщения, комнаты, история).

Теоретическая часть

Ключевые понятия:

Архитектура чата:

1. Централизованная (клиент-сервер): Все клиенты подключаются к одному серверу, который маршрутизирует сообщения. Сервер хранит историю и управляет комнатами.
2. Одноранговая (P2P): Клиенты общаются напрямую. Сложнее в реализации (требуется NAT traversal).

Протокол обмена сообщениями:

Необходимо определить формат сообщений, которыми обмениваются клиент и сервер.

1. Текстовый протокол (JSON): Удобен для отладки и расширения. Каждое сообщение содержит тип (команду) и данные.
2. Бинарный протокол: Компактнее, быстрее, но сложнее в отладке. Применяется в высоконагруженных сценариях, при передаче больших объёмов данных, в микросервисной архитектуре и в условиях ограниченной пропускной способности канала.

Управление подключениями:

Сервер должен хранить список активных клиентов (с их сетевыми потоками, именами, текущей комнатой).

При отключении клиента (обрыв соединения) сервер должен корректно удалить его из списка и оповестить других участников комнаты.

Асинхронная обработка:

Сервер должен использовать асинхронные методы (AcceptTcpClientAsync, ReadAsync, WriteAsync) для обработки множества клиентов без создания отдельного потока на каждый запрос.

Для каждого подключенного клиента обычно запускается асинхронная задача ProcessClientAsync, которая читает входящие сообщения в цикле.

Многопоточность и синхронизация:

Доступ к общим данным (список клиентов, история сообщений, комнаты) должен быть синхронизирован (lock, ConcurrentDictionary).

Дополнительные возможности:

1. Комнаты (channels): Разделение общения по темам.
2. Личные сообщения (private messages): Отправка сообщения конкретному пользователю.
3. История сообщений: Хранение последних N сообщений на сервере и отправка новому участнику.
4. Статус "печатает..." (typing indicator): Уведомление о том, что пользователь набирает сообщение.
5. Поддержка файлов: Передача файлов через отдельный канал или кодирование в base64.

Задание**Общая постановка задачи:**

Разработать приложения сервера и клиента для многопользовательского текстового чата. Сервер должен поддерживать множество одновременных подключений, комнаты общения и личные сообщения. Клиент должен предоставлять удобный интерфейс (консольный или графический) для общения.

Базовая функциональность (обязательная для всех):**Сервер:**

1. Асинхронная обработка клиентов.
2. Хранение списка подключенных пользователей (с именем).

3. Широковещательная рассылка сообщений всем участникам комнаты.
4. Оповещение о подключении/отключении пользователей.
5. Обработка команд: вход (/login имя), выход (/quit), список пользователей (/users).

Клиент:

1. Подключение к серверу по IP и порту.
2. Ввод имени пользователя (логин).
3. Отправка сообщений (просто текст направляется в общую комнату).
4. Прием и отображение входящих сообщений с указанием отправителя и времени.
5. Корректное завершение работы (отправка команды выхода, закрытие сокета).

Требования к реализации:

1. Сервер должен корректно обрабатывать неожиданные отключения клиентов (без падений).
2. Использовать асинхронные методы для всех операций ввода-вывода.
3. Обеспечить потокобезопасность доступа к общим данным на сервере.
4. Реализовать понятный протокол обмена (JSON или текстовые команды).
5. Клиент должен иметь понятный интерфейс (даже консольный – с цветами, разделением экрана).
6. Добавить обработку ошибок (потеря соединения, неверные команды).

Контрольные вопросы

1. Какую архитектуру вы выбрали для чата? Почему?
2. Как сервер обрабатывает множество одновременных подключений? Асинхронность или многопоточность?
3. Как организовано хранение списка подключенных клиентов? Какие проблемы синхронизации возникают?

4. Как сервер узнает, что клиент отключился (некорректно закрыл соединение)?
5. Какие данные содержит сообщение в вашем протоколе (структура JSON)?
6. Как реализована маршрутизация сообщений в комнаты и личные сообщения?
7. Какие меры вы предприняли для защиты от "шустрых" клиентов (flood)?
8. Как реализовано хранение и выдача истории сообщений?
9. Какие сложности возникли при реализации графического клиента (если делали)?

Лабораторная работа №23

«Клиент-серверное приложение»

Цель работы: Разработать законченное клиент-серверное приложение, объединяющее знания и навыки, полученные в течение всего курса. Приложение должно включать графический интерфейс клиента, сервер с бизнес-логикой и базой данных, а также обеспечивать безопасность и многопользовательскую работу.

Теоретическая часть

Ключевые понятия:

Полноценная клиент-серверная архитектура:

1. Тонкий клиент (Thin client): Клиент отвечает только за отображение и ввод, вся логика на сервере.
2. Толстый клиент (Fat client): Клиент содержит часть бизнес-логики (например, валидацию), сервер в основном отвечает за данные.

Компоненты системы:

1. Клиентское приложение: GUI (Windows Forms/WPF), сетевое взаимодействие, локальное кэширование (ЛР №14), локализация (ЛР №17).
2. Серверное приложение: Обработка клиентов, бизнес-логика, доступ к БД (ЛР №11, ЛР №16), аутентификация/авторизация (ЛР №13), логирование (ЛР №8), возможно, кэширование.
3. База данных: Хранение данных (SQL Server, SQLite, PostgreSQL).
4. Сетевой протокол: TCP с собственным протоколом поверх (JSON) или HTTP/REST API (ЛР №5, ЛР №6).

Требования к промышленному приложению:

1. Надежность: Обработка ошибок, восстановление после сбоев, логирование.
2. Безопасность: Аутентификация, авторизация, шифрование (ЛР №18, ЛР №19), защита от SQL-инъекций (ЛР №20).
3. Производительность: Кэширование, асинхронность, пулы соединений.

4. Масштабируемость: Возможность добавления новых серверов (хотя бы теоретическая).
5. Удобство использования: Интуитивный интерфейс, локализация, помощь.

Выбор архитектуры:

1. Монолит: Сервер – одно приложение, обрабатывающее все запросы.
2. Микросервисы: Разделение функциональности на отдельные сервисы (сложно для учебного проекта).

Этапы разработки:

1. Проектирование базы данных.
2. Определение протокола обмена (API).
3. Разработка сервера (слой доступа к данным, бизнес-логика, сетевой слой).
4. Разработка клиента (интерфейс, сетевое взаимодействие, локальное состояние).
5. Интеграция и тестирование.

Задание

Общая постановка задачи:

Разработать комплексное клиент-серверное приложение на свободную тему (или по предложенному списку), которое демонстрирует все ключевые аспекты разработки информационных систем, изученные в курсе. Приложение должно быть законченным, работоспособным и иметь практическую ценность.

Требования к проекту:

1. Клиент: Графический интерфейс (Windows Forms или WPF).
2. Сервер: Отдельное приложение (консольное или Windows-служба).
3. База данных: Реляционная СУБД (SQL Server, SQLite, PostgreSQL) с минимум 3–4 связанными таблицами.
4. Сетевое взаимодействие: TCP (собственный протокол) или HTTP (REST API).
5. Аутентификация и авторизация: Пользователи с ролями (админ, обычный пользователь).

6. Многопользовательскость: Сервер должен поддерживать одновременную работу нескольких клиентов.
7. Локализация: Минимум два языка интерфейса (русский/английский).
8. Безопасность: Хеширование паролей, защита от SQL-инъекций.

Дополнительные требования (выберите минимум 3 варианта из списка):

1. Кэширование данных на клиенте или сервере (ЛР №14).
2. Шифрование сетевого трафика (ЛР №18).
3. Работа с файлами (загрузка/выгрузка, ЛР №9).
4. Логирование действий на сервере (в файл или БД, ЛР №8).
5. Использование паттернов проектирования (Singleton для конфигурации, Factory для БД, ЛР №15, ЛР №16).
6. Возможность смены БД через фабрику (ЛР №16).
7. REST API вместо кастомного TCP-протокола (ЛР №6).
8. OAuth2-аутентификация (через внешнего провайдера, ЛР №21) как альтернатива внутренней.

План выполнения работы (рекомендуемый):

1. Проектирование (1–2 дня): Определить сущности, связи, нарисовать ER-диаграмму. Описать протокол (команды/методы API). Распределить функциональность между клиентом и сервером.
2. Разработка базы данных: Создать скрипты создания таблиц. Заполнить тестовыми данными.
3. Разработка сервера: Реализовать слой доступа к данным, бизнес-логику, сетевой слой. Добавить аутентификацию. Протестировать через простой тестовый клиент (консольный).
4. Разработка клиента: Реализовать GUI, подключить сетевое взаимодействие, добавить локализацию.
5. Интеграция и тестирование: Запустить несколько клиентов, проверить корректность работы, обработку ошибок.
6. Документирование: Написать краткую инструкцию пользователя, описать архитектуру.

Контрольные вопросы (защита проекта)

1. Какова общая архитектура вашего приложения? Из каких компонентов оно состоит?
2. Какая база данных использовалась? Почему выбрали именно ее? Покажите ER-диаграмму.
3. Как реализовано сетевое взаимодействие (протокол, синхронно/асинхронно)?
4. Как обрабатываются множественные подключения на сервере?
5. Как реализована аутентификация и авторизация? Где хранятся пароли?
6. Какие меры безопасности реализованы (шифрование, защита от инъекций)?
7. Как реализована локализация интерфейса?
8. Какие паттерны проектирования были использованы? Где и почему?
9. Какие дополнительные требования (из списка) реализованы?
10. Как тестировалось приложение? Какие инструменты использовались?
11. Какие сложности возникли в процессе разработки и как вы их решили?
12. Что можно улучшить в текущей версии? Какие функции добавить?

Лабораторная работа №24

«Отказоустойчивый сервис»

Цель работы: Изучить принципы построения отказоустойчивых и высокодоступных систем, освоить паттерны обеспечения надежности (retry, circuit breaker, health checks, graceful degradation), и реализовать сервис, способный корректно обрабатывать сбои и продолжать работу.

Теоретическая часть

Ключевые понятия:

Отказоустойчивость (Fault Tolerance):

Определение: Способность системы продолжать корректное функционирование после возникновения сбоев (отказов) в ее компонентах.

Цель: Минимизация времени простоя (downtime) и предотвращение потери данных.

Метрики:

1. MTBF (Mean Time Between Failures): Среднее время наработки на отказ.
2. MTTR (Mean Time To Repair): Среднее время восстановления.
3. Availability (Доступность): Процент времени, когда система работоспособна (например, "пять девяток" – 99.999 % доступности).

Основные стратегии обеспечения отказоустойчивости:

1. Резервирование (Redundancy):
 - Активное (Active-Active): Несколько экземпляров сервиса работают параллельно, нагрузка распределяется между ними.
 - Пассивное (Active-Passive/Hot Standby): Один экземпляр работает, другой (или несколько) находятся в режиме ожидания и принимают нагрузку при отказе основного.
2. Репликация данных: хранение копий данных на разных узлах (master-slave, multi-master).

3. Таймауты (Timeouts): предотвращение бесконечного ожидания ответа от зависшего компонента. Все внешние вызовы должны иметь ограничение по времени.
4. Повторные попытки (Retries) с экспоненциальной задержкой (Exponential Backoff): при временном сбое (например, сетевой проблеме) операция повторяется через увеличивающиеся промежутки времени. Предотвращает перегрузку системы.
5. Изоляция отказов (Bulkhead/Circuit Breaker):
 - Circuit Breaker (Автоматный выключатель): Защищает систему от каскадных отказов. Отслеживает количество ошибок при вызове внешнего сервиса. При превышении порога "размыкает цепь" – все последующие вызовы мгновенно завершаются ошибкой (без реальной попытки), давая сервису время восстановиться. Периодически пробует "замкнуть цепь" снова.
 - Bulkhead (Перегородка): Ограничивает количество одновременных запросов к определенному ресурсу, чтобы проблемы в одной части системы не привели к исчерпанию ресурсов во всей системе.
6. Graceful Degradation (Плавная деградация): при отказе некоторых компонентов система продолжает работать, но с ограниченной функциональностью (например, отключение "тяжелых" функций, показ кэшированных данных вместо реальных).
7. Health Checks (Проверки работоспособности): регулярная проверка состояния компонентов системы (баз данных, внешних API, внутренних сервисов). Эндпоинты/health и /ready в микросервисах.

Задание

Общая постановка задачи:

Разработать отказоустойчивый сервис (веб-сервис или TCP-сервер), который взаимодействует с несколькими внешними зависимостями (база данных, внешнее API, файловая система). Сервис должен корректно обрабатывать сбои этих зависимостей,

продолжать работу (возможно, с ограниченной функциональностью) и предоставлять эндпоинты для мониторинга состояния.

Базовая архитектура:

1. Сервис: ASP.NET Core Web API или TCP-сервер.
2. Зависимости (имитация):
 - База данных (может быть недоступна).
 - Внешнее API (может отвечать медленно или с ошибками).
 - Внутренний компонент (например, кэш или диск).
3. Клиент: Консольное приложение или нагрузочный тест.

Общие требования к реализации:

1. Сервис должен корректно обрабатывать сценарии недоступности зависимостей (без падений и утечек памяти).
2. Реализовать минимум два паттерна отказоустойчивости (retry, circuit breaker, fallback, bulkhead).
3. Предоставить эндпоинт для мониторинга состояния сервиса (health check).
4. Логировать все сбои и переключения режимов.
5. Написать простой клиент/тест, демонстрирующий поведение сервиса при сбоях.

Контрольные вопросы

1. Что такое отказоустойчивость? Чем она отличается от высокой доступности (High Availability)?
2. Какие стратегии обеспечения отказоустойчивости вы знаете? Опишите их.
3. Что такое паттерн Circuit Breaker? Для чего он нужен?
4. Чем отличается активное резервирование (Active-Active) от пассивного (Active-Passive)?
5. Что такое graceful degradation (плавная деградация)? Приведите пример.
6. Какие политики предоставляет библиотека Polly? Как они могут быть скомбинированы?

7. Что такое health checks и как они используются в балансировщиках нагрузки?
8. Как таймауты помогают в построении отказоустойчивых систем?
9. Какую роль играют очереди сообщений в обеспечении надежности?
10. Как можно протестировать отказоустойчивость системы (Chaos Engineering)?

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, 2021. – 896 с. – (Серия «Мастер-класс»).
2. Троелсен, Э. Язык программирования C# 9 и платформа .NET 5 / Э. Троелсен, Ф. Джепикс. – 10-е изд. – Санкт-Петербург : Вильямс, 2021. – 1440 с.
3. Фримен, А. ASP.NET Core MVC с примерами на C# для профессионалов / А. Фримен. – 7-е изд. – Москва : Вильямс, 2022. – 1024 с.
4. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – 2-е изд. – Санкт-Петербург : Питер, 2022. – 448 с. — (Серия «Библиотека программиста»).
5. Фримен, А. Паттерны проектирования для Microsoft .NET / А. Фримен. – Санкт-Петербург : Питер, 2023. – 480 с.
6. Бишоп, Дж. C# в кратких примерах / Дж. Бишоп. – Москва : ДМК Пресс, 2023. – 340 с.
7. Макконнелл, С. Совершенный код. Мастер-класс / С. Макконнелл. – 2-е изд. – Москва : Русская редакция ; Санкт-Петербург : Питер, 2023. – 896 с.
8. Сельман, С. SQL. Полное руководство / С. Сельман. – 3-е изд. – Москва : Диалектика, 2022. – 816 с.

Оглавление

Лабораторная работа №1 «Модуль обработки строк с регулярными выражениями».....	3
Лабораторная работа №2 «Обработчик исключений и структура данных».....	7
Лабораторная работа №3 «Файловый менеджер с сериализацией».	11
Лабораторная работа №4 «Многомодульное консольное приложение».....	16
Лабораторная работа №5 «TCP-сервер для обработки JSON запросов».	22
Лабораторная работа №6 «REST API для управления задачами». ..	27
Лабораторная работа №7 «HTTP-клиент с обработкой ошибок». ...	32
Лабораторная работа №8 «Асинхронный обработчик логов».	36
Лабораторная работа №9 «Файловый обмен через сокеты».	40
Лабораторная работа №10 «Форма ввода с валидацией».	44
Лабораторная работа №11 «Таблица с данными из базы данных»..	49
Лабораторная работа №12 «Интеграция GUI с бизнес логикой». ...	53
Лабораторная работа №13 «Система авторизации».	57
Лабораторная работа №14 «Кеширующий клиент».	61
Лабораторная работа №15 «Шаблон Singleton для конфигурации».	65
Лабораторная работа №16 «Фабрика подключений к базам данных».	68
Лабораторная работа №17 «Модуль локализации».	73
Лабораторная работа №18 «Шифрование данных».....	77
Лабораторная работа №19 «Цифровая подпись и HTTPS».	81
Лабораторная работа №20 «Защита от SQL-инъекций».	85
Лабораторная работа №21 «OAuth2-клиент».	88

Лабораторная работа №22 «Чат на сокетах».....	92
Лабораторная работа №23 «Клиент-серверное приложение».....	96
Лабораторная работа №24 «Отказоустойчивый сервис».....	100
Список литературы.....	104
Оглавление.....	105