МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Пятигорский институт (филиал) СКФУ
Колледж Пятигорского института (филиал) СКФУ

И.О. Директора
Пятигорского института
(филиал) СКФУ
В.А. Фурсов

ПМ.02 РАЗРАБОТКА И ИНТЕГРАЦИЯ МОДУЛЕЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ
МДК.02.02 Осуществление интеграции программных модулей

Специальности СПО
09.02.11 Разработка и управление программным обеспечением

Пятигорск 2026 г.

Методические указания к выполнению лабораторных работ по дисциплине ПМ.02 Разработка и интеграция модулей программного обеспечения составлены в соответствии с требованиями ФГОС СПО. Предназначены для студентов, обучающихся по специальности 09.02.11 Разработка и управление программным обеспечением.

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Дисциплина «Осуществление интеграции программных модулей» направлена на формирование у студентов компетенций в области проектирования, реализации и сопровождения интеграционных решений, обеспечивающих взаимодействие программных модулей и компонентов в распределённых системах.

В результате освоения учебной дисциплины обучающийся должен:

знать:

понятие, цели, задачи и виды интеграции программных модулей;

основные проблемы интеграции (гетерогенность, масштабируемость, безопасность, управление версиями);

архитектурные стили интеграции (клиент-серверная, многоуровневая, SOA, микросервисная);

шаблоны интеграции корпоративных приложений (EIP);

протоколы передачи данных и форматы обмена (HTTP/HTTPS, JMS, AMQP, MQTT, JSON, XML, Protobuf);

принципы синхронного и асинхронного взаимодействия;

принципы REST API, методы HTTP, коды состояния, документирование OpenAPI;

структуру SOAP-сообщений, WSDL, UDDI;

принципы работы брокеров сообщений (RabbitMQ, Apache Kafka);

методы обеспечения безопасности при интеграции (OAuth2, JWT, API-ключи, TLS/SSL, Rate limiting, CORS);

принципы контейнеризации (Docker) и оркестрации (Kubernetes);

основы облачных интеграций и Serverless;

стратегии управления версиями API и семантическое версионирование;

инструменты мониторинга и отладки распределённых систем (ELK, Jaeger, Prometheus, Grafana).

уметь:

анализировать архитектуру приложения и определять точки интеграции;

проектировать интеграционное взаимодействие с использованием диаграмм;

реализовывать синхронный и асинхронный обмен данными между модулями;

разрабатывать REST API и документировать его с помощью OpenAPI;

создавать и развертывать SOAP-сервисы;

настраивать брокеры сообщений (RabbitMQ, Kafka);

применять методы аутентификации и авторизации в интеграционных решениях;

контейнеризовать приложения с помощью Docker и запускать многоконтейренные приложения;

развертывать функции в Serverless-средах;

настраивать пайплайны CI/CD для интеграционных проектов;

настраивать сбор логов, метрик и трассировку запросов.

В результате освоения учебной дисциплины студент должен овладевать:

Общими компетенциями:

ОК 01. Выбирать способы решения задач профессиональной деятельности применительно к различным контекстам

ОК 02. Использовать современные средства поиска, анализа и интерпретации информации, и информационные технологии для выполнения задач профессиональной деятельности

ОК 03. Планировать и реализовывать собственное профессиональное и личностное развитие, предпринимательскую деятельность в профессиональной сфере, использовать знания по правовой и финансовой грамотности в различных жизненных ситуациях

ОК 04. Эффективно взаимодействовать и работать в коллективе и команде

ОК 05. Осуществлять устную и письменную коммуникацию на государственном языке Российской Федерации с учетом особенностей социального и культурного контекста

ОК 06. Проявлять гражданско-патриотическую позицию, демонстрировать осознанное поведение на основе традиционных российских духовно-нравственных ценностей, в том числе с учетом гармонизации межнациональных и межрелигиозных отношений, применять стандарты антикоррупционного поведения

ОК 07. Содействовать сохранению окружающей среды, ресурсосбережению, применять знания об изменении климата, принципы бережливого производства, эффективно действовать в чрезвычайных ситуациях

ОК 08. Использовать средства физической культуры для сохранения и укрепления здоровья в процессе профессиональной деятельности и поддержания необходимого уровня физической подготовленности

ОК 09. Пользоваться профессиональной документацией на государственном и иностранном языках.

Профессиональными компетенциями:

ПК 2.1. Проектировать модули программного обеспечения

ПК 2.2. Разрабатывать модули программного обеспечения

ПК 2.3. Выполнять интеграцию модулей и компонентов программного обеспечения

ПК 2.4. Выполнять тестирование и отладку программного обеспечения

ПК 2.5. Осуществлять документирование программных модулей программного обеспечения

ОБЩИЕ МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

Варианты лабораторных работ по дисциплине «Безопасность программного обеспечения» для каждого обучающегося – индивидуальные.

Задачи и ответы на вопросы, выполненные не по своему варианту, не засчитываются.

Лабораторная работа выполняется в отдельной тетради. Условия задачи и формулировки вопросов переписываются полностью. Формулы, расчеты, ответы на вопросы пишутся ручкой, а чертежи, схемы и рисунки выполняются карандашом, на графиках и диаграммах указывается масштаб. Вначале задача решается в общем виде, затем делаются расчёты по условию задания. Решение задач обязательно ведется в Международной системе единиц (СИ).

При выполнении лабораторной работы необходимо следовать методическим указаниям: повторить краткое содержание теории, запомнить основные формулы и законы, проанализировать пример выполнения аналогичного задания, затем преступить непосредственно к решению задачи. К зачету допускаются студенты, получившие положительные оценки по всем лабораторным работам.

Правила выполнения лабораторных работ.

Студент должен прийти на лабораторную работу подготовленным к выполнению лабораторной работы.

Каждый студент после проведения работы должен представить отчет о проделанной работе с анализом полученных результатов и выводом по работе.

Таблицы и рисунки следует выполнять с помощью чертежных инструментов (линейки, циркуля, и т.д.) карандашом с соблюдением ЕСКД.

Расчет следует проводить с точностью до двух значащих цифр.

Исправления проводить на обратной стороне листа. При мелких исправлениях неправильное слово (буква, число и т.п.) аккуратно зачеркивается и над ним пишут правильное пропущенное слово (букву, число и т.п.).

Вспомогательные расчеты можно выполнять на отдельных листах, а при необходимости на листах отчета.

Если студент не выполнит лабораторную работу или часть работы, то он выполнит ее во внеурочное время, согласованное с преподавателем.

Оценку по лабораторной работе студент получает с учетом срока выполнения работы, если;

- расчеты выполнены правильно и в полном объеме;

сделан анализ проделанной работы и вывод по результатам работы;

студент может пояснить выполнение любого этапа работы;

отчет выполнен в соответствии с требованиями к выполнению работы.

Лабораторная работа №1
Анализ архитектуры существующего приложения и определение точек интеграции

Цель: освоить методику анализа архитектуры программного обеспечения и выявления точек интеграции между модулями.

Краткие теоретические сведения

Интеграция программных модулей — процесс объединения отдельных модулей в единую работающую систему с обеспечением их взаимодействия.

Виды интеграции:

Горизонтальная — объединение модулей одного уровня иерархии (например, между двумя серверами приложений).

Вертикальная — объединение модулей разных уровней (например, приложение и база данных).

«Точка-точка» (Point-to-Point) — прямая связь между двумя модулями без посредников.

Шина данных (Data Bus) — центральный компонент, через который обмениваются данными все модули.

Основные проблемы интеграции:

Гетерогенность — модули на разных языках и платформах.

Масштабируемость — способность системы расти при увеличении нагрузки.

Безопасность — защита данных при передаче и хранении.

Управление версиями — согласование изменений в модулях.

Задание:

Получить от преподавателя описание существующего приложения (или выбрать открытое приложение: интернет-магазин, банковская система, система бронирования).

Проанализировать архитектуру приложения:

выделить основные модули и компоненты;

определить их назначение и ответственность;

выявить существующие связи между модулями.

Определить точки интеграции:

какие модули обмениваются данными;

какие протоколы и форматы используются;

синхронное или асинхронное взаимодействие.

Выявить проблемные места интеграции:

высокую связанность;

узкие места производительности;

риски безопасности.

Предложить пути улучшения интеграции.

Составить отчет, содержащий:

диаграмму компонентов (или блок-схему архитектуры);

таблицу точек интеграции;

список выявленных проблем;

рекомендации по улучшению.

Контрольные вопросы:

Что такое интеграция программных модулей? Какие цели она преследует?

Чем горизонтальная интеграция отличается от вертикальной?

Каковы преимущества и недостатки интеграции «точка-точка»?

Что такое шина данных? Какие проблемы она решает?

Какие основные проблемы возникают при интеграции разнородных систем?

Лабораторная работа №2

Проектирование интеграционного взаимодействия для заданного сценария (диаграммы последовательности, компонентов)

Цель: освоить проектирование интеграционного взаимодействия с использованием UML-диаграмм.

Краткие теоретические сведения

Диаграмма компонентов — показывает структуру системы, разбиение на компоненты и их зависимости.

Диаграмма последовательности — показывает временную последовательность обмена сообщениями между объектами.

Клиент-серверная архитектура — разделение на клиента (инициирует запросы) и сервер (обрабатывает запросы).

Многоуровневая архитектура — разделение на логические уровни: представление (UI), бизнес-логика (BLL), доступ к данным (DAL).

SOA (Service-Oriented Architecture) — сервис-ориентированная архитектура, где функциональность представлена в виде сервисов.

Микросервисная архитектура — набор небольших, независимо развертываемых сервисов.

Задание:

Получить заданный сценарий интеграции (например: «Пользователь оформляет заказ в интернет-магазине», «Клиент переводит деньги между счетами», «Система бронирования билетов»).

Разработать диаграмму компонентов, отображающую:

все участвующие модули/сервисы;

интерфейсы взаимодействия;

зависимости между компонентами.

Разработать диаграмму последовательности для основного сценария:

участники (клиент, сервисы, базы данных);

сообщения между участниками;

временная последовательность.

Определить тип взаимодействия (синхронное/асинхронное) для каждого обмена.

Обосновать выбор архитектурного стиля (клиент-сервер, SOA, микросервисы).

Составить отчет с диаграммами и пояснениями.

Контрольные вопросы:

Что показывает диаграмма компонентов?

Что показывает диаграмма последовательности?

В чем разница между синхронным и асинхронным вызовом?

Какие преимущества дает многоуровневая архитектура?

Чем SOA отличается от микросервисной архитектуры?

Лабораторная работа №3

Реализация шаблона «канал сообщений» на основе очередей

Цель: освоить реализацию шаблона интеграции «канал сообщений» с использованием очередей.

Краткие теоретические сведения

EIP (Enterprise Integration Patterns) — шаблоны интеграции корпоративных приложений.

Канал сообщений (Message Channel) — обеспечивает передачу сообщений между компонентами. Компоненты подключаются к каналу для отправки и получения сообщений.

Типы каналов:

Точка-точка (Point-to-Point) — сообщение получает только один потребитель.

Издатель-подписчик (Publish-Subscribe) — сообщение получают все подписчики.

Очередь (Queue) — канал типа точка-точка.

Топик (Topic) — канал типа издатель-подписчик.

Задание:

Установить и запустить брокер сообщений (RabbitMQ или ActiveMQ).

Реализовать программу-отправитель (producer), которая:

подключается к брокеру;

создает очередь (queue);

отправляет 10 сообщений с текущим временем и порядковым номером.

Реализовать программу-получатель (consumer), которая:

подключается к брокеру;

читает сообщения из той же очереди;

выводит их на экран.

Запустить несколько экземпляров получателя и убедиться, что сообщения распределяются между ними (каждое сообщение обрабатывается только одним получателем).

Модифицировать схему для использования топика (topic) — все получатели получают все сообщения.

Составить отчет с описанием реализации, листингами программ и результатами работы.

Контрольные вопросы:

Что такое шаблон «канал сообщений»?

Чем отличается очередь от топика?

Какие преимущества дает использование очередей сообщений?

Какой брокер сообщений был использован и почему?

Что произойдет, если получатель отключится во время отправки сообщений?

Лабораторная работа №4

Разработка клиент-серверного приложения с обменом JSON-сообщениями

Цель: освоить разработку клиент-серверного приложения с обменом данными в формате JSON.

Краткие теоретические сведения

JSON (JavaScript Object Notation) — текстовый формат обмена данными, основанный на синтаксисе JavaScript. Человекочитаемый, легко парсится.

Структура JSON:

объект: { "ключ": "значение" }

массив: ["элемент1", "элемент2"]

типы данных: строки, числа, булевы, null, объекты, массивы.

HTTP (HyperText Transfer Protocol) — протокол прикладного уровня для передачи данных.

Методы HTTP: GET, POST, PUT, DELETE, PATCH.

Задание:

Выбрать язык программирования и фреймворк для создания HTTP-сервера (Flask/Django для Python, Express для Node.js, Spring Boot для Java).

Разработать сервер, обрабатывающий следующие эндпоинты:

POST /echo — принимает JSON, возвращает тот же JSON с добавленным полем "timestamp";

POST /calculate — принимает JSON {"a": число, "b": число, "operation": "add/sub/mul/div"}, возвращает результат.

Разработать клиента (консольного или с GUI), который:

запрашивает у пользователя данные;

формирует JSON-запрос;

отправляет на сервер;

выводит ответ.

Добавить обработку ошибок (неверный JSON, деление на ноль, неизвестная операция).

Составить отчет с листингами сервера и клиента, примерами запросов и ответов.

Контрольные вопросы:

Что такое JSON? Какие типы данных поддерживает?

Чем JSON отличается от XML?

Как выполняется сериализация и десериализация JSON в выбранном языке?

Как сервер обрабатывает входящий JSON?

Как клиент формирует JSON-запрос?

Лабораторная работа №5

Реализация асинхронного взаимодействия с использованием очередей сообщений

Цель: освоить реализацию асинхронного взаимодействия между компонентами с использованием брокера сообщений.

Краткие теоретические сведения

Синхронное взаимодействие — отправитель блокируется и ожидает ответа от получателя. Пример: HTTP-запрос.

Асинхронное взаимодействие — отправитель не блокируется, продолжая свою работу. Ответ (если нужен) приходит позже через другой канал.

Очереди сообщений — основа асинхронного взаимодействия. Позволяют развязывать отправителя и получателя во времени и пространстве.

Плюсы асинхронности: отказоустойчивость, масштабируемость, разгрузка компонентов.

Задание:

Разработать систему обработки заказов:

API Gateway (HTTP-сервер) — принимает запросы на создание заказа, публикует событие в очередь orders.

Обработчик заказов (consumer) — читает из очереди orders, имитирует обработку (задержка 1-5 секунд), публикует результат в очередь results.

Уведомлятор (второй consumer) — читает из очереди results, выводит результат на экран.

Реализовать все компоненты как отдельные процессы/потоки.

Продемонстрировать:

что API Gateway не ждет завершения обработки;

что несколько заказов обрабатываются параллельно (если запустить несколько обработчиков);

что при отключении обработчика заказы накапливаются в очереди.

Составить отчет с архитектурной схемой, листингами и демонстрацией.

Контрольные вопросы:

Чем синхронное взаимодействие отличается от асинхронного?

В каких случаях асинхронное взаимодействие предпочтительнее?

Как очереди сообщений обеспечивают отказоустойчивость?

Что происходит с сообщениями, если потребитель временно недоступен?

Как обеспечить гарантированную доставку сообщений?

Лабораторная работа №6

Сравнение производительности синхронного и асинхронного обмена

Цель: провести экспериментальное сравнение производительности синхронного и асинхронного способов взаимодействия.

Краткие теоретические сведения

Ключевые метрики производительности:

Latency (задержка) — время от отправки запроса до получения ответа.

Throughput (пропускная способность) — количество запросов/сообщений в единицу времени.

Scalability (масштабируемость) — способность системы сохранять производительность при росте нагрузки.

Задание:

Реализовать два варианта одной и той же системы (например, генерация отчетов):

Синхронный вариант: клиент → сервер (ожидание) → ответ.

Асинхронный вариант: клиент → очередь → обработчик → очередь → клиент (с ожиданием результата через отдельный эндпоинт или WebSocket).

Провести нагрузочное тестирование для разных сценариев:

легкие операции (1-10 мс);

тяжелые операции (100-1000 мс);

разное количество параллельных запросов (1, 10, 50, 100).

Замерить:

среднее время ответа;

максимальное время ответа;

пропускную способность (запросов/сек).

Построить графики зависимости времени ответа от нагрузки.

Сделать выводы о том, в каких сценариях какой подход эффективнее.

Составить отчет с методикой тестирования, результатами, графиками и выводами.

Контрольные вопросы:

Какие метрики используются для оценки производительности?

Почему асинхронный подход может давать большую задержку для отдельного запроса?

В чем преимущество асинхронного подхода при высокой нагрузке?

Как влияет время обработки на выбор между синхронностью и асинхронностью?

Что такое деградация производительности и как её обнаружить?

Лабораторная работа №7

Работа с протоколом MQTT (брокер Mosquitto, publisher/subscriber)

Цель: освоить протокол MQTT для обмена сообщениями в сценариях IoT и микросервисов.

Краткие теоретические сведения

MQTT (Message Queuing Telemetry Transport) — легковесный протокол обмена сообщениями, основанный на модели издатель-подписчик. Оптимизирован для устройств с ограниченными ресурсами и нестабильными сетями.

Основные понятия MQTT:

Брокер — центральный сервер (Mosquitto, EMQX).

Издатель (Publisher) — отправляет сообщения в топик.

Подписчик (Subscriber) — получает сообщения из топика.

Топик (Topic) — именованный канал (например, sensors/temperature/room1).

Уровни качества обслуживания (QoS):

QoS 0 — не более одного раза (fire-and-forget).

QoS 1 — не менее одного раза (с подтверждением).

QoS 2 — ровно один раз.

Задание:

Установить и запустить брокер Mosquitto (локально или в Docker).

Реализовать издателя (publisher) на Python/Java, который:

подключается к брокеру;

публикует в топик sensors/temperature случайное значение температуры каждую секунду;

публикует в топик sensors/humidity случайное значение влажности каждые 5 секунд.

Реализовать подписчика (subscriber), который:

подписывается на топик sensors/# (все подтопики);

выводит полученные сообщения с временной меткой.

Реализовать второго подписчика, который подписывается только на sensors/temperature и вычисляет среднюю температуру за последние 10 измерений.

Продемонстрировать работу системы.

Составить отчет с описанием протокола MQTT, листингами программ и результатами.

Контрольные вопросы:

Для каких сценариев предназначен протокол MQTT?

Чем MQTT отличается от AMQP?

Что такое QoS в MQTT? Какие уровни существуют?

Как работает механизм «последняя воля» (Last Will) в MQTT?

Как брокер MQTT обрабатывает отключение подписчика?

Лабораторная работа №8

Создание простого REST-сервиса на выбранном фреймворке

Цель: освоить разработку REST API на одном из современных фреймворков.

Краткие теоретические сведения

Принципы REST (Representational State Transfer):

клиент-серверная архитектура;

отсутствие состояния (stateless) — каждый запрос содержит всю необходимую информацию;

кэширование;

единообразие интерфейса;

слои.

Ресурс — любой объект, к которому можно обратиться через API (пользователь, товар, заказ). Идентифицируется URL.

HTTP-методы:

GET — получение ресурса (без побочных эффектов);

POST — создание нового ресурса;

PUT — полная замена ресурса;

PATCH — частичное обновление;

DELETE — удаление.

Коды состояния:

2xx — успех;

3xx — перенаправление;

4xx — ошибка клиента;

5xx — ошибка сервера.

Задание:

Выбрать фреймворк: Flask/Django REST (Python), Express (Node.js), Spring Boot (Java), FastAPI (Python).

Создать REST-сервис для управления библиотекой книг:

GET /books — список книг (поддержка фильтрации по автору и году);

GET /books/{id} — книга по ID;

POST /books — добавление книги;

PUT /books/{id} — полное обновление;

PATCH /books/{id} — частичное обновление (только статус «в наличии/выдана»);

DELETE /books/{id} — удаление.

Данные хранить в памяти (список словарей) или в SQLite.

Добавить валидацию входных данных.

Добавить обработку ошибок (404 — книга не найдена, 400 — неверные данные).

Составить отчет с описанием API (эндпоинты, методы, параметры, примеры).

Контрольные вопросы:

Какие принципы лежат в основе REST?

Что такое ресурс в REST? Как он идентифицируется?

В чем разница между PUT и PATCH?

Какой код состояния возвращается при успешном создании ресурса?

Что такое stateless и почему это важно для REST?

Лабораторная работа №9

Разработка клиента для тестирования API с использованием Postman или cURL

Цель: освоить методы тестирования REST API с помощью специализированных инструментов.

Краткие теоретические сведения

Postman — популярный инструмент для тестирования, документирования и мониторинга API.

Возможности Postman:

отправка HTTP-запросов различных типов;

управление переменными (окружения);

создание коллекций запросов;

автоматическое тестирование (скрипты);

генерация документации.

cURL — командная строка для отправки HTTP-запросов.

Пример cURL:

bash

curl -X GET "http://localhost:8080/books"

curl -X POST "http://localhost:8080/books" -H "Content-Type: application/json" -d '{"title":"Война и мир"}'

Задание:

Использовать REST-сервис из предыдущей лабораторной работы.

Установить Postman (или использовать cURL).

Создать коллекцию запросов, покрывающую все эндпоинты сервиса:

получение списка книг (с фильтрацией);

получение книги по ID;

создание книги;

обновление книги (PUT и PATCH);

удаление книги.

Настроить переменные окружения (базовый URL).

Написать простые тесты (в Postman — вкладка Tests) для проверки:

кодов состояния;

структуры ответа;

наличия обязательных полей.

Сохранить коллекцию и экспортировать её в файл.

Составить отчет со скриншотами запросов/ответов и листингом тестов.

Контрольные вопросы:

Для чего используется Postman при разработке API?

Что такое коллекция в Postman?

Как в Postman настроить переменные окружения?

Как проверить код состояния ответа в Postman Tests?

Какие команды cURL соответствуют GET, POST, PUT, DELETE?

Лабораторная работа №10

Генерация и анализ документации OpenAPI для разработанного API

Цель: освоить документирование REST API с использованием спецификации OpenAPI и инструмента Swagger.

Краткие теоретические сведения

OpenAPI (ранее Swagger) — спецификация для описания REST API в формате JSON или YAML.

Что описывает OpenAPI:

эндпоинты и HTTP-методы;

параметры запроса (path, query, header);

тело запроса (схема данных);

ответы (коды состояния, схемы данных);

аутентификацию;

серверы.

Инструменты OpenAPI:

Swagger UI — интерактивная документация с возможностью выполнять запросы.

Swagger Editor — редактор спецификаций.

Swagger Codegen — генерация клиентов и серверов по спецификации.

Задание:

Использовать REST-сервис из предыдущих работ.

Создать OpenAPI-спецификацию (файл openapi.yaml или openapi.json) для разработанного API, включая:

информацию о сервере;

все эндпоинты с методами;

параметры запросов;

схемы данных (Book, Error);

коды состояния ответов.

Использовать Swagger Editor для проверки корректности спецификации.

Сгенерировать HTML-документацию с помощью Swagger UI.

Протестировать API через Swagger UI (выполнить запросы из документации).

Составить отчет со спецификацией OpenAPI и скриншотами Swagger UI.

Контрольные вопросы:

Что такое OpenAPI (Swagger)?

Какие элементы описываются в OpenAPI-спецификации?

Какую информацию содержит схема данных (schema)?

Чем полезен Swagger UI?

Как можно сгенерировать клиентский код по OpenAPI-спецификации?

Лабораторная работа №11

Создание и развертывание SOAP-сервиса, генерация WSDL

Цель: освоить создание SOAP-веб-сервиса и генерацию WSDL-документации.

Краткие теоретические сведения

SOAP (Simple Object Access Protocol) — протокол обмена структурированными сообщениями на основе XML.

Структура SOAP-сообщения:

Envelope (конверт) — корневой элемент;

Header (заголовок) — необязательный, содержит метаинформацию;

Body (тело) — содержит сам вызов или ответ;

Fault — информация об ошибке.

WSDL (Web Services Description Language) — XML-документ, описывающий:

доступные операции (методы);

входные и выходные параметры;

конечные точки (endpoint);

привязку к протоколу (SOAP/HTTP).

UDDI (Universal Description, Discovery and Integration) — реестр для публикации и поиска веб-сервисов.

Задание:

Выбрать технологию для создания SOAP-сервиса:

Python: zeep, spyne, soaplib;

Java: JAX-WS;

C#: WCF, ASP.NET Core SOAP;

PHP: SoapServer.

Создать SOAP-сервис «Калькулятор» с операциями:

add(a, b) — сложение;

subtract(a, b) — вычитание;

multiply(a, b) — умножение;

divide(a, b) — деление.

Сгенерировать WSDL-документ автоматически (через фреймворк).

Изучить структуру WSDL:

<types> — типы данных;

<message> — сообщения;

<portType> — операции;

<binding> — привязка к SOAP;

<service> — конечная точка.

Развернуть сервис локально.

Составить отчет с листингом сервиса, WSDL-файлом и описанием структуры.

Контрольные вопросы:

Что такое SOAP? Какова структура SOAP-сообщения?

Что такое WSDL? Какую информацию он содержит?

Что такое UDDI и для чего он используется?

Чем SOAP отличается от REST?

В каких сценариях SOAP предпочтительнее REST?

Лабораторная работа №12
Разработка клиента для SOAP-сервиса

Цель: освоить разработку клиентского приложения для взаимодействия с SOAP-сервисом.

Краткие теоретические сведения

SOAP-клиент — программа, формирующая SOAP-запросы, отправляющая их по HTTP и обрабатывающая SOAP-ответы.

Способы создания SOAP-клиента:

использование сгенерированных классов по WSDL (JAX-WS, wsimport);

ручное формирование SOAP-XML с помощью библиотек (zeep для Python, SoapClient для PHP);

низкоуровневая отправка HTTP с XML-телом.

Задание:

Использовать SOAP-сервис из предыдущей лабораторной работы.

Разработать SOAP-клиента на том же языке или другом, который:

подключается к WSDL (если доступен) или формирует запросы вручную;

вызывает все операции сервиса (add, subtract, multiply, divide);

выводит результаты на экран.

Реализовать обработку ошибок:

сервер недоступен;

деление на ноль (должен вернуть SOAP Fault).

Продемонстрировать работу клиента с выводом:

отправляемого SOAP-запроса;

полученного SOAP-ответа.

Составить отчет с листингом клиента, примерами запросов и ответов.

Контрольные вопросы:

Как SOAP-клиент узнаёт структуру запросов и ответов?

Что такое SOAP Fault и как его обработать?

Чем отличается использование сгенерированных классов от ручного формирования XML?

Какой протокол транспорта используется для SOAP?

Как в SOAP передаются сложные типы данных (массивы, структуры)?

Лабораторная работа №13

Установка и настройка RabbitMQ. Реализация producer-consumer

Цель: освоить установку и настройку брокера сообщений RabbitMQ, реализовать базовую схему producer-consumer.

Краткие теоретические сведения

RabbitMQ — брокер сообщений, реализующий протокол AMQP (Advanced Message Queuing Protocol).

Основные понятия RabbitMQ:

Producer — отправляет сообщения.

Consumer — получает сообщения.

Queue — очередь (буфер) для сообщений.

Exchange — распределяет сообщения по очередям.

Binding — связь между exchange и очередью.

Типы exchange:

direct — маршрутизация по точному совпадению ключа;

topic — маршрутизация по шаблону;

fanout — отправка во все связанные очереди;

headers — маршрутизация по заголовкам.

Задание:

Установить RabbitMQ (локально или в Docker).

Включить веб-интерфейс управления (порт 15672).

Реализовать producer на Python/Java, который:

подключается к RabbitMQ;

создает очередь task_queue;

отправляет 10 сообщений с текстом "Task #N".

Реализовать consumer, который:

подключается к той же очереди;

получает сообщения и имитирует обработку (time.sleep).

Запустить несколько consumer'ов и убедиться, что сообщения распределяются между ними (round-robin).

Изучить веб-интерфейс: посмотреть очереди, сообщения, каналы.

Составить отчет с листингами, скриншотами веб-интерфейса и результатами.

Контрольные вопросы:

Какой протокол использует RabbitMQ?

Что такое exchange и для чего он нужен?

Как RabbitMQ распределяет сообщения между несколькими consumer'ами?

Что происходит с сообщением, если consumer не подтвердил его обработку (ack)?

Как посмотреть состояние очередей в RabbitMQ?

Лабораторная работа №14

Работа с Kafka: создание продюсера и консюмера, настройка групп

Цель: освоить работу с Apache Kafka, создание продюсеров и консюмеров, использование consumer groups.

Краткие теоретические сведения

Apache Kafka — распределенная платформа для потоковой передачи данных (streaming platform).

Основные понятия Kafka:

Producer — публикует сообщения в топик.

Consumer — читает сообщения из топика.

Consumer Group — группа консюмеров, совместно читающих топик (каждое сообщение доставляется одному консюмеру в группе).

Topic — категория/канал сообщений.

Partition — разбиение топика для параллелизма.

Broker — сервер Kafka.

Ключевые отличия Kafka от RabbitMQ:

Kafka хранит сообщения на диске дольше (не удаляет сразу после прочтения);

Kafka поддерживает перечитывание сообщений заново;

Kafka ориентирована на высокую пропускную способность.

Задание:

Установить Kafka (локально или в Docker) и Zookeeper.

Создать топик test-topic с 3 разделами (partitions).

Реализовать продюсера, отправляющего 100 сообщений (ключ — случайный ID, значение — текст).

Реализовать консюмера, который:

подключается к топику в составе consumer group group1;

читает сообщения и выводит их.

Запустить несколько экземпляров консюмера в одной группе — убедиться, что сообщения распределяются между ними.

Запустить второй консюмер в другой группе — убедиться, что он получает все сообщения (как и первая группа).

Остановить консюмеров, затем запустить заново — убедиться, что можно перечитать сообщения с начала (или с определенного смещения).

Составить отчет с листингами, описанием настроек и результатами.

Контрольные вопросы:

Для каких сценариев Kafka более подходит, чем RabbitMQ?

Что такое consumer group и как она работает?

Что такое partition и как они влияют на параллелизм?

Как Kafka обеспечивает сохранность сообщений при сбоях?

Что такое offset и как его можно изменить?

Лабораторная работа №15

Добавление JWT-аутентификации в REST API

Цель: освоить добавление механизма аутентификации и авторизации в REST API с использованием JWT.

Краткие теоретические сведения

JWT (JSON Web Token) — компактный и самодостаточный способ передачи информации между сторонами в виде JSON-объекта.

Структура JWT: header.payload.signature

Header — алгоритм подписи (HS256, RS256).

Payload — данные (claims): sub, name, iat, exp, roles.

Signature — подпись, подтверждающая целостность.

Поток аутентификации с JWT:

Пользователь отправляет логин/пароль на /login.

Сервер проверяет учетные данные, генерирует JWT и возвращает его.

Клиент сохраняет JWT (в localStorage, cookie или памяти).

При каждом запросе к защищенному API клиент отправляет JWT в заголовке: Authorization: Bearer <token>.

Сервер проверяет подпись и срок действия токена, затем выполняет запрос.

Задание:

Использовать REST-сервис из лабораторной работы №8 (библиотека).

Добавить эндпоинт /auth/login, который принимает { "username": "admin", "password": "secret" } и возвращает JWT.

Добавить middleware/декоратор для проверки JWT на защищенных эндпоинтах:

создание, обновление, удаление книг требуют аутентификации;

GET-запросы (чтение) доступны без аутентификации.

Реализовать проверку ролей (например, только пользователь с ролью admin может удалять книги).

Продемонстрировать:

запрос без токена → 401 Unauthorized;

запрос с токеном → успех;

запрос с истекшим токеном → 401.

Составить отчет с листингами, примерами запросов и ответов.

Контрольные вопросы:

Из каких частей состоит JWT?

Чем JWT отличается от сессионных cookies?

Что такое refresh token и зачем он нужен?

Как сервер проверяет JWT без обращения к базе данных?

Какие угрозы существуют при использовании JWT и как их смягчить?

Лабораторная работа №16

Настройка HTTPS для веб-сервиса, использование самоподписанных сертификатов

Цель: освоить настройку HTTPS для веб-сервиса с использованием самоподписанных сертификатов.

Краткие теоретические сведения

TLS/SSL (Transport Layer Security / Secure Sockets Layer) — криптографические протоколы, обеспечивающие защищенную передачу данных в сети.

HTTPS — HTTP поверх TLS/SSL. Шифрует все данные между клиентом и сервером.

Самоподписанный сертификат — сертификат, подписанный самим создателем, а не доверенным центром сертификации (CA). Подходит для тестирования и внутренних сетей.

Шаги настройки HTTPS:

Генерация закрытого ключа и сертификата (openssl).

Настройка веб-сервера/фреймворка на использование сертификата.

Запуск сервера на порту 443 (или другом).

Задание:

Использовать любой веб-сервер/фреймворк (Express, Flask, Spring Boot).

Сгенерировать самоподписанный сертификат с помощью OpenSSL:

bash

openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365 -nodes

Настроить сервер на использование HTTPS с полученными файлами (cert.pem, key.pem).

Запустить сервер на порту 8443.

Проверить работу через браузер (предупредит о небезопасности — это нормально для самоподписанного).

Настроить клиент (Postman или cURL) для игнорирования проверки сертификата (или добавить сертификат в доверенные).

Продемонстрировать, что HTTP-запросы (порт 8080) не зашифрованы, а HTTPS-запросы — зашифрованы (можно перехватить трафик в Wireshark для сравнения).

Составить отчет с командами генерации сертификата, настройками сервера и проверкой.

Контрольные вопросы:

Чем TLS отличается от SSL?

Зачем нужен HTTPS, если данные уже шифруются на уровне приложения?

Что такое самоподписанный сертификат и где его можно использовать?

Какую информацию содержит сертификат?

Как работает механизм доверия к сертификатам (CA)?

Лабораторная работа №17
Контейнеризация ранее разработанного REST-сервиса, создание Docker-образа

Цель: освоить контейнеризацию приложения с помощью Docker.

Краткие теоретические сведения

Docker — платформа для контейнеризации приложений.

Основные понятия:

Dockerfile — текстовый файл с инструкциями для сборки образа.

Образ (image) — шаблон для создания контейнеров.

Контейнер (container) — запущенный экземпляр образа.

Реестр (registry) — хранилище образов (Docker Hub).

Лучшие практики Docker:

использовать минимальный базовый образ (alpine, slim);

не запускать контейнеры от root;

использовать многоступенчатую сборку (multi-stage);

минимизировать количество слоев.

Задание:

Взять ранее разработанный REST-сервис (например, из ЛР №8).

Создать Dockerfile:

выбрать базовый образ (python:3.11-slim, node:18-alpine, openjdk:17-slim);

установить рабочую директорию;

скопировать файлы зависимостей (requirements.txt, package.json);

установить зависимости;

скопировать исходный код;

создать непривилегированного пользователя;

указать команду запуска.

Собрать образ: docker build -t my-rest-service .

Запустить контейнер: docker run -p 8080:8080 my-rest-service

Проверить работу API через браузер или Postman.

Изучить список образов и контейнеров (docker images, docker ps).

Составить отчет с содержимым Dockerfile, командами сборки/запуска и проверкой работы.

Контрольные вопросы:

Что такое Dockerfile? Какие основные инструкции он содержит?

Чем образ отличается от контейнера?

Зачем создавать непривилегированного пользователя в Dockerfile?

Что такое многоступенчатая сборка и зачем она нужна?

Как пробросить порт из контейнера на хост?

Лабораторная работа №18

Запуск многоконтейнерного приложения с использованием docker-compose

Цель: освоить оркестрацию нескольких контейнеров с помощью Docker Compose.

Краткие теоретические сведения

Docker Compose — инструмент для определения и запуска многоконтейнерных приложений.

Файл docker-compose.yml описывает:

сервисы (контейнеры);

образы или сборку из Dockerfile;

порты;

переменные окружения;

тома (volumes) для данных;

сети;

зависимости между сервисами.

Задание:

Создать многоконтейнерное приложение, включающее:

REST-сервис (из ЛР №17);

базу данных PostgreSQL или MySQL;

брокер сообщений RabbitMQ (опционально).

Написать docker-compose.yml, описывающий все сервисы.

Настроить REST-сервис на подключение к БД через переменные окружения (хост, порт, пользователь, пароль, БД).

Запустить все сервисы: docker-compose up -d

Проверить, что все контейнеры запущены и взаимодействуют.

Остановить и удалить контейнеры: docker-compose down

Составить отчет с содержимым docker-compose.yml, командами запуска/остановки и проверкой взаимодействия.

Контрольные вопросы:

Для чего нужен Docker Compose?

Как в docker-compose.yml описать зависимость одного сервиса от другого?

Как передать переменные окружения в контейнер через docker-compose?

Что такое volume в Docker и зачем он нужен для баз данных?

Как посмотреть логи всех сервисов в docker-compose?

Лабораторная работа №19
Развертывание функции в Yandex Cloud Functions / AWS Lambda, интеграция с API Gateway

Цель: освоить развертывание бессерверных функций (Serverless) в облачной платформе.

Краткие теоретические сведения

Serverless (бессерверные вычисления) — модель, при которой разработчик не управляет серверами: код запускается в ответ на события, а облачный провайдер автоматически масштабирует ресурсы.

FaaS (Function as a Service) — форма Serverless, где код разбивается на отдельные функции.

AWS Lambda — сервис FaaS от Amazon.

Yandex Cloud Functions — аналогичный сервис от Yandex.

API Gateway — сервис, который создает HTTP-эндпоинты и перенаправляет запросы в функции.

Задание:

Зарегистрироваться в Yandex Cloud (или AWS) и получить доступ к Cloud Functions / Lambda.

Написать простую функцию (Python или Node.js), которая:

принимает HTTP-запрос;

извлекает параметры (имя);

возвращает приветствие Hello, {name}!.

Развернуть функцию через консоль или CLI.

Создать API Gateway, подключенный к этой функции.

Получить публичный URL и протестировать функцию через браузер или cURL.

Добавить функцию, которая обрабатывает POST-запрос с JSON-телом и возвращает результат вычислений.

Составить отчет со скриншотами настройки, кодом функции, URL и примерами запросов/ответов.

Контрольные вопросы:

Что такое Serverless? Какие преимущества он дает?

Чем FaaS отличается от обычного развертывания приложения на сервере?

Что такое API Gateway и зачем он нужен?

Что такое триггер в Serverless? Какие виды триггеров бывают?

Как происходит масштабирование Serverless-функций?

Лабораторная работа №20

Настройка пайплайна CI/CD в GitLab CI / GitHub Actions для автоматического тестирования и сборки интеграционного сервиса

Цель: освоить настройку непрерывной интеграции и доставки (CI/CD) для интеграционного проекта.

Краткие теоретические сведения

CI (Continuous Integration) — практика частого слияния кода разработчиков в общую ветку с автоматической сборкой и тестированием.

CD (Continuous Delivery/Deployment) — автоматическое развертывание протестированного кода в среду (staging/production).

GitLab CI — встроенная CI/CD-система GitLab. Конфигурация в файле .gitlab-ci.yml.

GitHub Actions — CI/CD-система GitHub. Конфигурация в .github/workflows/*.yml.

Задание:

Создать репозиторий на GitLab или GitHub с кодом REST-сервиса (из ЛР №8).

Написать конфигурационный файл CI/CD, который выполняет:

установку зависимостей;

запуск юнит-тестов (если есть);

сборку Docker-образа;

(опционально) пуш образа в реестр (Docker Hub, GitLab Container Registry).

Запушить код в репозиторий и убедиться, что пайплайн запустился автоматически.

Изучить логи выполнения пайплайна.

Добавить стадию (stage) для деплоя (например, на тестовый сервер или в Docker Compose).

Составить отчет с содержимым конфигурационного файла, скриншотами пайплайна и описанием.

Контрольные вопросы:

Что такое CI/CD и зачем это нужно?

Какие этапы обычно включает CI/CD-пайплайн?

В чем разница между Continuous Delivery и Continuous Deployment?

Как в GitLab CI задать последовательность выполнения этапов?

Как в GitHub Actions запустить пайплайн только при пуше в ветку main?

Лабораторная работа №21

Реализация поддержки нескольких версий API в одном сервисе

Цель: освоить стратегии версионирования API и реализовать поддержку нескольких версий.

Краткие теоретические сведения

Версионирование API — стратегия, позволяющая вносить несовместимые изменения, не ломая существующих клиентов.

Основные стратегии версионирования:

URI — /v1/users, /v2/users (наиболее простая и наглядная).

Заголовки — Accept: application/vnd.myapp.v1+json.

Параметры запроса — /users?version=1.

Медиатипы — Content-Type: application/vnd.myapp.user-v1+json.

Семантическое версионирование (SemVer): MAJOR.MINOR.PATCH

MAJOR — несовместимые изменения API.

MINOR — добавлена функциональность, обратно совместимая.

PATCH — исправления ошибок, обратно совместимые.

Задание:

Взять REST-сервис (библиотека из ЛР №8).

Реализовать две версии API:

v1: поля title, author, year.

v2: добавляет поле isbn, изменяет формат года на объект { "start": 2020, "end": 2021 }.

Реализовать маршрутизацию по URI (/v1/books, /v2/books).

Убедиться, что клиенты v1 продолжают работать без изменений.

Добавить версионирование через заголовок API-Version: 2 (как альтернативный способ).

Документировать обе версии API.

Составить отчет с описанием стратегий, листингом кода и примерами запросов.

Контрольные вопросы:

Зачем нужно версионирование API?

Какие стратегии версионирования существуют? Какая самая простая?

Что такое семантическое версионирование (SemVer)?

Когда следует увеличивать MAJOR-версию?

Можно ли поддерживать несколько версий API без дублирования кода?

Лабораторная работа №22

Настройка сбора логов и метрик для микросервисного приложения

Цель: освоить централизованный сбор логов и метрик в распределённой системе.

Краткие теоретические сведения

ELK Stack (Elasticsearch, Logstash, Kibana) — стек для сбора, хранения и визуализации логов.

Prometheus — система сбора метрик (pull-модель).

Grafana — инструмент визуализации данных из Prometheus и других источников.

Jaeger — система распределённой трассировки (отслеживание запросов через несколько сервисов).

Задание:

Развернуть микросервисное приложение (2-3 сервиса) с помощью docker-compose.

Настроить логирование в формате JSON.

Развернуть ELK (или использовать Loki + Grafana как более легковесную альтернативу):

настроить сбор логов из контейнеров;

просмотреть логи в Kibana/Grafana.

Настроить сбор метрик через Prometheus:

добавить экспорт метрик в каждом сервисе (например, /metrics с использованием клиентской библиотеки Prometheus);

настроить Prometheus на сбор метрик;

визуализировать метрики (количество запросов, задержки) в Grafana.

Составить отчет с конфигурациями (docker-compose, prometheus.yml, настройки логов) и скриншотами дашбордов.

Контрольные вопросы:

Для чего нужен централизованный сбор логов?

Какие компоненты входят в ELK Stack?

Что такое метрики и чем они отличаются от логов?

Как Prometheus собирает метрики?

Что такое распределённая трассировка и когда она необходима?

Лабораторная работа №23

Визуализация метрик в Grafana

Цель: освоить создание информативных дашбордов в Grafana для мониторинга интеграционного решения.

Краткие теоретические сведения

Grafana — интерактивная платформа для визуализации данных.

Типы панелей (panels):

Time series — графики временных рядов;

Stat — большие числа (текущее значение);

Table — табличные данные;

Bar gauge, Pie chart — гистограммы и круговые диаграммы;

Heatmap — тепловые карты.

Задание:

Использовать Prometheus с метриками из предыдущей лабораторной работы.

Подключить Grafana к Prometheus как источнику данных.

Создать дашборд для мониторинга API-сервиса, включающий:

количество запросов в секунду (график);

95-й перцентиль задержки (latency);

код состояния HTTP (распределение по статусам);

использование памяти и CPU.

Создать дашборд для мониторинга очередей сообщений (RabbitMQ или Kafka):

длина очереди;

количество processed/ failed сообщений.

Настроить алерты (например, если длина очереди > 1000).

Сохранить дашборд в JSON и приложить к отчету.

Составить отчет со скриншотами дашбордов, описанием метрик и настройкой алертов.

Контрольные вопросы:

Что такое Grafana и для чего она используется?

Как подключить источник данных в Grafana?

Что такое перцентиль (например, p95) и зачем он нужен?

Как в Grafana создать алерт?

Можно ли экспортировать/импортировать дашборды в Grafana?

Основная литература

Практическая подготовка обучающихся специальности 10.05.01 Компьютерная безопасность : методические указания / составители И. В. Влацкая, С. А. Герасименко. — Оренбург : ОГУ, 2024. — 70 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/503273

Череватова, Т. Ф. Нормативное обеспечение в сфере информационных технологий и систем : учебное пособие для СПО / Т. Ф. Череватова. — 2-е изд., стер. — Санкт-Петербург : Лань, 2024. — 84 с. — ISBN 978-5-507-47632-9. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/398513

Баланов, А. Н. Защита информационных систем. Кибербезопасность : учебное пособие для вузов / А. Н. Баланов. — 3-е изд., стер. — Санкт-Петербург : Лань, 2026. — 280 с. — ISBN 978-5-507-56255-8. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/514704

Дополнительная литература

Федорова Г.И. Разработка, внедрение и адаптация программного обеспечения отраслевой направленности. Учебное пособие. Изд.: КУРС, Инфра-М. Среднее профессиональное образование. 2016 г. 336 стр.