Министерство науки и высшего образования Российской Федерации

ФГАОУ ВО «Уральский федеральный университет   
имени первого Президента России Б.Н. Ельцина»

Институт новых материалов и технологий

Кафедра «Теплофизика и информатика в металлургии»

**Контейнеризация программных приложений и брокер сообщений**

Методические указания к выполнению практической работы   
по дисциплине «Основы методологии Development Operation»

Екатеринбург

2023

УДК 519.25 (075.8) М861

Одобрено Учебно-методическим советом института новых материалов и технологий УрФУ

Рецензенты: …

Контейнеризация программных приложений и брокер сообщений: методические рекомендации для выполнения практических работ по дисциплине «Основы методологии Development Operation» / В.В. Лавров, И.А. Гурин. – Екатеринбург: УрФУ, 2023. – 16 с.

Методические рекомендации предназначены для обучения студентов всех форм обучения по направлению подготовки 09.03.02 «Информационные системы и технологии».

УДК 519.25 (075.8)

© Лавров В.В., Гурин И.А.

УрФУ, 2024

Содержание

[1. Установка Docker 4](#_Toc152160400)

[1.1. Что такое Docker? 4](#_Toc152160401)

[1.2. Установка Docker 4](#_Toc152160402)

[1.3. Проверка установки 5](#_Toc152160403)

[2. Работа с контейнерами 5](#_Toc152160404)

[2.1. Запуск контейнеров 5](#_Toc152160405)

[2.2. Управление контейнерами 6](#_Toc152160406)

[3. Создание Docker-образов 6](#_Toc152160407)

[3.1. Что такое Dockerfile? 7](#_Toc152160408)

[3.2. Создание Docker-образа 7](#_Toc152160409)

[4. Работа с Docker Hub 8](#_Toc152160410)

[5. Многоконтейнерные приложения с Docker Compose 9](#_Toc152160411)

[5.1. Установка Docker Compose 9](#_Toc152160412)

[5.2. Использование Docker Compose 10](#_Toc152160413)

[6. Конвейер данных реального времени Apache Kafka 11](#_Toc152160414)

[7. Задания студентам для выполнения практической работы 14](#_Toc152160415)

[7.1. Сборка и запуск контейнера в Docker 14](#_Toc152160416)

[7.2. Управление контейнерами с помощью Docker Compose и использование   
готовых образов 17](#_Toc152160417)

[7.3. Использование конвейера данных реального времени с Kafka 20](#_Toc152160418)

[Приложение А Список основных команд для работы с контейнерами 23](#_Toc152160419)

[Приложение Б Фрагменты листинга программного кода тестового приложения «Калькулятор» для выполнения задания 7.1 24](#_Toc152160420)

[Приложение В Фрагменты листинга программного кода тестового приложения «Калькулятор» для выполнения задания 7.2 26](#_Toc152160421)

[Приложение Г Фрагменты листинга программного кода тестового приложения «Калькулятор» для выполнения задания 7.3 30](#_Toc152160422)

# Практическая работа «Контейнеризация программных приложений и брокер сообщений»

*Цель практической работы –* познакомить студентов с технологией контейнеризации программных приложений Docker и её применением для упрощения процессов разработки, тестирования и развертывания приложений, а также использованием брокера сообщений Apache Kafka для управления потоками данных в реальном времени.

*Задачи*:

* обобщить, систематизировать, углубить, закрепить полученные теоретические знания по теме «Контейнеризация программных приложений» дисциплины «Основы методологии Development Operation»;
* сформировать умения применять полученные знания на практике, в частности: устанавливать Docker на операционную систему; создавать Docker-образ; создавать контейнер с использованием Docker-образа; определять и настраивать окружение контейнера с помощью Dockerfile; выполнять сборку образа; использовать Docker Hub для публикации и загрузки Docker-образов; использовать Docker Compose для запуска многоконтейнерного приложения; использовать брокер сообщений Apache Kafka для управления потоками данных в реальном времени;
* выработать при решении поставленных задач следующих профессионально значимых качеств: самостоятельность, ответственность, точность.

На практическом занятии студенты овладевают первоначальными профессиональными умениями и навыками, которые в дальнейшем закрепляются и совершенствуются в процессе прохождения производственной практики и подготовки выпускной квалификационной работы.

## 1. Установка Docker

Технология Docker является мощным инструментом контейнеризации, который играет ключевую роль в DevOps-процессах. Docker позволяет упаковывать приложения и все их зависимости в единый контейнер, который можно легко переносить и запускать на различных платформах.

Цель этого раздела – научиться устанавливать Docker на свои рабочие станции и убедиться, что установка прошла успешно. Это будет основой для последующих шагов, где мы будем работать с контейнерами и создавать Docker-образы.

Основное внимание при рассмотрении уделим следующим вопросам:

1. что такое Docker и какую роль она играет в DevOps-процессах;
2. инструкция по установке Docker на различные операционные системы (Windows, macOS, Linux);
3. проверка установки Docker с помощью команды docker --version или docker info.

### 1.1. Что такое Docker?

Docker – это открытая платформа, которая позволяет разработчикам автоматизировать процессы разработки, доставки и запуска приложений в изолированных средах, называемых контейнерами. Контейнеры похожи на виртуальные машины, но они легковеснее, быстрее и более эффективны в использовании ресурсов.

### 1.2. Установка Docker

Установка Docker на свою рабочую станцию может немного отличаться в зависимости от операционной системы, которую вы используете.

*Для пользователей Windows:*

1. проверьте версию операционной системы. Убедитесь, что у вас установлена Windows 10 Professional, Enterprise или Education. Docker for Windows не поддерживает Windows 10 Home Edition;
2. для установки Docker Desktop перейдите на официальный сайт Docker: <https://www.docker.com/products/docker-desktop>;
3. загрузите установщик Docker Desktop для Windows;
4. запустите установщик и следуйте инструкциям на экране;
5. после установки запустите Docker Desktop из меню "Пуск".

*Для пользователей macOS:*

1. убедитесь, что у вас установлена macOS версии 10.13 или выше;
2. для установки перейдите на официальный сайт Docker: <https://www.docker.com/products/docker-desktop>;
3. загрузите установщик Docker Desktop для Mac;
4. перетащите иконку Docker в папку "Приложения";
5. запустите Docker Desktop из папки "Приложения".

*Для пользователей Linux:*

1. установите Docker Engine: откройте терминал и выполните следующие команды:

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io

1. добавьте пользователя в группу "docker". Чтобы использовать Docker без прав администратора, добавьте своего пользователя в группу "docker":

sudo usermod -aG docker $USER

### 1.3. Проверка установки

После завершения установки Docker, необходимо проверить, что все работает правильно. Откройте терминал (или командную строку) и выполните следующую команду:

docker --version

Если установка прошла успешно, вы должны увидеть вывод версии Docker, который указывает на то, что Docker установлен и готов к использованию.

Список основных команд для управления контейнерами Docker представлен в Приложении А.

## 2. Работа с контейнерами

В этом разделе мы изучим основные команды Docker, которые позволят создавать и взаимодействовать с контейнерами. Понимание работы с контейнерами является фундаментом для разработки, тестирования и развертывания приложений с помощью Docker.

### 2.1. Запуск контейнеров

Запуск простого контейнера осуществляется с помощью команды docker run. Эта команда позволяет нам создать и запустить контейнер на основе Docker-образа.

Для примера, давайте запустим контейнер с использованием одного из самых популярных Docker-образов – образа "hello-world". Выполните следующую команду в вашем терминале:

docker run hello-world

В результате выполнения команды Docker загрузит образ "hello-world" из Docker Hub (если его еще нет на вашей системе), создаст контейнер и выполнит его. Вы увидите приветственное сообщение «Hello from Docker!», которое подтверждает успешность установки Docker.

Чтобы просмотреть список всех активных контейнеров на вашей системе, выполните команду:

docker ps

Эта команда отобразит информацию о контейнерах, которые в данный момент запущены.

### 2.2. Управление контейнерами

Когда контейнер запущен, мы можем выполнять различные операции с ним, такие как остановка, удаление и т.д.

*Остановка контейнера*. Если у вас есть работающий контейнер, остановите его с помощью команды:

docker stop <CONTAINER\_ID или CONTAINER\_NAME>

Здесь <CONTAINER\_ID> – это уникальный идентификатор контейнера, или вы можете использовать <CONTAINER\_NAME>, если вы задали имя контейнеру при его запуске. Используйте команду docker ps для получения списка активных контейнеров и их идентификаторов.

*Удаление контейнера*. После остановки контейнера, вы можете удалить его с помощью команды:

docker rm <CONTAINER\_ID или CONTAINER\_NAME>

Важно помнить, что контейнеры изолированы друг от друга и от хостовой системы. Контейнеры являются временными, и после их остановки, изменения в них не сохраняются. Если вам необходимо сохранить состояние и изменения внутри контейнера, вы можете использовать механизмы сохранения данных Docker, такие как примонтированные тома (mounted volumes).

В следующем разделе мы научимся создавать собственные Docker-образы с помощью Dockerfile.

## 3. Создание Docker-образов

Docker-образы представляют собой шаблоны, на основе которых Docker создает контейнеры. Создание собственных образов позволяет нам упаковать наше приложение и его зависимости в единый, переносимый контейнер.

Рассмотрим процессы создания Docker-образов, определения и настройки окружения контейнера с помощью Dockerfile, а также сборку образов с использованием команды docker build.

### 3.1. Что такое Dockerfile?

Dockerfile – это текстовый файл, содержащий инструкции по тому, как Docker должен собирать Docker-образ. Dockerfile определяет базовый образ, устанавливает зависимости, копирует файлы и выполняет другие необходимые операции для создания полноценного образа.

### 3.2. Создание Docker-образа

Для создания Docker-образа требуется создать файл Dockerfile и определить в нем все необходимые инструкции.

*Создание Dockerfile*. Для начала создайте новый текстовый файл с именем "Dockerfile" (без расширения) в вашей рабочей директории или любой другой удобной папке. Простой пример Dockerfile для создания образа с установленным ASP.NET 6.0 выглядит следующим образом:

# Используем базовый образ с ASP.NET 6.0

FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base

# Устанавливаем рабочую директорию внутри контейнера

WORKDIR /app

# Используем SDK

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build

# Копируем файлы в контейнер

COPY . /src

# Устанавливаем рабочую директорию внутри контейнера

WORKDIR /src

# Устанавливаем зависимости приложения и параметры компиляции

RUN ls

RUN dotnet restore

RUN dotnet build "./Calculator.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "./Calculator.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish ./

# Определяем команду запуска контейнера

ENTRYPOINT ["dotnet", "Calculator.dll"]

Представленный Dockerfile организует процесс сборки, установки зависимостей, сборки и публикации приложения ASP.NET 6.0 внутри контейнера Docker, готового к запуску с помощью dotnet. Dockerfile создает образ для приложения на ASP.NET 6.0, используя два базовых образа - mcr.microsoft.com/dotnet/aspnet:6.0 и mcr.microsoft.com/dotnet/sdk:6.0.

Прокомментируем команды, используемые в представленном примере Dockerfile:

* FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base: устанавливает базовый образ с ASP.NET 6.0 и назначает ему псевдоним base. Этот образ используется для создания основы приложения;
* WORKDIR /app: устанавливает рабочую директорию внутри контейнера на /app;
* FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build: использует образ SDK для сборки приложения;
* COPY . /src: копирует все файлы из текущего каталога хоста в директорию /src внутри контейнера;
* WORKDIR /src: устанавливает /src как рабочую директорию внутри контейнера;
* RUN ls: выполняет команду ls для отображения содержимого текущей директории в контейнере (для отладочных целей);
* RUN dotnet restore и RUN dotnet build "./Calculator.csproj" -c Release -o /app/build: устанавливает зависимости приложения с помощью dotnet restore, затем выполняет сборку приложения с помощью dotnet build с указанием пути для сохранения результатов в /app/build;
* FROM build AS publish: создает временный этап publish на основе build;
* RUN dotnet publish "./Calculator.csproj" -c Release -o /app/publish: выполняет публикацию приложения в директорию /app/publish внутри временного этапа publish;
* FROM base AS final: создает финальный образ на основе базового образа base;
* WORKDIR /app: устанавливает /app как рабочую директорию внутри контейнера;
* COPY --from=publish /app/publish ./: копирует файлы из директории /app/publish временного этапа publish в текущую директорию /app финального образа;
* ENTRYPOINT ["dotnet", "Calculator.dll"]: определяет команду, которая будет выполнена при запуске контейнера. В данном случае, запускается исполняемый файл Calculator.dll с помощью dotnet.

*Сборка Docker-образа*. Теперь, когда у нас есть Dockerfile, мы можем собрать Docker-образ с помощью команды docker build. Перейдите в папку, где находится ваш Dockerfile, и выполните команду:

docker build -t my-web-app .

Здесь -t позволяет задать имя образа (в данном случае «my-web-app»), а точка означает текущую директорию, где находится Dockerfile. Docker выполнит инструкции из Dockerfile и создаст образ.

*Запуск контейнера на основе созданного образа.* Теперь, когда у нас есть собственный Docker-образ, давайте запустим контейнер на его основе. Выполните команду:

docker run -d -p 5001:5001 my-web-app

Здесь -d означает запуск контейнера в фоновом режиме, а -p задает проброс портов (в данном случае, контейнерный порт 5001 будет доступен на хостовом порту 5001).

В следующем разделе мы изучим возможность публикации и обмена Docker-образами.

## 4. Работа с Docker Hub

Docker Hub представляет собой облачный репозиторий образов Docker. Docker Hub предоставляет возможность публиковать, обнаруживать и совместно использовать Docker-образы, что делает его ценным инструментом для разработчиков и DevOps-инженеров.

Рассмотрим способы использования Docker Hub для публикации и загрузки Docker-образов, а также для обмена ими с другими участниками команды.

Для начала работы с Docker Hub требуется создать аккаунт на веб-сайте: <https://hub.docker.com/>. После создания аккаунта вы будете иметь доступ к своему личному пространству и сможете публиковать свои образы.

*Вход в Docker Hub*. Чтобы выполнить вход в свой аккаунт на Docker Hub, выполните команду:

docker login

Следуйте указаниям, чтобы ввести ваше имя пользователя и пароль.

*Публикация Docker-образа на Docker Hub*. После успешного входа вы можете опубликовать свой Docker-образ на Docker Hub. Для этого выполните следующие шаги:

1. Переименуйте ваш образ. Переименуйте ваш Docker-образ согласно правилам Docker Hub. Обычно образы называются в формате <USERNAME>/<IMAGE\_NAME>, где <USERNAME> – это ваше имя пользователя на Docker Hub, а <IMAGE\_NAME> – имя образа. Выполните команду:

docker tag my-web-app <USERNAME>/<IMAGE\_NAME>

1. Опубликуйте образ на Docker Hub. Выполните команду:

docker push <USERNAME>/<IMAGE\_NAME>

Эта команда загрузит ваш образ на Docker Hub, и он станет доступен для использования другими разработчиками.

1. Загрузка Docker-образа с Docker Hub. Для загрузки образа с Docker Hub на другой компьютер или сервер выполните команду:

docker pull <USERNAME>/<IMAGE\_NAME>

В следующем разделе мы изучим использование Docker Compose для управления многоконтейнерными приложениями.

## 5. Многоконтейнерные приложения с Docker Compose

Docker Compose – это инструмент для определения и запуска многоконтейнерных Docker-приложений с помощью одного файла конфигурации. С его помощью вы можете определить все контейнеры, их зависимости, настройки сети, примонтированные тома и другие параметры, необходимые для работы всего приложения.

### 5.1. Установка Docker Compose

Если у вас уже установлен Docker, то в большинстве случаев Docker Compose идет в комплекте с Docker. Однако, если Docker Compose не установлен, выполните следующие шаги для его установки:

* *Проверьте наличие Docker Compose*. Перед установкой Docker Compose, проверьте, не установлен ли он уже на вашей системе, выполнив команду:

docker-compose --version

Если вы видите версию Docker Compose, значит, он уже установлен.

* *Установка Docker Compose (Linux/Mac)*. Для пользователей Linux или macOS выполните следующую команду:

sudo curl -L "https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

* *Установка Docker Compose (Windows).* Для пользователей Windows скачайте исполняемый файл Docker Compose отсюда: <https://github.com/docker/compose/releases/latest/download/docker-compose-Windows-x86_64.exe>. Затем переместите файл в любую папку, включенную в переменную среды PATH.

### 5.2. Использование Docker Compose

Теперь, когда у вас установлен Docker Compose, давайте изучим, как использовать его для управления многоконтейнерными приложениями. Выполните следующую последовательность действий:

1. *Создание файла docker-compose.yml*. Создайте новый файл с именем docker-compose.yml в вашей рабочей директории. В этом файле мы опишем все контейнеры и их настройки для нашего многоконтейнерного приложения.
2. *Определение контейнеров и их зависимостей*. В файле docker-compose.yml опишите все необходимые контейнеры для вашего приложения, а также определите их зависимости и настройки. Пример содержания файла:

version: '3.7'

services:

mariadb:

container\_name: web-mariadb

hostname: mariadb

image: mariadb:10.5

restart: always

environment:

MARIADB\_ROOT\_PASSWORD: password

MARIADB\_DATABASE: webdb

MARIADB\_USER: webdb

MARIADB\_PASSWORD: password

ports:

- "0.0.0.0:5023:3306"

web:

container\_name: web-api

hostname: api

build: MyWebApplication/

ports:

- "0.0.0.0:5003:5003"

Представленный файл docker-compose.yml определяет конфигурацию для запуска двух сервисов: базы данных MariaDB – mariadb, и веб-приложения – web. Он описывает настройки контейнеров, их параметры и зависимости, а также определяет какие порты будут открыты для внешнего доступа к каждому сервису. Когда вы запустите docker-compose up из той же директории, где находится этот файл, Docker будет создавать и запускать два контейнера согласно этим настройкам.

Прокомментируем команды, используемые в представленном примере docker-compose.yml:

* mariadb сервис:
* container\_name: web-mariadb: задает имя контейнера как web-mariadb;
* hostname: mariadb: устанавливает имя хоста контейнера как mariadb;
* image: mariadb:10.5: использует образ MariaDB версии 10.5 для запуска контейнера;
* restart: always: настраивает перезапуск контейнера в случае его выхода из строя;
* environment: устанавливает переменные окружения для MariaDB, такие как пароль для root пользователя, название базы данных, пользователь и его пароль;
* ports: пробрасывает порт из контейнера (3306 – порт MariaDB) на хост машины по адресу 0.0.0.0:5023.
* web сервис:
* container\_name: web-api: задает имя контейнера как web-api;
* hostname: api: устанавливает имя хоста контейнера как api;
* build: MyWebApplication/: сборка контейнера происходит на основе указанного Dockerfile или пути к контексту сборки (MyWebApplication/);
* ports: пробрасывает порт из контейнера (5003 – порт для веб-приложения) на хост машины по адресу 0.0.0.0:5003.

1. *Запуск приложения с помощью Docker Compose*. Чтобы запустить ваше многоконтейнерное приложение, выполните команду:

docker compose up

Docker Compose выполнит инструкции из файла docker-compose.yml и создаст и запустит контейнеры согласно определенным настройкам.

1. *Остановка приложения*. Чтобы остановить приложение и удалить созданные контейнеры, выполните команду:

docker compose down

Таким образом, Docker Compose значительно упрощает процесс развертывания и масштабирования сложных приложений, состоящих из нескольких взаимосвязанных сервисов.

## 6. Конвейер данных реального времени Apache Kafka

Apache Kafka – это распределенная система сообщений, разработанная для управления потоками данных в реальном времени. Она предоставляет высокопроизводительную и масштабируемую платформу для обработки, хранения и передачи потоков данных.

Apache Kafka была создана внутри LinkedIn для управления их потоками данных и была открыта для общественности в 2011 году как open-source проект. Её цель – обеспечить надежное, масштабируемое и устойчивое решение для работы с потоками данных в реальном времени.

Основные компоненты системы представлены на рисунке 6.1:

* *Брокеры (brokers):* серверы, отвечающие за обработку и хранение данных. Брокеры составляют кластер Kafka;
* *Топики (topics):* категории, в которых сообщения публикуются и хранятся. Они делятся на партиции для обеспечения масштабируемости и отказоустойчивости;
* *Продюсеры (Producers):* компоненты, отправляющие сообщения в топики;
* *Консьюмеры (Consumers):* компоненты, читающие и обрабатывающие сообщения из топиков.



Рис. 6.1. Основные компоненты системы сообщений Apache Kafka

Преимущества Kafka заключаются в следующем:

* высокая пропускная способность. Способность обрабатывать огромные объемы данных в реальном времени;
* масштабируемость. Возможность горизонтального и вертикального масштабирования для удовлетворения изменяющихся потребностей;
* отказоустойчивость. Репликация данных и устойчивость к отказам обеспечивают надежность системы.

Операции с Apache Kafka включают создание топиков, отправку сообщений (публикацию) и чтение сообщений (подписку) из топиков. Рассмотрим конкретные примеры для каждой из этих операций.

*Создание топиков.*

Для создания топика в Kafka используется утилита командной строки kafka-topics. Например, чтобы создать топик с именем "test\_topic" с одной партицией (partition) и одной репликой (replica), необходимо выполнить команду:

kafka-topics --bootstrap-server localhost:9092 --create --topic test\_topic --partitions 1 --replication-factor 1

Эта команда создает топик "test\_topic" с одной партицией и одной репликой. Это базовый пример, но можно настроить большее количество партиций и реплик в зависимости от требований.

Прокомментируем термины «партиция» и «реплика».

В контексте Apache Kafka, партиция – это основной механизм, используемый для распределения данных в топиках. Топик в Kafka разделен на несколько партиций, и каждая партиция является упорядоченной последовательностью записей.

Каждая партиция представляет собой упорядоченный журнал записей, и данные в партиции хранятся в порядке их добавления. Каждая запись в партиции имеет уникальный смещение (offset), который используется для идентификации конкретной записи внутри партиции.

Партиции позволяют Kafka горизонтально масштабироваться, поскольку они позволяют распределять данные по разным брокерам в кластере. Они также обеспечивают отказоустойчивость, так как данные в топике реплицируются по разным брокерам внутри кластера Kafka.

Когда данные публикуются в топик, они попадают в одну из партиций. Механизм распределения записей по партициям может осуществляться различными стратегиями, например, раунд-робин, хеширование ключа или иная логика, определенная пользователем.

Использование партиций позволяет достичь параллельной обработки данных в Kafka. Консьюмеры могут читать данные из параллельно обрабатываемых партиций, что обеспечивает высокую пропускную способность при чтении и обработке сообщений.

В контексте Apache Kafka, реплика – это копия партиции, которая хранится на другом брокере в кластере. Каждая партиция в Kafka может иметь одну или несколько реплик.

Репликация данных – ключевой механизм, обеспечивающий отказоустойчивость и надежность данных в Kafka. Каждая запись, отправленная в топик, сохраняется в основной (leader) реплике партиции, которая обслуживает все записи и запросы на чтение из этой партиции. При этом также создаются реплики на других брокерах.

*Отправка сообщений (Публикация).*

Для отправки сообщений в топик используется продюсер Kafka. Например, создадим Java-приложение для отправки сообщений в топик "test\_topic":

import org.apache.kafka.clients.producer.\*;

public class KafkaProducerExample {

public static void main(String[] args) {

String topicName = "test\_topic";

String bootstrapServers = "localhost:9092";

Properties props = new Properties();

props.put("bootstrap.servers", bootstrapServers);

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);

try {

for (int i = 0; i < 10; i++) {

producer.send(new ProducerRecord<>(topicName, Integer.toString(i), "Message " + i));

}

} catch (Exception e) {

e.printStackTrace();

} finally {

producer.close();

}

}

}

Этот код создает продюсера Kafka, который отправляет 10 сообщений в топик "test\_topic". Он использует сериализаторы строк для ключа и значения сообщения.

*Чтение сообщений (Подписка).*

Для чтения сообщений из топика используется консьюмер Kafka. Например, создадим Java-приложение для чтения сообщений из топика "test\_topic":

import org.apache.kafka.clients.consumer.\*;

import java.time.Duration;

import java.util.Collections;

import java.util.Properties;

public class KafkaConsumerExample {

public static void main(String[] args) {

String topicName = "test\_topic";

String bootstrapServers = "localhost:9092";

String groupId = "group1";

Properties props = new Properties();

props.put("bootstrap.servers", bootstrapServers);

props.put("group.id", groupId);

props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

Consumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Collections.singletonList(topicName));

try {

while (true) {

ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));

for (ConsumerRecord<String, String> record : records) {

System.out.printf("Received message: key = %s, value = %s%n", record.key(), record.value());

}

}

} catch (Exception e) {

e.printStackTrace();

} finally {

consumer.close();

}

}

}

Этот код создает консьюмера Kafka, который подписывается на топик "test\_topic" и выводит полученные сообщения с их ключами и значениями.

Каждый из этих примеров демонстрирует основные операции с Apache Kafka: создание топиков, отправку сообщений в топик и чтение сообщений из топика. Это основа для работы с данными в Kafka, которую можно дополнить более сложными сценариями и настройками в зависимости от потребностей проекта.

## 7. Задания студентам для выполнения практической работы

Студенты должны самостоятельно выполнить все нижеперечисленные задания и предоставить развернутый отчет по проделанной работе.

### 7.1. Сборка и запуск контейнера в Docker

**Этап 1.1.** *Установить Docker на свою операционную систему и выполнить проверку установки*

Чтобы установить Docker на свою операционную систему, необходимо перейти на официальный сайт Docker (<https://www.docker.com/>) и ознакомиться с инструкцией по установке для различных операционных систем. Выберите инструкции, соответствующие вашей ОС, и следуйте им. После завершения установки, проверьте, что Docker установлен правильно, выполнив команду docker --version в командной строке или терминале.

**Этап 1.2.** *Разработать тестовое приложение Visual Studio .NET Core*

Для начала необходимо убедиться, что на рабочем компьютере установлена Visual Studio и .NET Core SDK. Затем создайте новый проект в Visual Studio и выберите шаблон, соответствующий типу приложения, который необходимо разработать – веб-приложение ASP.NET Core MVC. Напишите программный код приложения «Калькулятор» – простой калькулятор с веб-интерфейсом, который будет выполнять основные арифметические операции (сложение, разность, умножение и деление) и отображать результат. Архитектура приложения «Калькулятор» представлена на рисунке 7.1.1.

В Приложении Б представлены примеры листинга программного кода, фрагменты из которого надо скопировать в свой проект приложения. На главной странице веб-приложения, в футере должны отображаться ФИО и номер группы студента.

Выполните сборку и протестируйте работу приложения, чтобы убедиться, что оно выполняет требуемые функции.



Рис. 7.1.1. Архитектура приложения «Калькулятор»

**Этап 1.3.** *Создать контейнер с использованием Docker для тестового приложения ASP.NET Core Web Application, определить и настроить окружение контейнера с помощью Dockerfile*

В качестве тестового приложения используйте созданный проект «Калькулятор» – простой калькулятор с веб-интерфейсом, который выполняет основные арифметические операции (сложение, разность, умножение и деление) и отображает результат на веб-странице.

Сначала необходимо создать файл с именем "Dockerfile" в корневой папке тестового проекта. Для этого откройте текстовый редактор и добавьте следующий код в файл:

# Используем базовый образ с ASP.NET 6.0

FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base

# Устанавливаем рабочую директорию внутри контейнера

WORKDIR /app

# Используем SDK

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build

# Копируем файлы в контейнер

COPY . /src

# Устанавливаем рабочую директорию внутри контейнера

WORKDIR /src

# Устанавливаем зависимости приложения и параметры компиляции

RUN ls

RUN dotnet restore

RUN dotnet build "./Calculator.csproj" -c Release -o /app/build

FROM build AS publish

RUN dotnet publish "./Calculator.csproj" -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish ./

# Определяем команду запуска контейнера

ENTRYPOINT ["dotnet", "Calculator.dll"]

**Этап 1.4.** *Разместить приложение в системе контроля версий GitHub*

После того, как тестовое приложение будет работать без ошибок, создайте личный репозиторий на GitHub. Для этого перейдите на сайт GitHub (https://github.com/), войдите в свой аккаунт (или зарегистрируйтесь, если аккаунта нет) и создайте новый репозиторий. Затем, добавьте тестовый проект в Git и привяжите его к личному репозиторию на GitHub.

**Этап 1.5.** *Выполнить сборку образа с использованием команды docker build*

Сначала в терминальном режиме на удаленном сервере необходимо клонировать проект из личного репозитория GitHub. Для этого выполнить команду:

git clone <HTTPS>

Здесь <HTTPS> – это HTTPS-адрес проекта в GitHub.

Далее необходимо осуществить сборку Docker-образа. Откройте командную строку или терминал в папке с вашим Dockerfile, затем выполните команду для сборки Docker-образа:

docker build -t <IMAGE\_NAME>:latest .

Здесь <IMAGE\_NAME> – это имя образа. Вы можете заменить его на любое удобное вам имя. При выполнении работы рекомендуется имя образа начинать с номера из студенческого журнала. Например, студент под номером 3 использует имя образа 3-calculator.

Проверьте существование созданного образа в списке активных контейнеров и их идентификаторов с помощью команды:

docker ps -a

**Этап 1.6.** *Создать контейнер с использованием Docker-образа и выполнить его*

Для создания контейнера с использованием Docker-образа воспользуйтесь командой docker run:

docker run -d -p 5001:5001 --name <CONTAINER\_NAME> <IMAGE\_ID | IMAGE\_NAME>

Здесь:

-d означает, что контейнер будет запущен в фоновом режиме (daemon mode);

-p 5001:5001 пробрасывает порт 5001 на вашем хосте к порту 5001 в контейнере (на котором запущено ваше ASP.NET Core приложение). Обратите внимание, что каждый студент должен указать индивидуальный номер порта, на котором будет работать приложение. Номер порта должен соответствовать порядковому номеру из студенческого журнала. Например, студент под номером 3 использует порт 5003, а студент под номером 15 – порт 5015;

<CONTAINER\_NAME> – это уникальное в рамках системы имя контейнера, по которому в дальнейшем его можно будет остановить, запустить или удалить;

<IMAGE\_ID> – это идентификатор или имя образа. Используйте команду docker images -a для получения списка всех образов.

Используйте команду docker ps -a для получения списка активных контейнеров и их идентификаторов:

docker ps -a

По завершении работы в контейнере необходимо выйти из него, используя команду exit.

**Этап 1.7.** *Продемонстрировать возможность публикации и остановки контейнера*

Запустите (опубликуйте) контейнер с помощью команды:

docker start <CONTAINER\_ID или CONTAINER\_NAME>

Здесь <CONTAINER\_ID> – это уникальный идентификатор контейнера, или вы можете использовать <CONTAINER\_NAME>, если вы задали имя контейнеру при его запуске.

Убедитесь в браузере, что веб-приложение работает.

Остановите работающий контейнер с помощью команды:

docker stop <CONTAINER\_ID или CONTAINER\_NAME>

Убедитесь в браузере, что веб-приложение перестало работать.

В отчете по практической работе студенты должны описать все шаги и предоставить скриншоты результатов выполнения задания. Вставить в отчет следующие рисунки:

1. Фрагмент Visual Studio c отображением листинга программного кода файла appsettings.json.
2. Окно личного репозитория GitHub с размещением тестового проекта.
3. Окно терминальной консоли с результатами клонирования тестового проекта.
4. Окно терминальной консоли с демонстрацией созданного образа.
5. Окно терминальной консоли с демонстрацией запущенного контейнера.
6. Окно веб-браузера с демонстрацией работающего тестового приложения на выбранном порту.

В отчет по практической работе добавить Приложение, в котором представить листинги программного кода файлов:

* контроллера;
* appsettings.json;
* appsettings.Development.json;
* Program.cs.

### 7.2. Управление контейнерами с помощью Docker Compose и использование готовых образов

**Этап 2.1.** *Усовершенствовать тестовое приложение Visual Studio .NET Core*

Необходимо усовершенствовать тестовое приложение «Калькулятор» – добавить хранение всех выполненных операций в базе данных MariaDB. Архитектура усовершенствованного приложения «Калькулятор» представлена на рисунке 7.2.1.



Рис. 7.2.1. Архитектура приложения «Калькулятор» с использованием MariaDB

В Приложении В представлены примеры листинга программного кода, фрагменты из которого надо скопировать в свой проект приложения. Обратите внимание, что каждый студент должен указать индивидуальный номер порта, на котором будет работать приложение. Номер порта должен соответствовать порядковому номеру из студенческого журнала.

На главной странице веб-приложения, в футере должны отображаться ФИО и номер группы студента.

В проект необходимо дополнительно подключить следующие пакеты NuGet:

* Microsoft.EntityFrameworkCore (6.0.21);
* Microsoft.EntityFrameworkCore.Design (6.0.21);
* Microsoft.EntityFrameworkCore.Tools (6.0.21);
* Pomelo.EntityFrameworkCore.MySql (6.0.2);

Выполните сборку и протестируйте работу приложения, чтобы убедиться, что оно выполняет требуемые функции.

**Этап 2.2.** *Создать файл docker-compose.yml опишите все необходимые контейнеры для вашего приложения, а также определите их зависимости и настройки*.

В корневой папке тестового проекта необходимо создать файл с именем docker-compose.yml. Для этого откройте текстовый редактор и добавьте следующий код в файл:

version: '3.7'

services:

mariadb:

container\_name: web-mariadb

hostname: mariadb

image: mariadb:10.5

restart: always

environment:

MARIADB\_ROOT\_PASSWORD: password

MARIADB\_DATABASE: webdb

MARIADB\_USER: webdb

MARIADB\_PASSWORD: password

ports:

- "0.0.0.0:5003:3306"

web:

container\_name: web-app-calculator

hostname: app

build: Calculator/

ports:

- "0.0.0.0:5003:5003"

В этом файле описаны настройки двух контейнеров web-mariadb (база данных MariaDB) и web-app-calculator (веб-приложение «Калькулятор»). Контейнер web-mariadb использует следующий формат портов <IP-сервера>:<Порт-приложения>: внешний порт 5003, а порт приложения 3306. Контейнер web-app-calculator использует для работы следующий формат портов <IP-сервера>:<Порт-приложения>: внешний порт 5003 и внутренний 5003.

При выполнении работы студенту необходимо установить номера портов, соответствующих номеру из студенческого журнала.

**Этап 2.3.** *Разместить приложение в системе контроля версий GitHub*

После того, как тестовое приложение будет работать без ошибок, загрузите его в личный репозиторий на GitHub.

**Этап 2.4.** *Выполнить сборку образов с использованием команды docker compose*

В терминальном режиме на удаленном сервере необходимо клонировать проект из личного репозитория GitHub. Для этого выполнить команду:

git clone <HTTPS>

Здесь <HTTPS> – это HTTPS-адрес проекта в GitHub.

Далее необходимо запустить многоконтейнерное приложение. Откройте командную строку или терминал в папке с файлом docker-compose.yml, затем выполните команду:

docker compose up -d

Проверьте существование созданных образов в списке активных контейнеров и их идентификаторов с помощью команды:

docker ps -a

**Этап 2.5.** *Установить программу HeidiSQL для сопровождения базы данных в СУБД MariaDB*

Скачать и установить на рабочий компьютер программу HeidiSQL с сайта <https://www.heidisql.com/>. Запустить файл heidisql.exe.

В окне программы создать новый сеанс. Настроить подключение к хосту IP 93.88.178.186. Ввести следующие реквизиты подключения:

* пользователь root;
* пароль password;
* порт 5023.

Сохранить настройки сеанса и открыть сеанс. В появившемся окне создать базу данных с именем в формате <номер варианта>-WebCalculatorDb. Например, студент с вариантом 3 создает базу данных с именем 3-WebCalculatorDb.

**Этап 2.6.** *Настроить миграцию базы данных в MariaDB*

В среде Visual Studio, где открыт тестовый проект «Калькулятор» вызвать Консоль диспетчера пакетов и выполнить команду:

add-migration m\_init

В случае успешного выполнения в проекте будет создана папка Migrations с двумя файлами.

Далее выполнить обновление базы данных с помощью команды:

update-database

В окне программы HeidiSQL убедиться, что появилась таблица DataInputVariants (ID\_DataInputVariant, Operand\_1, Operand\_2, Type\_operation, Result).

**Этап 2.7.** *Запустить приложение Калькулятор, выполнить несколько тестовых расчетов и убедиться, что данные попадают в базу данных, созданную в MariaDB*

Запустить приложение Калькулятор в браузере по ссылке http://93.88.178.186:5003/. Номер порта 5003 соответствует номеру 3 из студенческого журнала. Выполнить несколько тестовых расчетов и убедиться, что данные попадают в базу данных 3-WebCalculatorDb.

В отчете по практической работе студенты должны описать все шаги и предоставить скриншоты результатов выполнения задания. Вставить в отчет следующие рисунки:

1. Фрагмент Visual Studio c отображением листинга программного кода файла appsettings.json.
2. Фрагмент Visual Studio c отображением листинга программного кода файла appsettings.Development.json.
3. Окно личного репозитория GitHub с размещением тестового проекта с открытым файлом docker-compose.yml.
4. Окно терминальной консоли с результатами клонирования тестового проекта.
5. Окно терминальной консоли с демонстрацией созданных образов.
6. Окно программы HeidiSQL с отображением наполненной базы в MariaDB.
7. Окно веб-браузера с демонстрацией работающего тестового приложения на выбранном порту.

В отчет по практической работе добавить Приложение, в котором представить листинги программного кода файлов:

* контроллера;
* appsettings.json;
* appsettings.Development.json;
* Program.cs.

### 7.3. Использование конвейера данных реального времени с Kafka

**Этап 3.1.** *Усовершенствовать тестовое приложение Visual Studio .NET Core*

Разработку новой версии приложения необходимо производить в отдельной ветке репозитория GitHub с названием kafka.

Необходимо усовершенствовать структуру тестового приложения «Калькулятор», которое было разработано в п. 7.2, с использованием системы Apache Kafka. В частности, необходимо сформировать обмен сообщениями между веб-страницей ввода исходных данных и веб-страницей получения результатов, записи выполненной операции в базу данных MariaDB через брокер сообщений Kafka. Архитектура усовершенствованного приложения «Калькулятор» представлена на рисунке 7.3.1.

В Приложении Г представлены примеры листинга программного кода, фрагменты из которого надо скопировать в свой проект приложения. На главной странице веб-приложения, в футере должны отображаться ФИО и номер группы студента.

В проект необходимо дополнительно подключить пакет расширения Confluent.Kafka 2.3.0.

Выполните сборку и протестируйте работу приложения, чтобы убедиться, что оно выполняет требуемые функции.



Рис. 7.3.1. Архитектура приложения «Калькулятор» с использованием Kafka

**Этап 3.2.** *Разместить приложение в системе контроля версий GitHub*

После того, как тестовое приложение будет работать без ошибок, загрузите его в личный репозиторий на GitHub.

**Этап 3.3.** *Выполнить сборку образов с использованием команды docker compose*

В терминальном режиме на удаленном сервере необходимо клонировать проект из личного репозитория GitHub. Для этого выполнить команду:

git clone <HTTPS>

Здесь <HTTPS> – это HTTPS-адрес проекта в GitHub.

Далее необходимо запустить многоконтейнерное приложение. Откройте командную строку или терминал в папке с файлом docker-compose.yml, затем выполните команду:

docker compose up -d

Проверьте существование созданных образов в списке активных контейнеров и их идентификаторов с помощью команды:

docker ps -a

**Этап 3.4.** *Установить программу HeidiSQL для сопровождения базы данных в СУБД MariaDB*

Скачать и установить на рабочий компьютер программу HeidiSQL с сайта <https://www.heidisql.com/>. Запустить файл heidisql.exe.

В окне программы создать новый сеанс. Настроить подключение к хосту IP 93.88.178.186. Ввести следующие реквизиты подключения:

* пользователь root;
* пароль password;
* порт 5023.

Сохранить настройки сеанса и открыть сеанс. В появившемся окне создать базу данных myWebCalculatorDb.

**Этап 3.5.** *Настроить миграцию базы данных в MariaDB*

В среде Visual Studio, где открыт тестовый проект «Калькулятор» вызвать Консоль диспетчера пакетов и выполнить команду:

add-migration m\_init

В случае успешного выполнения в проекте будет создана папка Migrations с двумя файлами.

Далее выполнить:

update-database

В окне программы HeidiSQL убедиться, что появилась таблица DataInputVariants (ID\_DataInputVariant, Operand\_1, Operand\_2, Type\_operation, Result).

**Этап 3.6.** *Запустить приложение Калькулятор, выполнить несколько тестовых расчетов и убедиться, что данные попадают в базу данных myWebCalculatorDb и сведения о сообщениях выводятся в топике Kafka*

Запустить приложение Калькулятор в браузере по ссылке http://93.88.178.186:5015/. Здесь в примере ссылки указан порт приложения 5015, что соответствует номеру 15 из студенческого журнала. Выполнить несколько тестовых расчетов и убедиться, что данные попадают в базу данных 15-WebCalculatorDb.

Проверить запись в Kafka в созданном топике по ссылке http://93.88.178.186:10025/.

В отчете по практической работе студенты должны описать все шаги и предоставить скриншоты результатов выполнения задания. Вставить в отчет следующие рисунки:

1. Фрагмент Visual Studio c отображением листинга программного кода файла appsettings.json.
2. Фрагмент Visual Studio c отображением листинга программного кода файла appsettings.Development.json.
3. Окно терминальной консоли с результатами клонирования тестового проекта.
4. Окно терминальной консоли с демонстрацией созданных образов.
5. Окно программы HeidiSQL с отображением наполненной базы в MariaDB.
6. Окно веб-браузера с демонстрацией записей в Kafka в созданном топике.
7. Окно веб-браузера с демонстрацией работающего тестового приложения на выбранном порту.

В отчет по практической работе добавить Приложение, в котором представить листинги программного кода файлов:

* контроллера;
* KafkaConsumerService.cs;
* appsettings.json;
* appsettings.Development.json;
* Program.cs.

## Приложение А Список основных команд для работы с контейнерами

Таблица А.1

Список основных команд для управления контейнерами Docker

|  |  |
| --- | --- |
| **Формат команды** | **Описание команды** |
| docker run <имя\_образа> | Запускает контейнер на основе указанного образа |
| docker ps | Показывает список запущенных контейнеров |
| docker ps -a | Показывает список всех контейнеров (включая остановленные) |
| docker stop <ID\_контейнера> | Останавливает указанный контейнер |
| docker start <ID\_контейнера> | Запускает остановленный контейнер |
| docker rm <ID\_контейнера> | Удаляет контейнер по его ID |
| docker images | Показывает список загруженных образов |
| docker pull <имя\_образа> | Загружает образ из репозитория Docker Hub |
| docker rmi <ID\_образа> | Удаляет образ по его ID |
| docker exec -it <ID\_контейнера> <команда> | Запускает новый процесс внутри контейнера с указанной командой |
| docker build -t <имя\_образа> <путь> | Собирает образ из Dockerfile по указанному пути с тегом <имя\_образа> |
| docker inspect <ID\_контейнера> | Выводит информацию о контейнере в формате JSON |
| docker logs <ID\_контейнера> | Показывает логи контейнера |
| docker cp <локальный\_файл> <ID\_контейнера>:<путь\_в\_контейнере> | Копирует файлы/директории между локальной машиной и контейнером |
| docker-compose up | Запускает приложение согласно настройкам из файла docker-compose.yml |
| docker compose up -d | Запускает все сервисы, описанные в файле docker-compose.yml, и запускает их в фоновом режиме, позволяя освободить текущую консоль для других команд |
| docker-compose down | Останавливает и удаляет контейнеры, созданные через docker-compose |
| docker network ls | Показывает список сетей Docker |
| docker network inspect <имя\_сети> | Выводит информацию о сети в формате JSON |
| docker network create <имя\_сети> | Создает новую пользовательскую сеть Docker |
| docker network connect <имя\_сети> <ID\_контейнера> | Подключает контейнер к указанной сети |

## Приложение Б Фрагменты листинга программного кода тестового приложения «Калькулятор» для выполнения задания 7.1

*Программный код контроллера, файл CalculatorController.cs*

using Microsoft.AspNetCore.Mvc;

namespace Calculator.Controllers

{

public enum Operation { Add, Subtract, Multiply, Divide }

public class CalculatorController : Controller

{

[HttpGet]

public IActionResult Index()

{

return View();

}

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Calculate(double num1, double num2, Operation operation)

{

double result = 0;

switch (operation)

{

case Operation.Add:

result = num1 + num2;

break;

case Operation.Subtract:

result = num1 - num2;

break;

case Operation.Multiply:

result = num1 \* num2;

break;

case Operation.Divide:

result = num1 / num2;

break;

}

ViewBag.Result = result;

return View("Index");

}

}

}

*Программный код представления, файл Index.cshtml*

@{

ViewData["Title"] = "Simple Calculator";

}

<h2>@ViewData["Title"]</h2>

<form method="post" asp-controller="Calculator" asp-action="Calculate">

@Html.AntiForgeryToken()

<input type="text" name="num1" required />

<select name="operation" required>

<option value="Add">+</option>

<option value="Subtract">-</option>

<option value="Multiply">\*</option>

<option value="Divide">/</option>

</select>

<input type="text" name="num2" required />

<input type="submit" value="Calculate" />

</form>

@if (ViewBag.Result != null)

{

<h3>Result: @ViewBag.Result</h3>

}

*Программный код файла appsettings.json*

{

"Kestrel": {

"Endpoints": {

"Http": {

"Url": "http://0.0.0.0:5001" // Заменить последние 2 цифры порта на порядковый номер из студенческого журнала. Например, порт 5012 соответствует номеру 12

}

}

},

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

},

"AllowedHosts": "\*"

}

## Приложение В Фрагменты листинга программного кода тестового приложения «Калькулятор» для выполнения задания 7.2

*Программный код контроллера, файл CalculatorController.cs*

using Calculator.Data;

using Calculator.Models;

using Microsoft.AspNetCore.Components.Forms;

using Microsoft.AspNetCore.Mvc;

namespace Calculator.Controllers

{

public enum Operation { Add, Subtract, Multiply, Divide }

public class CalculatorController : Controller

{

private CalculatorContext \_context;

public CalculatorController(CalculatorContext context)

{

\_context = context;

}

[HttpGet]

public IActionResult Index()

{

return View();

}

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Calculate(double num1, double num2, Operation operation)

{

double result = 0;

switch (operation)

{

case Operation.Add:

result = num1 + num2;

break;

case Operation.Subtract:

result = num1 - num2;

break;

case Operation.Multiply:

result = num1 \* num2;

break;

case Operation.Divide:

result = num1 / num2;

break;

}

ViewBag.Result = result;

DataInputVariant dataInputVariant = new DataInputVariant();

dataInputVariant.Operand\_1 = num1.ToString();

dataInputVariant.Operand\_2 = num2.ToString();

dataInputVariant.Type\_operation = operation.ToString();

dataInputVariant.Result = result.ToString();

\_context.DataInputVariants.Add(dataInputVariant);

\_context.SaveChanges();

return View("Index");

}

}

}

*Программный код класса DataInputVariant.cs в папке Data*

using Microsoft.EntityFrameworkCore.Metadata.Internal;

using System.ComponentModel.DataAnnotations.Schema;

using System.ComponentModel.DataAnnotations;

namespace Calculator.Data

{

public class DataInputVariant

{

[Key]

public int ID\_DataInputVariant { get; set; }

[Column(TypeName = "varchar(128)")]

public string? Operand\_1 { get; set; }

[Column(TypeName = "varchar(128)")]

public string? Operand\_2 { get; set; }

[Column(TypeName = "varchar(128)")]

public string? Type\_operation { get; set; }

[Column(TypeName = "varchar(128)")]

public string? Result { get; set; }

}

}

*Программный код класса CalculatorContext.cs в папке Data*

using Microsoft.EntityFrameworkCore;

namespace Calculator.Data

{

public class CalculatorContext : DbContext

{

public DbSet<DataInputVariant> DataInputVariants { get; set; }

public CalculatorContext(DbContextOptions<CalculatorContext> options) : base(options)

{

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

//OnModelCreating(modelBuilder);

}

}

}

*Программный код файла appsettings.json*

{

"ConnectionStrings": {

"DefaultConnection": "Server=mariadb; Database=myWebCalculatorDb; Uid=root; Pwd=password; Character Set=utf8; ConvertZeroDatetime=True;"

},

"Kestrel": {

"Endpoints": {

"Http": {

"Url": "http://0.0.0.0:5003" // Заменить последние 2 цифры порта на порядковый номер из студенческого журнала. Например, порт 5012 соответствует номеру 12

}

}

},

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

},

"AllowedHosts": "\*"

}

*Программный код файла appsettings.Development.json*

{

"ConnectionStrings": {

"DefaultConnection": "Server=93.88.178.186; Port=5023; Database=myWebCalculatorDb; Uid=root; Pwd=password; Character Set=utf8; ConvertZeroDatetime=True;"

},

"DetailedErrors": true,

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

}

}

*Программный код файла Program.cs*

using Microsoft.EntityFrameworkCore;

using Calculator.Data;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

string mariadbCS = builder.Configuration.GetConnectionString("DefaultConnection");

builder.Services.AddDbContext<CalculatorContext>(options =>

{

options.UseMySql(mariadbCS, new MySqlServerVersion(new Version(10, 5, 15)));

});

builder.Services.AddRazorPages();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Calculator/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

pattern: "{controller=Calculator}/{action=Index}/{id?}");

app.Run();

## Приложение Г Фрагменты листинга программного кода тестового приложения «Калькулятор» для выполнения задания 7.3

*Программный код контроллера, файл CalculatorController.cs*

using Calculator.Data;

using Calculator.Models;

using Calculator.Services;

using Confluent.Kafka;

using Microsoft.AspNetCore.Mvc;

using System.Text.Json;

namespace Calculator.Controllers

{

public class CalculatorController : Controller

{

private readonly CalculatorContext \_context;

private readonly KafkaProducerService<Null, string> \_producer;

public CalculatorController(CalculatorContext context, KafkaProducerService<Null, string> producer)

{

\_context = context;

\_producer = producer;

}

/// <summary>

/// Отображение страницы Index.

/// </summary>

public IActionResult Index()

{

var data = \_context.DataInputVariants.OrderByDescending(x => x.ID\_DataInputVariant).ToList();

return View(data);

}

/// <summary>

/// Обработка запроса на вычисление.

/// </summary>

/// <param name="num1">Первый операнд.</param>

/// <param name="num2">Второй операнд.</param>

/// <param name="operation">Тип операции (сложение, вычитание, умножение, деление).</param>

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> ProcessingCalculationRequest(double num1, double num2, Operation operation)

{

// Подготовка объекта для расчета

var dataInputVariant = new DataInputVariant

{

Operand\_1 = num1,

Operand\_2 = num2,

Type\_operation = operation,

};

// Отправка данных в Kafka

await SendDataToKafka(dataInputVariant);

// Перенаправление на страницу Index

return RedirectToAction(nameof(Index));

}

public IActionResult Callback([FromBody] DataInputVariant inputData)

{

// Сохранение данных и результата в базе данных

SaveDataAndResult(inputData);

return Ok();

}

/// <summary>

/// Сохранение данных и результата в базе данных.

/// </summary>

/// <param name="num1">Первый операнд.</param>

/// <param name="num2">Второй операнд.</param>

/// <param name="operation">Тип операции (сложение, вычитание, умножение, деление).</param>

/// <param name="result">Результат математической операции.</param>

/// <returns>Объект с данными и результатом.</returns>

private DataInputVariant SaveDataAndResult(DataInputVariant inputData)

{

\_context.DataInputVariants.Add(inputData);

\_context.SaveChanges();

return inputData;

}

/// <summary>

/// Отправка данных в Kafka.

/// </summary>

/// <param name="dataInputVariant">Объект с данными и результатом.</param>

/// <returns>Task.</returns>

private async Task SendDataToKafka(DataInputVariant dataInputVariant)

{

var json = JsonSerializer.Serialize(dataInputVariant);

await \_producer.ProduceAsync("lavrov", new Message<Null, string> { Value = json });

}

}

}

*Программный код файла KafkaConsumerService.cs*

using Confluent.Kafka;

using Calculator.Models;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.Hosting;

using System;

using System.Text.Json;

using System.Threading;

using System.Threading.Tasks;

using Calculator.Controllers;

using Calculator.Data;

namespace Calculator.Services

{

using System;

using System.Text.Json;

using System.Threading;

using System.Threading.Tasks;

using Confluent.Kafka;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Hosting;

using System.Net.Http;

public class KafkaConsumerService : BackgroundService

{

private readonly string \_topic;

private readonly IConsumer<Null, string> \_kafkaConsumer;

private readonly IServiceProvider \_serviceProvider;

private readonly IHttpClientFactory \_clientFactory;

public KafkaConsumerService(IConfiguration config, IServiceProvider serviceProvider, IHttpClientFactory clientFactory)

{

// Конфигурирование настроек Kafka и инициализация компонентов

var consumerConfig = new ConsumerConfig();

config.GetSection("Kafka:ConsumerSettings").Bind(consumerConfig);

\_topic = config.GetValue<string>("Kafka:TopicName");

\_kafkaConsumer = new ConsumerBuilder<Null, string>(consumerConfig).Build();

\_serviceProvider = serviceProvider;

\_clientFactory = clientFactory;

}

/// <summary>

/// Выполнение работы Kafka Consumer'а.

/// </summary>

protected override Task ExecuteAsync(CancellationToken stoppingToken)

{

return Task.Run(() => StartConsumerLoop(stoppingToken), stoppingToken);

}

/// <summary>

/// Цикл обработки сообщений из Kafka.

/// </summary>

/// <param name="cancellationToken">Токен отмены операции.</param>

private async Task StartConsumerLoop(CancellationToken cancellationToken)

{

\_kafkaConsumer.Subscribe(\_topic);

while (!cancellationToken.IsCancellationRequested)

{

try

{

var cr = \_kafkaConsumer.Consume(cancellationToken);

var ip = cr.Message.Value;

// Исходные данные

var inputData = JsonSerializer.Deserialize<DataInputVariant>(cr.Message.Value);

// Выполнение расчета

var result = CalculatorLibrary.CalculateOperation(inputData.Operand\_1, inputData.Operand\_2, inputData.Type\_operation);

inputData.Result = result.ToString();

var httpClient = \_clientFactory.CreateClient();

await httpClient.PostAsJsonAsync($"http://localhost:5015/Calculator/Callback", inputData);

// Обработка сообщения...

Console.WriteLine($"Message key: {cr.Message.Key}, value: {cr.Message.Value}");

}

catch (OperationCanceledException)

{

break;

}

catch (ConsumeException e)

{

if (e.Error.IsFatal)

{

// https://github.com/edenhill/librdkafka/blob/master/INTRODUCTION.md#fatal-consumer-errors

break;

}

}

catch (Exception e)

{

break;

}

}

}

/// <summary>

/// Очистка ресурсов Consumer'а при завершении работы сервиса.

/// </summary>

public override void Dispose()

{

\_kafkaConsumer.Close(); // Фиксация оффсетов и корректное выход из группы.

\_kafkaConsumer.Dispose();

base.Dispose();

}

}

}

*Программный код файла KafkaProducerHandler.cs*

using Confluent.Kafka;

using Microsoft.Extensions.Configuration;

using System;

namespace Calculator.Services

{

public class KafkaProducerHandler : IDisposable

{

IProducer<byte[], byte[]> kafkaProducer;

public KafkaProducerHandler(IConfiguration config)

{

var conf = new ProducerConfig();

config.GetSection("Kafka:ProducerSettings").Bind(conf);

kafkaProducer = new ProducerBuilder<byte[], byte[]>(conf).Build();

}

public Handle Handle { get => kafkaProducer.Handle; }

public void Dispose()

{

// Block until all outstanding produce requests have completed (with or

// without error).

kafkaProducer.Flush();

kafkaProducer.Dispose();

}

}

}

*Программный код файла KafkaProducerService.cs*

using Confluent.Kafka;

using System;

using System.Threading.Tasks;

namespace Calculator.Services

{

public class KafkaProducerService<K, V>

{

IProducer<K, V> kafkaHandle;

public KafkaProducerService(KafkaProducerHandler handle)

{

kafkaHandle = new DependentProducerBuilder<K, V>(handle.Handle).Build();

}

/// <summary>

/// Asychronously produce a message and expose delivery information

/// via the returned Task. Use this method of producing if you would

/// like to await the result before flow of execution continues.

/// <summary>

public Task ProduceAsync(string topic, Message<K, V> message)

=> kafkaHandle.ProduceAsync(topic, message);

/// <summary>

/// Asynchronously produce a message and expose delivery information

/// via the provided callback function. Use this method of producing

/// if you would like flow of execution to continue immediately, and

/// handle delivery information out-of-band.

/// </summary>

public void Produce(string topic, Message<K, V> message, Action<DeliveryReport<K, V>> deliveryHandler = null)

=> kafkaHandle.Produce(topic, message, deliveryHandler);

public void Flush(TimeSpan timeout)

=> kafkaHandle.Flush(timeout);

}

}

*Программный код файла Operation.cs*

using System.ComponentModel.DataAnnotations;

namespace Calculator.Models

{

public enum Operation

{

[Display(Name = "Сложение")]

Add = 1,

[Display(Name = "Вычитание")]

Subtract = 2,

[Display(Name = "Умножение")]

Multiply = 3,

[Display(Name = "Деление")]

Divide = 4

}

}

*Программный код файла CalculatorLibrary.cs*

using Calculator.Controllers;

namespace Calculator.Models

{

public class CalculatorLibrary

{

/// <summary>

/// Выполнение математической операции.

/// </summary>

/// <param name="num1">Первый операнд.</param>

/// <param name="num2">Второй операнд.</param>

/// <param name="operation">Тип операции (сложение, вычитание, умножение, деление).</param>

/// <returns>Результат математической операции.</returns>

public static double CalculateOperation(double num1, double num2, Operation operation)

{

return operation switch

{

Operation.Add => num1 + num2,

Operation.Subtract => num1 - num2,

Operation.Multiply => num1 \* num2,

Operation.Divide => num1 / num2,

\_ => throw new ArgumentOutOfRangeException(nameof(operation), "Invalid operation"),

};

}

}

}

*Программный код файла EnumExtensions.cs*

using System.ComponentModel.DataAnnotations;

using System.Reflection;

namespace Calculator.Extension

{

public static class EnumExtensions

{

/// <summary>

/// Retrieves the <see cref="DisplayAttribute.Name" /> property on the <see cref="DisplayAttribute" />

/// of the current enum value, or the enum's member name if the <see cref="DisplayAttribute" /> is not present.

/// </summary>

/// <param name="val">This enum member to get the name for.</param>

/// <returns>The <see cref="DisplayAttribute.Name" /> property on the <see cref="DisplayAttribute" /> attribute, if present.</returns>

public static string GetDisplayName(this Enum val)

{

return val.GetType()

.GetMember(val.ToString())

.FirstOrDefault()

?.GetCustomAttribute<DisplayAttribute>(false)

?.Name

?? val.ToString();

}

}

}

*Программный код файла CalculatorContext.cs*

using Microsoft.EntityFrameworkCore;

namespace Calculator.Data

{

public class CalculatorContext : DbContext

{

public DbSet<DataInputVariant> DataInputVariants { get; set; }

public CalculatorContext(DbContextOptions<CalculatorContext> options) : base(options)

{

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

//OnModelCreating(modelBuilder);

}

}

}

*Программный код файла DataInputVariant.cs*

using Microsoft.EntityFrameworkCore.Metadata.Internal;

using System.ComponentModel.DataAnnotations.Schema;

using System.ComponentModel.DataAnnotations;

using Calculator.Models;

namespace Calculator.Data

{

public class DataInputVariant

{

[Key]

public int ID\_DataInputVariant { get; set; } // Уникальный идентификатор для объекта DataInputVariant

public double Operand\_1 { get; set; } // Первый операнд для операции

public double Operand\_2 { get; set; } // Второй операнд для операции

public Operation Type\_operation { get; set; } // Тип операции (например, сложение, вычитание и т.д.)

[Column(TypeName = "varchar(128)")]

public string? Result { get; set; } // Результат выполнения операции

}

}

*Программный код файла appsettings.json*

{

"ConnectionStrings": {

"DefaultConnection": "Server=mariadb; Database=myWebCalculatorDb; Uid=root; Pwd=password; Character Set=utf8; ConvertZeroDatetime=True;"

},

"Kafka": {

"ProducerSettings": {

"BootstrapServers": "kafka:9092",

"SaslMechanism": "Plain",

"SecurityProtocol": "Plaintext"

},

"ConsumerSettings": {

"BootstrapServers": "kafka:9092",

"GroupId": "lavrov",

"SaslMechanism": "Plain",

"SecurityProtocol": "Plaintext",

"AutoOffsetReset": "Earliest",

"EnableAutoCommit": true

},

"TopicName": "lavrov"

},

"Kestrel": {

"Endpoints": {

"Http": {

"Url": "http://0.0.0.0:5015" // Заменить последние 2 цифры порта на порядковый номер из студенческого журнала. Например, порт 5012 соответствует номеру 12

}

}

},

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

},

"AllowedHosts": "\*"

}

*Программный код файла appsettings.Development.json*

{

"ConnectionStrings": {

"DefaultConnection": "Server=93.88.178.186; Port=5023; Database=myWebCalculatorDb; Uid=root; Pwd=password; Character Set=utf8; ConvertZeroDatetime=True;"

},

"Kafka": {

"ProducerSettings": {

"BootstrapServers": "93.88.178.186:9094"

},

"ConsumerSettings": {

"BootstrapServers": "93.88.178.186:9094"

}

},

"DetailedErrors": true,

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft": "Warning",

"Microsoft.Hosting.Lifetime": "Information"

}

}

}

*Программный код файла Program.cs*

using Microsoft.EntityFrameworkCore;

using Calculator.Data;

using Calculator.Services;

using Confluent.Kafka;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllersWithViews();

string mariadbCS = builder.Configuration.GetConnectionString("DefaultConnection");

builder.Services.AddDbContext<CalculatorContext>(options =>

{

options.UseMySql(mariadbCS, new MySqlServerVersion(new Version(10, 5, 15)));

});

builder.Services.AddRazorPages();

builder.Services.AddHttpClient();

builder.Services.AddHostedService<KafkaConsumerService>();

builder.Services.AddSingleton<KafkaProducerHandler>();

builder.Services.AddSingleton<KafkaProducerService<Null, string>>();

var app = builder.Build();

// Configure the HTTP request pipeline.

if (!app.Environment.IsDevelopment())

{

app.UseExceptionHandler("/Calculator/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(

name: "default",

pattern: "{controller=Calculator}/{action=Index}/{id?}");

app.Run();