Существуют два подхода к архитектуре приложения – **монолитный** и **микросервисный**.

 

**В монолитной архитектуре** приложение можно разделить по слоям, но его части все равно тесно связаны между собой (их пишут внутри одного проекта и собирают одновременно). Подходит для небольших приложений, которые не будут развиваться и усложняться.

Преимущества – скорость разработки, линейный процесс деплоя, полный взаимный доступ частей приложения, легкая поддерживать транзакционность процессов, единая БД.

Недостатки – сложности изменения или добавления нового функционала и технологий, проблемы с масштабированием, много кода в одном проекте, сложно разобраться в проекте.

**В микросервисной архитектуре** приложение делится на сервисы, которые отвечают за отдельную функциональность и общаются между собой. Каждый микросервис связан со своей БД, которая хранит только те данные, которые относятся к его функциональности. К другим БД у микросервиса доступа нет.

Преимущества – легкость изменения или добавления нового функционала и технологий, масштабируемость, легче переиспользовать код, удобнее распределять задачи при разработке.

Недостатки – скорость разработки, задержки и ошибки при общении микросервисов между собой, ниже производительность, сложно поддерживать транзакционность процессов, несколько БД, сложный процесс *CI/CD* (деплоя).

Межсервисные взаимодействия могут быть синхронными и асинхронными.



При **синхронном взаимодействии** задачи выполняются друг за другом. На примере - сверху вниз.

Недостатки:

* Клиенту нужно дождаться ответа на запрос, чтобы убедиться, что все прошло успешно. Иногда это занимает много времени, при этом процесс загружен ожиданием, что отнимает ресурсы.
* Приходиться продумывать сценарии обработки долгих запросов. Чтобы не ждать вечно, время ожидания ограничивают.
* Если клиент выполняет слишком много запросов, сервер может не выдержать нагрузки.



При **асинхронном взаимодействии** между клиентом и сервером используется посредник – **брокер** **сообщений** (например, *Apache Kafka* или *RabbitMQ*). Он принимает запросы от сервисов в сформированные ею очереди, сохраняет у себя, а затем передает другим сервисам.

Преимущества:

* Процессы не блокируются, потому что клиент не ждет ответа от сервера.
* Сервер не перегружен, а все избыточные запросы остаются в очереди.
* На очередь могут подписаться несколько серверов. Например, и микросервис по отправке писем, и микросервис по возврату средств.
* Не нужно продумывать сценарии обработки долгих запросов. Клиент не ждет ответа.

Недостатки:

* Нужно разворачивать и настраивать дополнительный сервис (брокер сообщений).
* Если запросы попадают в очередь быстрее, чем их успевает обрабатывать сервер, то рано или поздно закончится допустимый объем хранимых сообщений.

***Docker*** – это платформа, которая позволяет запустить виртуальную среду (**контейнер**) и задеплоить в нее приложение. Таким образом обеспечивается стандартизация и исключаются возможные конфликты с другим ПО или настройками ОС.

Нативной ОС для *Docker* является *Linux*, поэтому запуск на *Windows* должен происходить внутри виртуальной машины с ОС *Linux*. Десктопная версия работает на основе гипервизора *Hyper-V*, а виртуальная – на основе подсистемы *Linux* (*WSL2*).

Список команд для установки на *Linux*:

*// установка утилиты для скачивания файлов*

*sudo apt install curl*

*// скачать скрипт для установки Docker*

*curl -fsSL https://get.docker.com -o get-docker.sh*

*// запустить скачанный скрипт*

*sh get-docker.sh*

*// обновить список пакетов*

*sudo apt update*

*// установить необходимые пакеты для загрузки через HTTPS*

*sudo apt install \*

*apt-transport-https \*

*ca-certificates \*

*curl \*

*gnupg-agent \*

*software-properties-common -y*

*// добавить ключ GPG для подтверждения подлинности в процессе установки (выведется ОК)*

*curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -*

*// добавить репозиторий Docker в пакеты apt*

*sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"*

*// снова обновить список пакетов*

*sudo apt update*

*// установить Docker Community Edition и Docker Compose*

*sudo apt install docker-ce docker-compose -y*

*// проверить, что Docker работает*

*sudo systemctl status docker*

***Dockerfile*** – специальный файл, описывающий настройки для создания *docker-image*. Команда *“docker build”* собирает из него *docker-image*. Сборка образа происходит послойно и кэшируется после каждого изменения.

***Docker-image*** – это образ-шаблон, который содержит все необходимое для автономной работы контейнера. В образ может быть упакована ОС, сервер, интерпретатор языка, виртуальное окружение проекта, файлы приложения.

***Docker*-контейнер** – это окружение, которое создается из образа, как по шаблону. Он запускается командой *“docker run имя\_образа”*. Например, *“docker run hello-world”*.

Рассмотрим процесс создания и запуска *docker*-контейнера:

1. С помощью команды *“mvn clean package”* соберем текущее приложение в *jar*-архив в папке *target*.
2. С помощью команды *“java -jar target/[имя\_архива].jar”* запустим приложение из *jar*-архива и проверим его работоспособность. Командой *Ctrl+C* остановим его.
3. Добавим в корень проекта текстовый файл без расширения и назовем его *Dockerfile*. Он будет служить схемой, по которой Docker будет собирать образ для контейнеров.
4. Добавим в *Dockerfile* инструкции для сборки. Каждая инструкция пишется заглавными буквами, а их выполнение происходит последовательно.

*FROM amazoncorretto:11-alpine-jdk // ОС Linux с java 11 от Amazon Correto*

*COPY target/\*.jar app.jar // скопировать все jar-файлы в контейнер*

*ENTRYPOINT ["java","-jar","/app.jar"] // запустить java-приложение после старта контейнера*

Команда ***FROM <image>:<tag>*** определяет базовый образ, на основе которого создается локальный образ. Необязательный параметр ***tag*** указывает конкретную версию образа. По умолчанию используется версия с тегом ***latest***. Подробнее о тегах можно почитать [тут](https://hub.docker.com/_/amazoncorretto?tab=tags).

Команда ***COPY <локальная директория> <путь внутри контейнера>*** копирует файлы и директории из указанной локальной директории в директорию внутри контейнера. Если целевой директории не существует, то она создается.

Консольная команда ***ENTRYPOINT <command> <param1> <param2>*** выполняется внутри контейнера после его запуска.

Команда ***WORKDIR <директория>*** устанавливает рабочую директорию в образе.

Команда ***ENV*** позволяет задать переменные среды окружения внутри образа.

Команда ***ENV TZ=”Europe/Moscow”*** позволяет установить часовой пояс +3 внутри образа.

Команда ***ARG*** позволяет задать значение для аргумента сразу при сборке образа.

1. Соберем собственный образ *dock\_image*.

*docker build -t dock\_image .*

Команда ***build*** запускает сборку образа по инструкциям из *Dockerfile*.

Ключ ***-t*** позволяет задать имя образу.

***Точка*** внутри команды – это путь до *Dockerfile*, на основе которого производится сборка.

1. Запустим собственный контейнер.

*docker run --name dock\_container -p 8080:8080 dock\_image*

Команда ***run*** запускает контейнер.

Ключ ***--name*** позволяет задать имя контейнеру. В случае его отсутствия, *Docker* сгенерирует его самостоятельно.

Ключ ***-p <внешний порт>:<внутренний порт приложения>*** позволяет настроить порты.

Ключ ***-d*** позволяет не занимать текущий процесс консоли запущенным приложением.

***dock\_image*** – образ, из которого будет запущен контейнер.

*docker image ls // вывести список образов*

*docker image rm имя\_образа // удалить образ*

*docker container ls // вывести список запущенных контейнеров*

*docker container ls -a // вывести список всех контейнеров*

*docker container stop <container\_id> // остановить контейнер*

*docker container start <container\_id> // запустить контейнер*

*docker container // вывести список всех команд для контейнеров*

*docker --version // узнать версию докера*

*docker logs -f <container\_id> // вывести логи контейнера*

Ключ ***-f*** включает режим непрерывного просмотра. Все изменения будут отображаться автоматически.

*docker exec -it dock\_container /bin/sh // открыть терминал для работы с контейнером*

*exit // выйти из терминала контейнера*

Команда ***docker exec*** позволяет выполнять команды в запущенном контейнере.

Ключ ***-it*** запускает интерактивный режим.

Параметр ***/bin/sh*** запускает командную строку. Для разных ОС используются разные параметры.

Подробнее по работе с командами докера можно [тут](https://docs.docker.com/engine/reference/commandline/container/).

Все его данные, накопленные за время работы контейнера, уничтожаются после его удаления или перезагрузки. Эту проблему решает ***volume*** – внешняя директория, доступная одному или нескольким контейнерам. Подробнее можно посмотреть [тут](https://docs.docker.com/storage/volumes/).



Секретные ключи, доступы и токены не следует хранить в коде. Передать их в контейнер можно несколькими способами:

* Прописать переменные окружения прямо в *Dockerfile* с помощью команды *ENV*.

*ENV token 12345*

* Задать переменные окружения при сборке контейнера, выполнив команду run с ключом ***-e***

*docker run <имя образа> -e token=12345*

* Сохранить переменные окружения в настроечном файле *Docker Compose*.

Для загрузки значений из переменных окружения нужно изменить файл *application.properties*:

*jdbc.url=jdbc:postgresql://${DB\_HOST}:${DB\_PORT}/${DB\_NAME}*

*jdbc.username=${POSTGRES\_USER}*

*jdbc.password=${POSTGRES\_PASSWORD}*

Для управления взаимодействием нескольких контейнеров используется ***Docker Compose***. Например, в одном контейнере работает сервис, а в другом – база данных.

Инструкции по развертыванию проекта в нескольких контейнерах пишут в файле ***docker-compose.yaml***. Например:

*# версия Docker Compose*

*version: '3.8'*

*# имена и описания контейнеров, которые должны быть развёрнуты*

*services:*

*# описание контейнера db*

*db:*

*# образ, из которого должен быть запущен контейнер*

*image: postgres:13.7-alpine*

*# volume и связанная с ним директория в контейнере*

*volumes:*

*- db-data:/var/lib/postgresql/data/*

*# переменные окружения*

*environment:*

*- POSTGRES\_DB=later*

*- POSTGRES\_USER=root*

*- POSTGRES\_PASSWORD=root*

*# пробросим текущую тайм-зону в контейнер*

*volumes:*

* + */etc/timezone:/etc/timezone:ro*
  + */etc/localtime:/etc/ localtime:ro*

*web:*

*build: .*

*ports:*

*- "8080:8080"*

*# "зависит от"*

*depends\_on:*

*- db*

*# перезапускать если упадет БД*

*restart: on-failure*

*# переменные окружения*

*environment:*

*- DB\_NAME=later*

*- POSTGRES\_USER=root*

*- POSTGRES\_PASSWORD=root*

*- DB\_HOST=db*

*- DB\_PORT=5432*

*# пробросим текущую тайм-зону в контейнер*

*volumes:*

* + */etc/timezone:/etc/timezone:ro*
  + */etc/localtime:/etc/ localtime:ro*

*# создать свой именованный том db-data*

*volumes:  
 db-data:*

Команда ***services*** задает список имен и описаний контейнеров, которые должны быть развернуты. Имена могут быть любыми.

Команда ***depends\_on*** определяет, после какого контейнера должен быть запущен описываемый.

Команды ***docker-compose up*** и ***docker-compose stop*** запускает и останавливает проект соответственно.

Подробнее про модульные приложения можно почитать [тут](https://spring.io/guides/gs/multi-module/).

Для отладки приложения, необходимо запустить его со следующими параметрами:

*-agentlib:jdwp=transport=dt\_socket,server=y,suspend=n,address=\*:[номер порта для отладки]*

*transport=dt\_socket* – в качестве способа подключения к *JVM* будут использоваться сокеты.

*server=y* – к приложению будут подключаться для отладки, а не наоборот.

*suspend=n* – *JVM* будет ждать, пока отладчик подключится для выполнения команд.

*address=\*:[номер порта]* – номер порта для подключения (обычно +1 от используемого порта)

В результате финальная версия *Dockerfile* будет выглядеть так:

*FROM amazoncorretto:11-alpine-jdk*

***ENV JAVA\_TOOL\_OPTIONS -agentlib:jdwp=transport=dt\_socket,server=y,suspend=n,address=\*:8081***

*COPY target/\*.jar app.jar*

*ENTRYPOINT ["java","-jar","/app.jar"]*

Запустим контейнер, пробросив уже два порта:

*docker run --name dock\_container -p 8081:8081 -p 8080:8080 dock\_image*

Теперь можно запустить в *IDEA* удаленный дебаг приложения.



***DockerHub*** – облачное хранилище *images*.

Имя загружаемого образа должно выглядеть так – ***username/imagename::tag***. Имя может содержать только строчные буквы, цифры, дефисы (не с начала) и нижние подчеркивания. **Тэг** – необязательная часть имени, например, версия. По умолчанию все загружаемые образы публичные.

*billglasses/gates:v1.1985 // пример правильного имени*

*docker login // авторизация в докере*

*docker push billglasses/gates:v1.1985 // загрузка образа на DockerHub*

Команды ***docker push <image name>*** и ***docker pull <image name>*** загружают и скачивают образ с *DockerHub* соответственно.

Рассмотрим приемы работы в командной строке.

*<консольная команда> --help // вывести справку по команде*

*pwd // вывести путь до текущей директории*

*ls // вывести содержимое текущей директории*

*ls - l // вывести содержимое текущей директории, включая скрытые*

*ls -a // вывести содержимое текущей директории и информацию о них*

*ls -t // вывести содержимое текущей директории, отсортировав по возрастанию даты*

*ls <путь до директории> //вывести содержимое другой директории*

*cd licenses // переход в другую директорию*

*cd / // возврат в корневую директорию*

*cd ./dev // переход в директорию dev в текущем каталоге*

*cd .. // возврат на один уровень выше*

*cd ../folder2 // возврат на один уровень выше и провалиться в соседнюю директорию*

*touch answer.txt // создать один файл. Если файл существует – обновить дату редактирования*

*touch answer.txt pom.xml // создать два файла*

*touch new\_folder/answer.txt // создать один файл в другой директории*

*mkdir new\_folder // создать папку внутри текущей директории*

*mkdir /var/log/temp // создать папку внутри по указанному пути*

*rm file1.txt // удалить файл*

*rm file1.txt file2.txt // удалить два файла*

*rm folder/file1.txt // удалить файл из директории*

*rm -r folder // удалить папку рекурсивно со всем содержимым*

*cp logs.txt double\_logs.txt // переименовать файл logs.txt в double\_logs.txt*

*cp ../logs/logs.txt double\_logs.txt // скопировать файл в текущую директорию и назвать по-новому*

*cp -r ../docs/ ../Documents/ // скопировать папку со всем содержимым*

*mv card.txt / // перенести файл из текущей директории в корневую*

*mv card.txt /home/logs/2020 // перенести файл из текущей директории в указанную*

*mv my\_app.ssh you\_app.sh // переименовать файл*

*echo “Ау!” // Вывод текста в терминал*

*echo “Testing” > secret.txt // записать в него строку в файл. Если файла нет – он создастся*

*cat 1.txt 2.txt 3.txt // вывод содержимого трех файлов*

*cat a.txt > b.txt // стереть файл b.txt и записать в него содержимое файла a.txt*

*cat a.txt >> b.txt // записать содержимое файла a.txt в конец файла b.txt*

*cat 1.txt 3.txt >> b.txt // записать содержимое обоих файлов 1.txt и 2.txt в конец файла b.txt*

Для редактирования текста внутри контейнера можно использовать ***nano*** редактор. Клавиша *Ctrl* в нем обозначена символом *^*.

Специальные сочетания клавиш можно увидеть, нажав *Ctrl+G* в запущенном редакторе.

Сохранить файл можно сочетанием клавиш *Ctrl+O*.

Выйти из редактора можно сочетанием клавиш *Ctrl+X*.

*apk add nano // установить редактор nano*

*nano <название или путь к файлу> // открыть файл в редакторе*

*grep <ключи> <искомое слово/фраза> <путь до файла> // поиск текста внутри файла*

*grep -i ERROR logs.txt > errors.txt // найти строки с “ERROR”, игнорируя регистр, и сохранить в файл*

Команда ***grep*** чувствительна к регистру искомого текста.

Ключ ***-i*** позволяет игнорировать регистр.

Ключ ***-n*** дополнительно отобразит номер строки.

Ключ ***-c*** вернет количество строк, которое соответствует условиям поиска.

Ключ ***-B*** (*--before-context*) задает количество отображаемых строк до искомой.

Ключ ***-A*** (*--after-context*) задает количество отображаемых строк после искомой.

Ключ ***-C*** (*--context*) задает количество отображаемых строк до и после искомой одновременно.

*tail <имя файла> // вывести последние 10 строк файла*

*tail -5 <имя файла> // вывести последние 5 строк файла*

*tail -f <имя файла> // вывести последние 10 строк файла в режиме непрерывного просмотра*

Ключ ***-f*** включает режим непрерывного просмотра. Все изменения будут отображаться автоматически.

*ps // вывод списка запущенных процессов*

*ps | grep nano // вывод списка запущенных процессов редактора nano*

*top // вывод списка запущенных процессов в реальном времени*

*top | grep nano // вывод списка запущенных процессов редактора nano в реальном времени*

*fuser -k 9999/tcp // остановить процесс, использующий порт 9999 по протоколу tcp*

*kill [ключ] [PID] // остановить процесс с указанным номером PID*

Ключ ***-9*** делает команду принудительной. Подробнее [тут](https://pubs.opengroup.org/onlinepubs/9699919799/utilities/kill.html#tag_20_64).

*chmod +x script.sh // добавить права для выполнения файла*

*ssh kbaturin:password:51.250.15.136 // подключиться через ssh*

*// проброс портов через ssh (при обращении к порту 8081 перенаправлять их на порт 8080)*

*ssh kbaturin:password:51.250.15.136 -N -L 8081:51.250.15.136:8080*

*// проброс портов через ssh и через промежуточный сервер*

*ssh kbaturin:password:51.250.15.136 -N -L 8081:51.250.15.136:8080 51.250.84.187*