|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **<https://youtu.be/O8N1lvkIjig?si=M4p8SXxCRR5Adno5>** | | | | | | | | | | | | | |
| **Что нужно знать разработчику про Docker:**   1. Из чего состоит Docker и как он устроен внутри? 2. Что такое image и контейнер и чем они отличаются? 3. Как писать Dockerfile? 4. Как использовать производный объект из Dockerfile? (image и контейнер) 5. Как запускать несколько Dockerfile, чтобы они работали вместе? (Docker Compose) | | | | | | | | | | | | | |
| **Что такое Docker?**  В 2008 году с целью построения [PaaS](https://ru.wikipedia.org/wiki/Платформа_как_услуга)-платформы (есть ещё [Iaas](https://ru.wikipedia.org/wiki/Инфраструктура_как_услуга)) для разных языков программирования стартанула разработка будущего Docker! Docker - это платформа для управления контейнерами и содержит все необходимое для создания, запуска и взаимодействия этих контейнеров! Docker упаковывает ПО в стандартизованные контейнеры. Разработчики при помощи Docker создают контейнер таким образом, чтобы он содержал минимально необходимый набор программ: библиотеки, системные инструменты, код и среду исполнения. | | | | | | | | | | | | | |
| **Из чего состоит Docker?**  Основные компоненты Docker   1. **Daemon.** Фоновая служба на хосте, которая отвечает за создание, запуск и уничтожение контейнеров. Любое взаимодействие с контейнером проходит через Daemon. 2. **Client.** Утилита CLI в Docker для управления демоном. 3. **Registry.** Служба в Docker, которая является репозиторием для хранения записей об образах и контейнерах! Устанавливается отдельно и обычно на серверах внутри компаний если нужно сделать локальное хранилище или в облаке если нужно сделать публичное хранилище! Это инструмент хранения и обмена docker образами. Его образ можно скачать и запустить с официального репозитория Docker Hub.   Примеры Registry:   * **Docker hub** — центральный реестр, используемый для загрузки docker-image. * **Azure registry** — контейнер с службой, который предназначен для работы с образами в директории Azure (Azure Active Directory). * **DTR** — служба docker-реестра для установки на локальном компьютере или сети компании.  1. **Dockerfile.** Обычный текстовый файл, в котором прописывают специальные инструкции для создания образа Docker. Файл создаётся по принципу «одна строка — одна команда». После специальной команды Daemon выполняет инструкции по порядку и собирает image 2. **Image.** Неизменяемый файл (образ), из которого можно неограниченное количество раз развернуть контейнер. 3. **Container.** Запущенное приложение, которое развернули из образа. 4. **Docker Desktop.** Приложение, позволяющее локально собирать, выполнять и тестировать контейнеры. Работает на Windows и macOS. 5. **Docker Volumes.** Тома для постоянного хранения информации. По умолчанию в Docker папки хранилищ создаются на хост-машине, но предусмотрена и возможность подключения удаленных хранилищ. Использование томов позволяет лучшим образом настроить хранение данных 6. **Docker Secrets** Специальные хранилища, которые шифруют всю конфиденциальную информацию и тем самым делая её недоступной для чтения с стороны, но при этом эта информация легко используется внутри контейнера! ВАЖНО: секрет нельзя изменить - его можно только создать снова с новыми значениями! 7. **Docker Network** Перечень всех сетей, которые доступны сейчас в пространстве у Daemon и в рамках которых контейнеры могут взаимодействовать друг с другом! | | | | | | | | | | | | | |
| **Как Docker использует** **UnionFS?**  [UnionFS](https://unionfs.filesystems.org/) - это файловая система, которая используется Docker для хранения файлов приложения! UnionFS создаёт иллюзию того, что в вашем image находятся все папки которые вы туда положили! Почему иллюзию?  Потому что по факту UnionFS обеспечивает хранение файлов в виде слоёв и эти слои используются разными image!! И каждый image думает что слой принадлежит только ему! Такой формат хранения данных позволяет кратно сократить место для хранения файлов, потому что один и тот же файл скачивается один раз и используется всеми!! С UnionFS началось развитие и сейчас она уже [не развивается](http://git.fsl.cs.sunysb.edu/?p=unionfs-2.6.39.y.git;a=summary), но принцип её остался и применяется в других реализациях (OverlayFS)! Если хотите разобраться в деталях, то нужно изучить драйверы и вам [сюда](https://docs.docker.com/storage/storagedriver/select-storage-driver/)!  **Вопрос: но как тогда файлы изменяются, если они используются всеми сразу?**  **Ответ:** скаченный слой с файлом НИКОГДА НЕ МЕНЯЕТСЯ!!! UnionFS обеспечивает доступ к слоям контейнера только для чтения и не позволяет менять слои! | | | | | | | | | | | | | |
| Тут мы подходим к вопросу **что такое image и что такое контейнер и чем они отличаются**? Как выполняется контейнеризация с помощью Docker?   * **image** - это набор файлов, которые помещены в слои! * **контейнер** - это тот же набор файлов, но сверху добавлен специальный слой для изменений, в котором и происходит все операции в приложении!   ВЫВОД: отличие в верхнем слое, который используется для всех операций ! | | |  | | | | | | | | | | |
| **Вопрос: если слои не меняются, то как тогда мы можем настраивать нашу JVM, наш postgres или Linux в нашем контейнере**?  Ответ: UnionFS создаёт верхний слой доступный для изменений и все что в нем можно менять! Если вы поработали с контейнером и потом из этого результата/контейнера захотели создать image с своими настройками то UnionFS создаст 2 отдельных слоя - один базовый postgres а второй верхний ваш сделает тоже неизменяемым и с вашими настройками и если вы потом из своего этого нового image запустите контейнер, то контейнер запуститься с новым верхнем слоем, но изменить предыдущий уже нельзя будет и так слои будут копиться и так image может разрастись невероятно !!!! а по факту вы будете использовать только предпоследний слой с настройками! | | | | | | | | | | | | | |
| **Вопрос с собеседования: В чем разница между 'docker start' и** **'docker run'?**  Ответ:   1. 'docker start' стартует УЖЕ СОЗДАННЫЙ контейнер и если его нет, то будет ошибка! 2. 'docker run' вначале из image создаст конетйнер (добавит верхний слой для изменений) и потом под капотом для контейнера выполнит 'docker start' для созданного контейнера! | | | | | docker run | | | | | | | | |
| **Вопрос: Чем отличается виртуальная машина от докера?** |  | | | | | | | | | | | | |
| |  |  |  |  | | --- | --- | --- | --- | |  | **виртуальные машины** |  | **контейнеры** | |  | **ПРЕИМУЩЕСТВА** | | | | 1 | **Независимость**. Изолированность каждой ВМ означает, что сбои и отказы в ОС или приложениях ВМ не повлияют на другие ВМ на том же компьютере или на других компьютерах в среде центра обработки данных. | 1 | **Размер**. Контейнеры используют одно общее ядро ОС и не используют собственные уникальные ОС, поэтому контейнеры являются гораздо меньшими логическими единицами, чем виртуальные машины. Это позволяет компьютеру одновременно размещать гораздо больше контейнеров, чем ВМ. | | 2 | **Использование ресурсов**. Поскольку несколько ВМ могут быть инициализированы и развернуты на одном физическом компьютере, система может эффективно размещать несколько рабочих нагрузок без необходимости покупать несколько компьютеров. Это позволяет консолидировать серверы в центре обработки данных. | 2 | **Время создание и развертывание**. Небольшой размер экземпляров контейнеров позволяет создавать и уничтожать их гораздо быстрее, чем ВМ. Благодаря этому контейнеры хорошо подходят для быстрого масштабирования и краткосрочного использования, что может быть нецелесообразно для ВМ. | | 3 | **Совместимость.** Переносимость виртуальных машин позволяет работать на разных Гипервизорах и тем самым облегчает поддержку если специалисты разные по уровню знаний! | 3 | **Уникальный гипервизор**. Для хостинга и управления контейнерами используются специализированные гипервизорные платформы, такие как Docker, rkt и Apache Mesos. | | 4 | **Гибкость**. Для каждой ВМ требуется своя ОС, но каждая ОС может быть разной. Это позволяет предприятию использовать несколько ОС на одном физическом компьютере. | 4 | **Неизменяемость**. В отличие от ВМ, контейнеры не изменяются. Вместо этого контейнерный оркестратор в программном слое контейнера запускает и останавливает контейнеры, когда они необходимы. Аналогичным образом, программное обеспечение, работающее в контейнерах, не обновляется, как традиционное ПО. Вместо этого обновления включаются в новый образ контейнера, который может быть развернут там, где это необходимо. | | 5 | **Безопасность**. Гипервизоры и логическая изоляция, которую они обеспечивают, доказали свою безопасность для виртуальных машин. Даже если одна ВМ может быть заражена, это не приведет к заражению других ВМ. | 5 | **Лицензирование программного обеспечения.** Контейнеры создаются на базе Linux и открытого ПО и не так зарегулированы лицензиями как операционные системы и виртуальные машины! | |  | **НЕДОСТАТКИ** | | | | 1 | **Размер**. Экземпляры ВМ могут быть большими, с использованием нескольких процессоров и значительного объема памяти. Это хорошо подходит для рабочих нагрузок корпоративного класса, но на практике существует ограничение на количество виртуальных машин, которые можно развернуть на одном компьютере. | 1 | **Производительность**. Контейнеров может быть много, и контейнеры используют общую ОС в дополнение к контейнерному слою. Это означает, что контейнеры эффективно используют ресурсы, но между контейнерами могут возникать разногласия при попытке получить доступ к аппаратным ресурсам, таким как сети. Такое соперничество может повлиять на общую производительность контейнера. | | 2 | **Время создание и развертывание** виртуальных машин может занимать от нескольких секунд до нескольких минут. Хотя это не так много времени с человеческой точки зрения, ВМ могут недостаточно быстро масштабироваться для удовлетворения динамических или краткосрочных вычислительных потребностей. | 2 | **Совместимость**. Контейнеры, упакованные для одной платформы, такие как Docker, могут не работать с другими платформами. Аналогично, некоторые контейнерные инструменты могут не работать с различными контейнерными платформами. Например, Red Hat OpenShift работает только с оркестратором Kubernetes (по сути Red Hat OpenShift это настроенный Kubernetes). Учитывайте экосистему контейнеров при оценке контейнерных технологий для предприятия. | | 3 | **Лицензирование программного обеспечения**. Каждая ВМ нуждается в ОС и рабочей нагрузке, поэтому стоимость лицензирования ОС и приложений может стать значительной. Предприятие должно внимательно управлять развертыванием ВМ для учета лицензий и обеспечения того, чтобы ВМ с дорогостоящими лицензиями ОС и приложений работали и выполняли продуктивную работу для предприятия, избегая разрастания ВМ. | 3 | **Хранение**. Контейнеры изначально спроектированы так, чтобы быть без статических данных — данные в контейнере исчезают, когда исчезает контейнер. Существуют способы обеспечения постоянного хранения для контейнеров, например, Docker Data Volumes, но вопрос постоянного хранения контейнеров часто рассматривается как нечто отдельное. | |  |  | 4 | **Пригодность**. Контейнеры, как правило, представляют собой небольшие и гибкие структуры, лучше всего подходящие для компонентов приложений или микросервисов. Полнофункциональные корпоративные приложения обычно плохо работают в контейнерах. Учитывайте пригодность при планировании развертывания контейнеров. | |  |  | 5 | **Безопасность.** Если получить доступ к одному контейнеру, то легче получить доступ к всему Docker и другим контейнерам - это очень большие риски! | |  |  | 6 | **Независимость**. Так как каждый ПО присутствует в слое , а слои используются разными контейнерами, то баг безопасности или баг приложения в одном слое сразу отразиться на всех контейнерах, которые используют этот слой! | | | | | | | | | | | | | | |
| **Что такое Dockerfile? Опишите процесс создания образа Docker из Dockerfile, его сохранения в реестре и запуска контейнера на основе этого образа.**  Dockerfile - обычный текстовый файл, в котором прописывают специальные инструкции для создания образа Docker.   1. Файл создаётся по принципу «одна строка — одна команда». 2. При помощи инструкций пишем Dockerfile! Сохраняем Dockerfile без расширения! 3. После написания Dockerfile нужно выполнить команду docker build -t имя\_image . , чтобы создать image из Dockerfile | | | | | | | | | | | | | |
| [**Перечень команд**](https://docs.docker.com/engine/reference/builder/) **для написания Dockerfile** | | | | | | | | | | | | | |
| 1. **FROM** — задаёт базовый (родительский) образ. Dockerfile обязательно должен иметь FROM.    1. **одноэтапная сборка** - FROM должен быть первой инструкцией    2. **многоэтапная сборка -** Каждая FROM использует свои базовый образ и каждая начинает новую стадию сборки образа. Многоэтапная нужна, чтобы копировать артефакты из одной стадии в другую. | | | | | | |  | | | | | | |
| 1. **LABEL** — описывает метаданные. Например — сведения о том, кто создал и поддерживает образ. Инструкция LABEL (метка) позволяет добавлять в образ метаданные. В случае с рассматриваемым сейчас файлом, она включает в себя контактные сведения создателя образа. Объявление меток не замедляет процесс сборки образа и не увеличивает его размер. Они лишь содержат в себе полезную информацию об образе Docker, поэтому их рекомендуется включать в файл. | | | | | | |  | | | | | | |
| 1. **ENV** — устанавливает постоянные переменные, которые будут доступны в контейнере. ENV хорошо подходит для задания констант. Разработчики используют ENV для настройки приложения! | | | | | | |  | | | | | | |
| 1. **ARG** —позволяет задать переменную, значение которой можно передать при команде docker run с помощью флага --build-arg <varname>=<value>. Порядок следующий:   В Dockerfile мы указываем ARG и значение по умолчанию  а при команде docker run и флага --build-arg <varname>=<value> меняем значение  Отличие ARG от ENV-переменных, ARG-переменные недоступны во время выполнения контейнера, а ENV доступны! Однако ARG-переменные можно использовать для задания значений по умолчанию для ENV-переменных из командной строки в процессе сборки образа. А ENV-переменные уже будут доступны в контейнере во время его выполнения. | | | | | | |  | | | | | | |
| 1. **WORKDIR** —позволяет создать директорию или изменить рабочую директорию контейнера. С этой директорией работают инструкции COPY, ADD, RUN, CMD и ENTRYPOINT, идущие за WORKDIR. Некоторые особенности:    1. Лучше устанавливать с помощью WORKDIR абсолютные пути к папкам, а не перемещаться по файловой системе с помощью команд cd в Dockerfile.    2. Инструкция WORKDIR автоматически создаёт директорию в том случае, если она не существует.    3. Можно использовать несколько инструкций WORKDIR. Если таким инструкциям предоставляются относительные пути, то каждая из них меняет текущую рабочую директорию. | | | | | | | **можно использовать с ENV** | | | | | | |
| 1. **RUN** — выполняет команду и !!!!создаёт слой образа!!!!!. RUN используется для установки в контейнер пакетов. Инструкция RUN и схожие с ней инструкции — такие, как CMD и ENTRYPOINT, могут быть использованы:    1. **либо в** [**exec-форме**](https://docs.docker.com/engine/reference/builder/#exec-form)**,** (все элементы в квадратных скобках и внутри в кавычках - ENTRYPOINT ["/bin/bash", "-c", "echo", "hello"]) Если этот вариант, то вы должны написать команду в формате JSON массива и указать **полный путь** к исполняемому файлу. Этот формат является предпочтительным для CMD и RUN.    2. **либо в** [**shell-форме**](https://docs.docker.com/engine/reference/builder/#shell-form)**.** (команда пишется как для командной оболочки - RUN echo hello) Если этот вариант, то будет выполнена в /bin/sh -c для Linux   Exec-форма использует синтаксис, напоминающий описание JSON-массива. Например, это может выглядеть так: RUN ["my\_executable", "my\_first\_param1", "my\_second\_param2"]. Использована exec-форма инструкции RUN, в виде RUN ["mkdir", "/a\_directory"]. нужно помнить о необходимости оформления строк с помощью двойных кавычек, как это принято в формате JSON.  Как говорилось ранее RUN используют для установки списка пакетов. Если ставятся пакеты, то вместе с ним создаётся мусор, который нужно удалять!! Если использовать несколько подряд команд RUN то каждая будет создавать слой, который нельзя будет удалить и следвоательно нельзя удалить муссор из предыдущего слоя! Чтобы избежать этого объедините все команды в одну инструкцию и разделите её символами разрыва строки с помощью символа \. Благодаря такому подходу файлы будут выглядеть аккуратно и это приведёт к добавлению в образ меньшего количества слоёв, чем было бы добавлено при использовании нескольких инструкций RUN.  Кроме того, для установки нескольких пакетов можно поступить и по-другому. Их можно перечислить в файле и передать менеджеру пакетов этот файл с помощью RUN. Обычно таким файлам дают имя requirements.txt. | | | | | | | | | | | | | |
| 1. **COPY**— копирует в контейнер файлы и папки. Она сообщает Docker о том, что нужно взять файлы и папки из локального контекста сборки и добавить их в текущую рабочую директорию образа. Если целевая директория не существует, эта инструкция её создаст.   **ВАЖНО:** квадратные скобки для путей с пробелами! | | | | | | |  | | | | | | |
| 1. **ADD** — инструкция позволяет решать те же задачи, что и COPY, но с ней связана ещё пара вариантов использования:    1. можно добавлять в контейнер файлы,    2. загруженные из удалённых источников,    3. распаковывать локальные .tar-файлы. | | | | | | |  | | | | | | |
| 1. **CMD** — описывает команду с аргументами, которую нужно выполнить когда контейнер будет запущен. **ВАЖНО:** аргументы могут быть переопределены при запуске контейнера.    1. В одном файле Dockerfile может присутствовать лишь одна инструкция CMD. Если в файле есть несколько таких инструкций, система проигнорирует все кроме последней.    2. Инструкция CMD может иметь exec-форму. **НЮАНС:** Если в CMD инструкцию не входит упоминание исполняемого файла, тогда в файле должна присутствовать инструкция ENTRYPOINT с исполняемым файлом. И если указаны CMD и ENTRYPOINT, то обе инструкции должны быть представлены в формате JSON.    3. Аргументы командной строки, которые передаются при команде docker run, переопределяют аргументы инструкции CMD, которая указана в Dockerfile. | | | | | | |  | | | | | | |
| 1. **ENTRYPOINT** — предоставляет команду с аргументами для вызова во время выполнения контейнера. Аргументы не переопределяются. ENTRYPOINT похожа на команду CMD, но параметры, задаваемые в ENTRYPOINT, не перезаписываются в том случае, если контейнер запускают с параметрами командной строки. Аргументы командной строки, передаваемые в конструкции вида docker run my\_image\_name, добавляются к аргументам, задаваемым инструкцией ENTRYPOINT. Например, после выполнения команды вида docker run my\_image bash аргумент bash добавится в конец списка аргументов, заданных с помощью ENTRYPOINT.   Пояснения о том какую инструкцию (CMD или ENTRYPOINT) выбирать при написании Dockerfile:   * 1. В Dockerfile должна быть хотя бы одна из команд - CMD или ENTRYPOINT.   2. Если при каждом запуске контейнера нужно выполнять одну и ту же команду — используйте ENTRYPOINT.   3. Если контейнер будет использоваться в роли приложения — используйте ENTRYPOINT.   4. Если вы знаете, что при запуске контейнера вам понадобится передавать ему аргументы, используйте CMD - тогда передаваемый параметр (команда или набор команд) затрет CMD | | | | | | |  | | | | | | |
| 1. **EXPOSE** — указывает, какие порты планируется открыть для того, чтобы через них связаться с контейнером. Эта инструкция не открывает порты!!!!!! Она, скорее, играет роль документации к образу, средством общения того, кто собирает образ, и того, кто запускает контейнер. Для того чтобы открыть порт (или порты) и настроить перенаправление портов, нужно выполнить команду docker run с ключом -p. Если использовать ключ в виде -P (с заглавной буквой P), то открыты будут все порты, указанные в инструкции EXPOSE. | | | | | | |  | | | | | | |
| 1. **VOLUME** — создаёт точку монтирования для работы с постоянным хранилищем.  VOLUME позволяет указать место, которое контейнер будет использовать для постоянного хранения файлов и для работы с такими файлами. Если удалить контейнер, то VOLUME все равно останется и значит останутся ваши файлы в нем - например БД! Из другого нового контейнера можно обратиться по имени к VOLUME и работать с данными как ни в чём не бывало!   В примере команда RUN создаёт файл greeting в каталоге /myvol а потом командой VOLUME делает этот каталог VOLUME и Docker монтирует весь внутренний каталог /myvol контейнера в volume с именем /myvol и теперь все данные из каталога /myvol внутри контейнера будут храниться в volume с именем /myvol внутри Docker и если убить контейнер то volume с именем /myvol будет доступен, потому что он создаётся в самом Dcoker (вне контейнера) и доступен для других контейнеров по имени /myvol | | | | | | |  | | | | | | |
| 1. **USER** — Определяет пользователя по умолчанию, все команды будут запускаться от его имени. Может быть либо UID, либо имя пользователя. | | | | | | |  | | | | | | |
| 1. **HEALTHCHEACK** [OPTIONS] CMD command – Команда HEALTHCHECK указывает Docker как проверить работает ли контейнер. Данная функция может помочь в выявлении ситуаций когда веб-сервер вошел в бесконечный цикл и не принимает соединения! например   HEALTHCHECK --interval=5s --timeout=10s --retries=3 CMD curl -sS 127.0.0.1 || exit 1  HEALTHCHECK будет запускаться самим Docker!!!!!!!, Docker будет обращаться к серверу по 127.0.0.1, раз в 5 секунд будет отправлять запрос на localhost, ждать максимум 10 секунд, чтобы localhost ответил, и если 3 раза подряд localhost  не ответит — будем считать что контейнер умер или неисправен, поэтому exit 1! | | | | | | |  | | | | | | |
| 1. **ONBUILD -** Инструкция ONBUILD добавляет триггерную инструкцию, которая выполняется в последнюю очередь если образ используется в качестве базового для другой сборки, например, когда исходный код, нужный для образа, еще не доступен, но требует для работы конкретного окружения! | | | | | | |  | | | | | | |
| **Популярные рекомендации для создания image и контейнеров:**   1. Соблюдайте принцип 1 приложение = 1 контейнер. 2. Запускайте процесс на переднем плане (не используйте systemd, upstart или другие похожие инструменты). 3. Для хранения данных вне контейнера используйте тома. 4. Не используйте SSH (если вам надо внутрь контейнера, используйте docker exec). 5. Избегайте ручных настроек или действий внутри контейнера. 6. Включайте только необходимый контекст — используйте файл .dockerignore  (как .gitignore в git). 7. Избегайте установки ненужных пакетов — это займет лишнее дисковое пространство, от которого не избавиться! 8. Всегда добавляйте контекст, который часто меняется, например, исходный код вашего проекта, в конец Dockerfile — кэш Docker будет использоваться более эффективно - Docker будет хранить меньше данных в части связи всех слоёв с вашими изменениями. 9. Будьте осторожны с томами (Volume). Вы должны помнить, какие данные находятся в томах. Поскольку тома постоянны!!!! и не исчезают вместе с контейнерами 10. Используйте RUN, EXPOSE, VOLUME и ENV. Это сделает ваш Dockerfile более гибким. | | | | | | | | | | | | | |
| **Как запустить контейнер? Основные параметры команды run**  docker run имя\_image  **ВАЖНО:** если контейнер ничего не делает, то docker его остановит - это [принцип работы docker](https://stackoverflow.com/questions/52955987/docker-container-keeps-stopping-after-docker-start): если контейнер не нужен он будет остановлен! Например у образа ubuntu точка входа (entrypoint или CMD) заключается в вызове bin/bash, которая не содержит команд к выполнению или что-то запускает для выполнения и поэтому контейнер сразу остановиться! Если точка входа запускает процесс, то контейнер остановиться когда задача завершиться! Если задача в бесконечном цикле то контейнер никогда не остановиться!  Опции команды:   1. **--detach, -d** - по умолчанию, docker-контейнер запускается присоединенным (attached) к стандартным потокам ввода-вывода. Параметр -d, --detach позволяет запускать контейнер в фоне и не выводить на экран содержимое потоков ввода-вывода. 2. **--entrypoint** - устанавливает или переопределяет используемую по умолчанию команду (и параметры) из docker-образа. Подробнее о Docker ENTRYPOINT можно узнать [здесь](https://docs.docker.com/glossary/?term=ENTRYPOINT).   docker run --entrypoint {{command}} {{image}}   1. **--env, -e** - устанавливает переменные окружения в формате пар КЛЮЧ=ЗНАЧЕНИЕ. Если у вас есть список переменных окружения в файле, тогда необходимо использовать опцию --env-file. BASE=asdf — в контейнере будет создана переменная окружения с именем BASE и значением asdf   docker run -e '{{variable}}={{value}}' -e {{variable}} {{image}} {{command}}   1. **--ip** - назначает docker-контейнеру IP-адрес, например, --ip=10.10.9.75. 2. **--name** - назначает docker-контейнеру имя (--name my-super-container). 3. **--hostname=value OR docker run -h value** - значение, поставляемое командой –hostname, указывается в файлах контейнера /etc/hostname и /etc/hosts. В дальнейшем оно появляется в командной строке. 4. **--publish, -p** или **--publish-all, -P** - сопоставляет порт(ы) docker-контейнера порту(ам) на хост системе. Стоит отметить, что --publish-all откроет доступ к портам, описанным в Dockerfile с помощью директивы EXPOSE.   docker run -p {{host\_port}}:{{container\_port}} {{image}} {{command}}   1. **--rm** - автоматически удалить контейнер после завершения его работы (в том числе, при завершении с ошибкой). 2. **--tty, -t** - назначает псевдо-TTY, подключенный к STDIN контейнера. Зачастую используется с опцией i 3. --**interactive, -i** - например, если вам нужно подключиться к оболочке (bash) внутри docker-контейнера для выполнения каких-то действий. 4. **--volume, -v** - монтирование тома. Подробнее о томах можно почитать [здесь](https://docs.docker.com/engine/reference/commandline/run/#mount-volume--v---read-only).   docker run -v {{path/to/host\_path}}:{{path/to/container\_path}} {{image}} {{command}}  По умолчанию используется опция -rw (чтение/запись). Можно также настроить права только на чтение, установив опцию -ro  Тома из других контейнеров: (--volumes-from)  *docker run --volumes-from 777f7dc92da7 --volumes-from ba8c0c54f0f2:ro -i -t ubuntu pwd*  Флаг --volumes-from монтирует все определенные тома из указанных контейнеров. Контейнеры могут быть указаны повторением аргумента --volumes-from. Идентификатор контейнера может дополнительно иметь суффикс :**ro или :rw** для монтирования томов в режиме только для чтения или чтения-записи соответственно. По умолчанию тома монтируются в том же режиме (чтение-запись или только чтение), что и эталонный контейнер.   1. **--workdir, -w** - устанавливает рабочую директорию внутри docker-контейнера. Например, если вы скопировали некие файлы в папку app контейнера, то вполне разумно будет установить этот каталог в качестве рабочей директории. 2. **--network** Запускаем новый контейнер и присоединяем к указанной сети   docker run --network {{network}} {{image}} (docker run -itd --network=my-net --ip=10.10.9.75 busybox)   1. **--memory** , **-m** установить лимит по памяти 2. **--restart**=always -d {{ name }}  * no – не перезапускать docker-контейнер (по умолчанию); * on-failure – рестартануть контейнер, в случае если entry point процесс завершился с ненулевым кодом (docker run --restart=on-failure:10 {{ name }} -Docker попытается только 10 раз. Если контейнер поднимется хотя бы и завершится с нулевым кодом возврата, то счётчик начнётся с начала. * always – перезапускать всегда, вне зависимости от того, что случилось с docker-контейнером.  1. **--label , -l** – установить метаданные (docker run -l my-label --label com.example.foo=bar ubuntu bash) | | | | | | | | | | | | | |
| **Какие важные команды требуется знать разработчику:**   * **exec** * **logs** | | | | | | | | | | | | | |
| **docker exec name\_container команда с параметрами**  Если хотим зайти внутрь то нужно попросить TTY - указать параметр -it  Чтобы отключиться ctrl + p и ctrl + q | | |  | | | | | | | | | | |
| **docker logs name\_container** | | |  | | | | | | | | | | |
| **Другие важные команды для повторения:**  **Информация**   1. **docker info**- Информация обо всём в установленном Docker 2. **docker history** - История образа 3. **docker tag** - Дать тег образу локально или в registry 4. docker login - Залогиниться в registry 5. docker search - Поиск образа в registry 6. **docker pull** - Загрузить образ из Registry себе на хост 7. **docker push** - Отправить локальный образ в registry   **Управления контейнерами**   1. **docker ps -а** - Посмотреть все контейнеры 2. docker start container-name - Запустить контейнер 3. docker kill/stop container-name - Убить (SIGKILL) /Остановить (SIGTERM) контейнер 4. **docker logs --tail 100 container-name** - Вывести логи контейнера, последние 100 строк 5. **docker inspect container-name** - Вся инфа о контейнере + IP 6. **docker rm container-name** - Удалить контейнер (поле каждой сборки Dockerfile) 7. **docker rm -f $(docker ps -aq)** - Удалить все запущенные и остановленные контейнеры 8. **docker events container-name** - показать список активностей в процессе работы сервиса Docker. Вот некоторые из них: подключение к консоли контейнера, commit, переименование, удаление и т.п., а также есть возможность фильтрации нужных событий. 9. **docker stats container-name** - Статистика использования ресурсов в контейнере: использование оперативной памяти и процессорного времени в контейнере. 10. **docker port container-name** - Показать публичный порт контейнера 11. **docker top container-name** - Отобразить процессы в контейнере 12. **docker diff container-name** - Изменения в ФС контейнера - перечислит все изменения, внесенные в файлы и каталоги внутри определенного контейнера. Каждая строка вывода docker diff имеет префикс A, C или D в зависимости от типа произошедшего изменения:  * A – Файл или каталог был добавлен в файловую систему контейнера. Он не существовал в исходном образе. * C – Содержимое файла или каталога отличается от версии в образе. * D – Путь, присутствующий в образе, был удален из контейнера.  1. **docker run --name=имя \_контейнера name \_image** – запустить с присвоением имени (имя не будет отображаться в командной строке или в лог-файлах контейнера. Для этого нужно также присвоить имя хоста) 2. **docker run --hostname=value OR docker run -h value** - значение, поставляемое командой –hostname, указывается в файлах контейнера /etc/hostname и /etc/hosts. В дальнейшем оно появляется в командной строке. Это важно для настройки DNS и определения уровней установок, состоящих из нескольких контейнеров. Имя хоста появится в лог-файлах контейнера; в дальнейшем эти файлы записываются в независимые тома, где с помощью имени хоста будет проще идентифицировать контейне 3. **docker container attach** - прикрепление терминала к корневому контейнеру. Осоединить контейнер, нажав ctrl + p и ctrl + q один за другим. Контейнер может отказывается отсоединяться, даже когда вы выполняете последовательность клавиш. Это может произойти, если входной поток контейнера не подключен (не использовался -it) к вашему терминалу (-i flag) или ему не выделен псевдо-TTY (-t флаг). 4. **docker update (параметры) container -** можно использовать для изменения ограничений ресурсов, применяемых к контейнерам. Вы должны передать один или несколько идентификаторов или имен контейнеров вместе со списком флагов, определяющих ограничения, устанавливаемые для этих контейнеров. **docker update** -cpus 4 -memory 1024M **container1 container2 container3** Аппаратные ограничения, политики ресурсов и имена контейнеров — единственные параметры конфигурации, которые Docker CLI позволяет вам изменить. Вы не можете изменить образ работающего контейнера; вы также не можете легко изменить другие параметры, такие как привязки портов и тома. Вам следует создать другой контейнер, если эти значения устарели. Уничтожьте текущий экземпляр и используйте docker run, чтобы начать замену новым образом и скорректированными настройками.   **Управления образами**   1. **docker build -t my\_app .** - Билд контейнера в текущей папке, Скачивает все слои для запуска образа 2. docker images / docker image ls - Показать все образы в системе 3. **docker image rm** / docker rmi image - Удалить image 4. **docker commit  <containerName/ID>** - Создает образ из контейнера 5. docker insert URL - вставляет файл из URL в контейнер 6. **docker save -o backup.tar** - Сохранить образ в backup.tar в STDOUT с тегами, версиями, слоями 7. docker load - Загрузить образ в .tar в STDIN с тегами, версиями, слоями 8. docker import - Создать образ из .tar 9. **docker image history --no-trunc** - Посмотреть историю слоёв образа 10. **docker image prune -** Для удаления всех неиспользуемых образов 11. docker system prune -f - Удалит все, кроме используемого (лучше не использовать на проде, ещё кстати из-за старого кеша может собираться cтарая версия контейнера),вщслук   **Управления разделами (Volumes)**   1. **docker cp file <containerID>:/** - Скопировать в корень контейнера file 2. **docker cp <containerID>:/file** . - Скопировать file из корня контейнера в текущую директорию командной строки 3. **docker volume create todo-db** - Создать volume для постоянного хранения файлов   docker run -dp 3000:3000 --name=dev -v todo-db:/etc/todos container-name - Добавить named volumу todo-db к контейнеру (они ok когда мы не заморачиваемся где конкретно хранить данные)  docker run -dp 3000:3000 --name=dev --mount source=todo-db,target=/etc/todos container-name - тоже самое что команда сверху   1. **docker volume ls** - Отобразить список всех volume’ов 2. **docker volume inspect** - Инспекция volume’ов 3. **docker volume rm** - Удалить volume    1. В случае параметра «-v» указывается два пути «откуда:куда».    2. В случае «--mount» это именованные параметры разделенные запятыми. В mount мы используем следующие параметры:       1. **type**- со значением 'bind' говорит, что мы монтируем папку или файл хоста;       2. **source** - источник т.е. папка или файл, который мы хотим подключить к контейнеру;       3. **destination**- папка или файл внутри контейнера.   docker run -d --name nginx\_vol2 --mount type=bind,source=/home/alex/docker\_data,destination=/usr/share/nginx/html nginx (пробелов между запятыми нет)  **Управления сетью (Network)**   1. **docker network create todo-app** - Создать сеть 2. **docker network rm** - Удалить сеть 3. **docker network ls** - Отразить все сеть 4. **docker network inspect** - Вся информация о сети 5. **docker network connect** - Соединиться с сетью (docker network connect bridge SERVER4) 6. **docker network disconnect** - Отсоединиться от сети (docker network disconnect bridge SERVER4)   **Управления плагинами**   1. **docker plugin install <org/image>** - установить плагин 2. docker plugin ls – посмотреть список плагинов 3. **docker plugin inspect <org/image>** - отображение подробной информации об одном или нескольких плагинах   *docker plugin inspect tiborvass/sample-volume-plugin:latest*   1. docker plugin rm <org/image> – удаление плагина 2. **docker plugin set <org/image>** - Изменить настройки для плагина 3. docker plugin upgrade <org/image> - Обновите существующий плагин   **Управления секретами**  Для того, чтобы добавить к работающему контейнеру или обновить секрет у сервиса, его придется пересоздать**.**   1. **docker secret create name file** – создать секрет из файла file   *docker secret create db-password password.txt*  создаст «секрет» с именем db-password, используя содержимое файла password.txt   1. **docker secret ls** – просмотреть список секретов 2. **Добавление секрета происходит через создание сервиса с опцией –secret name\_secret**   *docker service create --name myservice -p 5000:5000 --secret mysecret name\_image* | | | | | | | | | | | | | |
| **Что такое** **Docker Compose?** Какие разделы присутствуют в Docker Compose?  Какие основные команды в Docker Compose?  Чтобы создать и запустить контейнер, требуется указать docker run и множество параметров, при этом требуется создать заранее общие volume, secrets и network. чтобы к ним подключиться и использовать - все это хлопотно и требует строгой очерёдности действий, а в случае ошибки или сбоя требует перезапуска всех элементов! Чтобы облегчить работу с множеством элементов, придумали инструмент оркестрации Docker Compose!  Docker Compose - это инструмент для настройки и запуска многоконтейнерных приложений! Docker Compose входит в Docker по умолчанию! | | |  | | | | | | | | | | |
| Docker Compose создаётся в файле и в отличии от dockerfile имеет расширение yml! Чтобы запустить созданный файл, нужно вызывать утилиту docker сompose и потом команду:   1. build Создать или пересоздать сервисы 2. bundle Создать пакет Docker из файла Compose 3. config Проверка и просмотр файла Compose 4. create Создать сервис 5. **up** Создание и запуск контейнеров 6. **down** Остановить и удалить контейнеры, сети, образы и тома 7. events Получать события в реальном времени от контейнеров 8. exec Выполнить команду в работающем контейнере 9. help Получить помощь по команде 10. images Список образов 11. kill Немедленно убить контейнер 12. **logs** Просмотреть логи контейнеров 13. pause Пауза сервисов 14. port Распечатать общедоступный порт для привязки порта 15. ps Список контейнеров 16. pull Загрузить образы сервисов 17. push Выгрузить образы сервисов 18. restart Рестарт сервисов 19. rm Удалить остановленные контейнеры 20. run Запустить одноразовую команду 21. scale Установить номер (может количество) контейнеров для сервисов 22. start Старт сервисов 23. **stop** Стоп сервисов 24. top Отобразить запущенные процессы 25. unpause Снять с паузы сервисы - запсутить 26. version Показать версию Docker Compose | | |  | | | | | | | | | | |
| **Docker Compose могут быть следующие элементы верхнего уровня:**   1. [version](https://github.com/compose-spec/compose-spec/blob/master/spec.md#version-top-level-element) : информация о версии файла конфигурации 2. [services](https://github.com/compose-spec/compose-spec/blob/master/spec.md#services-top-level-element) (обязательный): список всех контейнеров, которые нужно будет запустить; 3. [networks](https://github.com/compose-spec/compose-spec/blob/master/spec.md#networks-top-level-element): список подсетей Docker Network, которые используют контейнеры в виртуальной локальной сети (она может быть доступна из внешнего мира); 4. [volumes](https://github.com/compose-spec/compose-spec/blob/master/spec.md#volumes-top-level-element): список томов, которые будут доступны контейнерам, 5. [configs](https://github.com/compose-spec/compose-spec/blob/master/spec.md#configs-top-level-element): список параметров, которые позволяют запускать контейнеры в различных режимах, не собирая их заново; 6. [secrets](https://github.com/compose-spec/compose-spec/blob/master/spec.md#secrets-top-level-element): список чувствительных с точки зрения безопасности параметров (то же, что и configs, но специального назначения). | | | | | | | | | | | | | |
| Начинать надо всегда с раздела services:   1. указываем имя сервиса - по этому имени docker будет перенаправлять сетевые пакеты и если IP адрес у сервиса поменяется, то сервисы В РАМКАХ ОДНОЙ СЕТИ все равно друг друга увидят | | | | | | | | | | | | | |
| 1. указываем из чего будет создан наш сервис и тут 2 враианта    1. из dockerfile - тогда используем поле build: - указывается путь к папке, где лежит dockerfile    2. из готового image - тогда указываем image:, который доступен в нашем реестре. если в реестре нет, то docker сходит на docker hub за образом | | |  | | | | | | |  | | | |
| 1. если мы захотим переопределить CMD команду из dockerfile или образа, то нужно использовать **command:** - тогда указанная команда из файла docker compose переопределит CMD команду из dockerfile или image 2. если мы захотим изменить ENTRYPOINT команду из dockerfile или образа, то нужно указывать команду **entrypoint:**   ВАЖНО: команды можно указывать в shell варианте или exec! В exec команда записывается как массив слов по синтаксису YAML | | | в shell формате:    в exec формате: | | | | | | |  | | | |
| 1. мы захотим открытые порты и поэтому указываем expose: Важно, что по факту эти порты уже в image должны быть открыты и expose: это все го лишь документирование и порты она не откроет! Можно указать только внутренние порты контейнера. | | | | | | | | | | | | |  |
| 1. мы захотим настроить (мапить) наш порт нашей локальной машины на внутренний порт контейнера и тогда используем ports: - тут есть несколько вариантов записи! Порт может быть либо одним значением, либо диапазоном. Хост и контейнер ДОЛЖНЫ использовать эквивалентные диапазоны.   Либо укажите оба порта ( HOST:CONTAINER), либо только порт контейнера. В последнем случае реализация Compose автоматически выделит любой неназначенный хост-порт вашей машины для связи с контейнером. | | | | | | | |  | | | | | |
| 1. нам потребуется настраивать наш сервис и тут мы будем использовать    1. environment: - если хотим внутренний переменные    2. env\_file: - если хотим использовать внешний файл с переменными | | | |  | | | | | | |  | | |
| 1. чтобы использовать не только переменные, но и целые конфигурации (например конфигурации для nginx в виде абзацев настроек), то нужно использовать configs: но перед указанием в services нужно прописать configs в глобальном разделе configs текущего compose файла:  * создаем имя my\_config * для имени указываем что конфигурация будет из файла * для второй конфигурации my\_other\_config указан флаг external: true - это значит, что конфигурация должна быть создана в пространстве Docker (через команду docker config create …….) Если в пространстве не будет указанного config то запус будет с ошибкой! | | | | | | | | |  | | | | |
| 1. мы захотим подключить volume, например в которых будет БД, чтобы к ней обращаться из нашего сервиса и тут мы используем volume: Необязательными параметрами являются:    1. **:ro** - read only контейнер доступен только для чтения    2. **:rw** - read - write контейнер доступен для чтения и записи | |  | | | | | | | | | | | |
| 1. нам потребуется указать зависимость загрузки наших сервисов при старте и тут потребуется depends\_on   Compose гарантирует запуск зависимых служб до запуска зависимой службы. Compose ждут, пока службы зависимостей будут «готовы»  Синтаксис длинной формы позволяет настраивать дополнительные поля:  **condition:** условие, при котором текущий сервис стартанет   * **service\_started**: запустит текущий сервис когда указанный стартанет * **service\_healthy**: запустит текущий сервис только если зависимый будет здоровым - используют вместе с Healthcheck * **service\_completed\_successfully**: запустит текущий когда зависимый завершит своё выполнение и при этом без ошибок | | | | | |  | | | | | | | |
| 1. конечно наш сервис будет падать и нужно чтобы он стартовал сам или не стартовал, чтобы мы увидели ошибку работы сервиса и тут используют поле restart:, у которого есть несколько режимов: 2. **no**: политика перезапуска по умолчанию. Ни при каких обстоятельствах не перезапускает контейнер. 3. **always**: политика всегда перезапускает контейнер до его удаления. 4. **on-failure**: политика перезапускает контейнер, если код выхода указывает на ошибку. 5. **unless-stopped**: политика перезапускает контейнер независимо от кода выхода, но прекращает перезапуск при остановке или удалении службы. | | | | | | | | | | | |  | |
| 1. мы можем присвоить имена каждому контейнеру через container\_name: | | | | | | | | | | | |  | |
| 1. мы захотим ограничить используемые ресурсы для каждого сервиса, чтобы протестировать как сервис будет работать в ограниченных ресурсах и делается это при помощи блока deploy:, в котором указываются resources: в двух вариантах:    1. limits: - указывает сколько максимально service сможет сожрать    2. reservation: - указывает сколько под сервис зарезервировано на машине   В каждом разделе нужно указывать cpus и memory !!! Важно такие параметры доступны только в версии 3, у ранних версий команды отличаются | | | | | | | | | | | |  | |
| 1. возможно мы захотим настроить несколько реплик нашего сервиса, тогда в раздел deploy: нужно добавить ещё поле replicas: и указать сколько реплик сервиса нужно запустить! Эта функция появилась в версии 3.8 если не изменяет память))) | | | | | | | | | | | |  | |

**Осталось разобрать network и secrets:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| В Docker compose при старте создаёт сеть с именем default : |  | | | | |
| Управление сетями в Docker compose очень глубокая тема, так как настройки сети могут содержать множество параметров, поэтому мы рассмотрим простые варианты для ознакомления: | | | | | |
| 1. создание сети по умолчанию - тогда вообще не нужно указывать раздел networks: и соответственно указывать имена сетей в services: - docker все сервисы поместит в одну сеть! Такой вариант у нас в проекте в файле docker-compose.yml Docker compose создаст сеть с именем default: | | | | | |
| **ВАЖНО:** с версии 2 сеть по умолчанию имеет свойство attachable:true - это значит, что контейнеры вне данного файла могут присоединяться к текущей сети ! Параметр можно отключить на false! Раньше приходилось задавать параметр в явном виде как на картинке! | | | | |  |
| Если мы хотим сделать сеть изолированной от всех, то нужно указать параметр internal: true | |  | | | |
| 1. если нам требуется подключиться к уже существующей сети, то нам нужно поменять параметры у сети default: и указать параметр external: и name: существующей сети! Сеть должна уже существовать в пространстве docker ! Docker не будет создавать сеть а подключиться к существующей! | |  | | | |
| 1. если нам требуется создать сервисы в разных сетях, но нас устроит любая конфигурация по умолчанию, то мы можем просто создать 3 разных сети и потом в сервисах пропсиать сеть для каждого! Важно: должна быть третья сеть, которая будет связывать 2 несвязанные сети - смотри пример! | |  | |  | |
| 1. создание внутренних сетей с конкретным ip , чтобы разделить сервисы на разные сети и чтобы сервисы РАЗНЫХ СЕТЕЙ не видели друг друга: - в этом случае нужно в специальном разделе config: указать параметр subnet:   Для каждой сети нужно задать имя, если их больше чем одна в разделе networks: и потом прописывать в каждом сервисе какая сеть у какого сервиса! (сеть с именем default указывать md сервисах не нужно!)  В примере указано создание одной сети, другие по аналогии! | вариант сети с своим именем | | вариант для default сети | | |
| При описании сетей приходилось указывать driver: Чтобы объяснить драйвера, нужно утвердить факт, что сама платформа Docker на вашем компе имеет выход в интернет и соответственно внутри докер есть входная точка, через которую все контейнеры выходят в внешнюю сеть! Так вот, разные типы драйверов заставляют конетйнеры вести себя по разному с этой входной точкой docker:  **3 основных типа драйверов:**   1. **none** - никакой сети не будет и будет максимальная безопасность и контейнеры никогда не смогут обратиться к нулевой точке docker 2. **host** - каждый контейнер будет иметь такой же IP адрес, как на вашей машине, но ходить в интернет они будут по разным портам (например один контенер по 8080, а другой по 5000), т.е контейнеры будут на одном host но с разными портами! 3. **bridge** - каждый контейнер будет иметь свою сетевую карту (т.е ip), но IP будут в рамках одной сети! (драйвер по умолчанию.)   Существуют множество [пользовательских](https://docs.docker.com/engine/extend/plugins_network/) драйверов! | | |  | | |
| **Где и что лучше использовать:**   * **bridge** или Мост лучше всего использовать для связи нескольких контейнеров на одном и том же Docker хосте. Можно юзать docker-compose и выберать даную сеть для такой связки * **host** или Хост лучше всего юзать, когда сетевой стек не должен быть изолирован от хоста Docker (т.е от вашей локальной машины, на которой сам Docker) Таким образом, контейнеры будут юзать сеть вашей машины, будут иметь ip как у вашей машины, но отличаться будут портами! * **overlay/overlay2** или Оверлей Наложенная сеть лучше всего заюзать, когда вам нужны контейнеры, работающие на разных Docker хостах для связи, или, когда несколько приложений работают вместе, используя docker-swarm. Overlay предназначен для взаимодействия по сети с несколькими хостами, как в Docker Swarm или Kubernetes. Это позволяет контейнерам на хосте взаимодействовать друг с другом, не беспокоясь о настройке. В общем используйте при подготовке к Kubernetes))) * **macvlan** или Маквлан сети лучше всего использовать, когда вы переходите с VM/дедикейта  на контейнеры или хотите, чтобы ваши контейнеры выглядели как физические хосты в вашей сети, каждый с уникальным MAC-адресом Используй если нужен уникальный MAC-адрес! * **network plugins** сетевые плагины позволяют интегрировать Docker со специализированными сетевыми стеками | | | | | |

**Раздел secrets**

|  |  |  |  |
| --- | --- | --- | --- |
| В secrets помещается информация, которая должна быть зашифрована и не доступна для простого чтения дургими пользователями, например лицензии, пароли, секретные слова и тому подобное! Алгоритм работы следующий:   1. создаётся простой файл, например txt в корне docker compose 2. вариант 1: в пространстве docker через команду docker secrets create создаётся секрет с именем и потом это имя указывается в разделе secrets: в файле docker compose и указывается параметр external 3. вариант 2: в самом файле docker compose в разделе secrets: указывается имя секрета и потом указывается поле file: и указывается путь к файлу который будет зашифрован 4. после прописания секрета в разделе secrets: нужно уже в самих сервисах указать имена секретов, которые должны быть заинжекчены как каталог внутрь контейнера! Чтобы использовать заинжекченный секрет, можно прописать путь к нему в переменную environment: а уже саму переменную использовать в своей программе (смотри пример) |  | вариант 1:    вариант 2: | |
| После запуска в контейнере создаётся папка secrets (в категории run) и в ней будет лежать файл, который можно прочитать из нашего приложения |  |  |  |