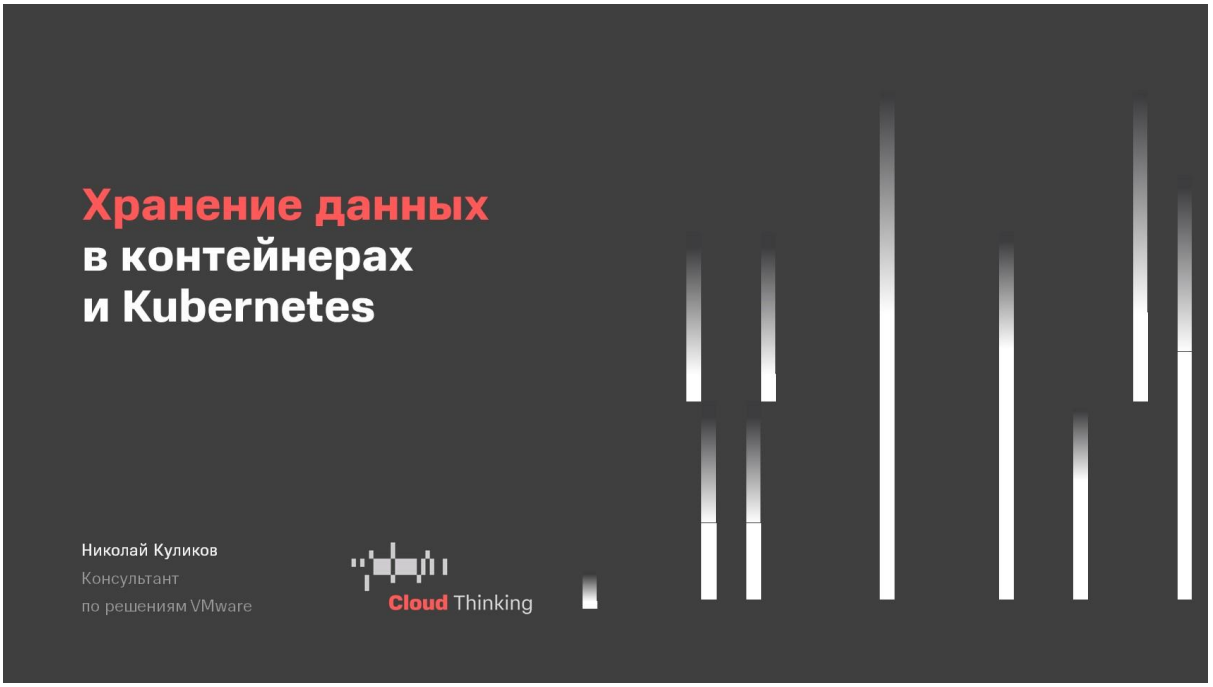
Cloud Thinking

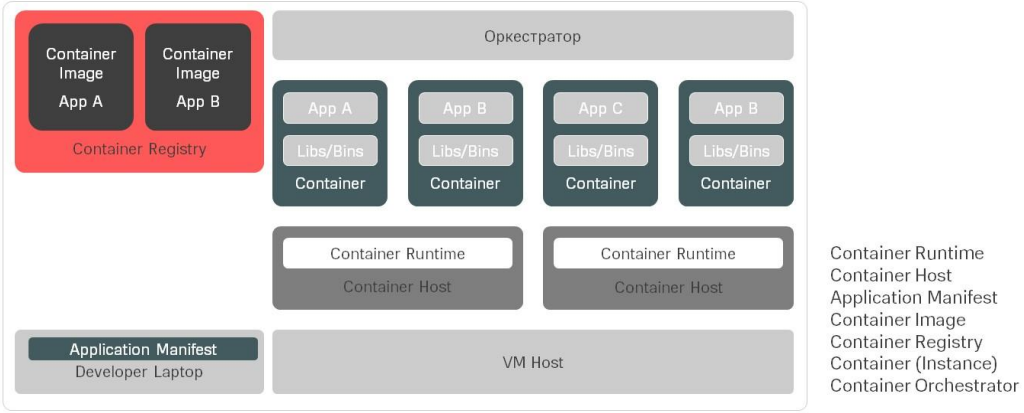
Модуль 3, урок 1



В прошлом модуле вы познакомились с основными компонентами и принципами работы контейнеров. В этом уроке предлагаем перейти к теме хранения данных в контейнерах и Kubernetes.

Краткий обзор контейнеров с точки зрения хранения данных

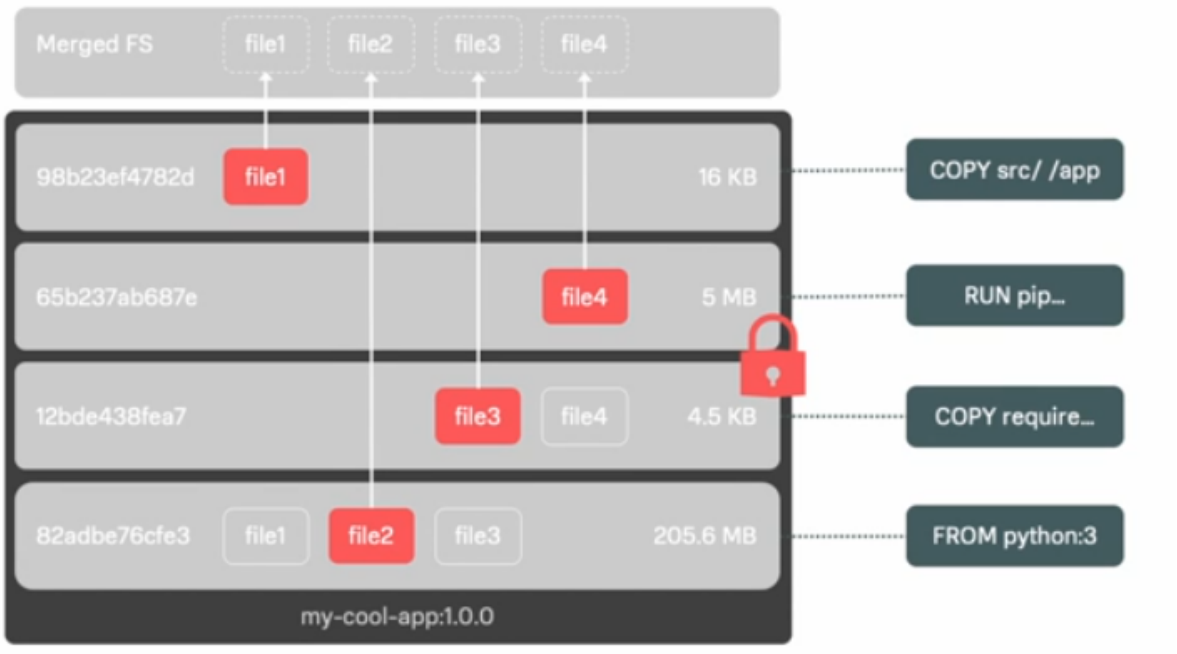
Основные компоненты контейнерной инфраструктуры



Container Host. Операционная система, на которой запускаются контейнеры. В большинстве случаев работает в BM, но может быть и на bare-metal. Как правило, это LinuxOS — Ubuntu, CentOS, PhotonOS. Windows тоже бывает, но это отдельная история. Внутри ОС установлен **Container Runtime**, который запускает контейнеры в форме процессов.

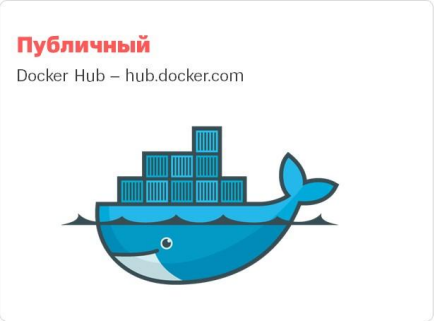
Container Image. Immutable образ приложения, обычно Tarball. Содержит все необходимые для запуска приложения зависимости и компоненты. Использует **Application Manifest** для сбора в Container Image.

Для управления образами используется OverlayFS — файловая система, которая работает по принципу Copy-on-Write (CoW) и представляет собой набор слоев. Каждое изменение добавляет новый слой поверх базового. Все слои в образе — Read-Only, неизменяемые.



Как только мы собрали образ, нам нужно где-то его хранить. Местом хранения образов выступают **Container Registry**. Они играют роль источника для Runtime.

Container Registry бывают публичные и приватные.



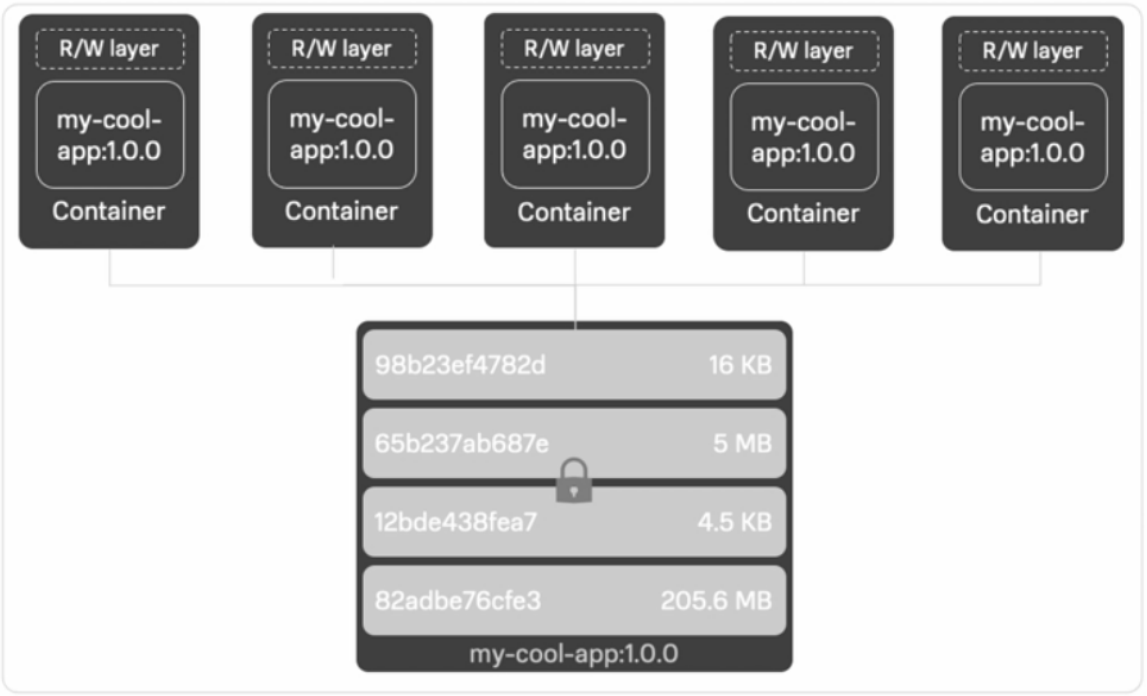
- **Публичные** Registry, как правило, располагаются в облаке. Сегодня наиболее популярен Docker Hub, где различные производители и участники сообщества выкладывают собственные образы приложений — их может скачать любой пользователь.
- У многих компаний из корпоративного сегмента есть **частные** Registry. Там хранятся данные только одной компании. Чаще всего для этих задач используется Harbor.

Кроме непосредственно возможности хранить образы, у них есть дополнительный функционал. Остановимся на наиболее важном.

- Проверка целостности — позволяет проверить, что мы запускаем ровно то, что записали, используются цифровые подписи образов.
- Сканирование на различные уязвимости — дает возможность убедиться, что как минимум публичные критические уязвимости в наших образах уже закрыты.

Запускаем контейнер

Когда мы говорим, что мы «запускаем контейнер», как правило, речь идет о **Container (Instance)** — запущенном экземпляре контейнера, Image которого мы взяли из Registry и запустили его на Docker Runtime.



Immutability образа позволяет ему быть общим. Проще говоря, мы можем запустить множество экземпляров из одного-единственного Image. При этом они будут изолированы друг от друга различными способами, о которых вы узнали из прошлого модуля.

С точки зрения хранения данных каждый из них имеет собственный тонкий R/W слой. Как вы помните, в Image у нас только Read Only, но для того, чтобы приложение работало и запускалось, ему нужно куда-то писать данные (как минимум временные).
Поэтому при каждом запуске у нас создается тонкий R/W слой, куда приложение внутри каждого инстанса пишет свои состояния и изменения.

Важно: при удалении контейнера изменения не сохраняются — все, что находилось в R/W слое, теряется. Это и позволяет нам обеспечивать immutability.

Смещение парадигмы

Домашние животные

- Каждый из них уникален
- Управляем, заботимся, обслуживаем исходя и индивидуальных потребностей
- Стараемся избежать гибели всеми силами, т.к. они «незаменимы». Лечим при необходимости

Скот

- Относимся просто «голова в стаде»
- Автоматически создаются – управляем только количеством
- Гибель – нормальный процесс и часто дешевле/быстрее заменить, чем лечить

Одна из концепций управления инфраструктурой называется Pets vs. Cattle — «домашние животные против сельскохозяйственного скота».

Подход Pets. Представьте, что у вас есть домашнее животное, например кот. Он уникален: у него есть имя и некое состояние. Если с котом что-то случится, мы побежим его лечить к ветеринару.

Подход Cattle. В то же время у вас есть, допустим, стадо овец. Здесь уже не идет речи об индивидуальности. Мы считаем головы в стаде, и единственное, что нас волнует, — сколько этих голов. Больную овцу легче заменить, чем вылечить, если что-то пошло не так.

То же самое применимо к инфраструктуре. Можно «держать» уникальные ресурсы или большие объемы однотипных. Во втором случае у нас появляется возможность автоматизации их создания и управления количеством. Ведь мы хотим просто иметь эти ресурсы, и совершенно неважно, что внутри.

И у того, и у другого подхода есть свои преимущества и недостатки. Однако концепция скота прекрасно легла на концепцию контейнеров: если приложение в контейнере стало плохо работать, мы можем просто удалить его и запустить новый.

В суровой реальности все не так однозначно: огромное число реальных контейнеров — stateful.

Со stateless все только начиналось. В контейнеры стали убирать примерно все: не только stateless-, но и stateful-приложения.

По статистике, 7 из 10 из наиболее популярных приложений — stateful. В них есть состояние, и нельзя утверждать, что они совершенно одинаковы.

Как возникла эта ситуация? На самом деле разработчикам стало очень удобно применять K8s **для любых типов приложений**, используя концепцию автоматизации.

Сначала K8s действительно использовался для stateless-приложений, но потом разработчики решили, что им нужны единые инструменты, процессы развертывания, управления, апгрейда, поэтому стали убирать в контейнеры stateful-приложения, которые изначально туда не особо хорошо ложились.

Что же делать с такими stateful-приложениями? Stateful-приложение требует хранить данные, например БД или загружаемые пользователем файлы. При перезапуске контейнера все изменения, записанные в R/W слой, потеряются.

Есть несколько вариантов решения проблемы.

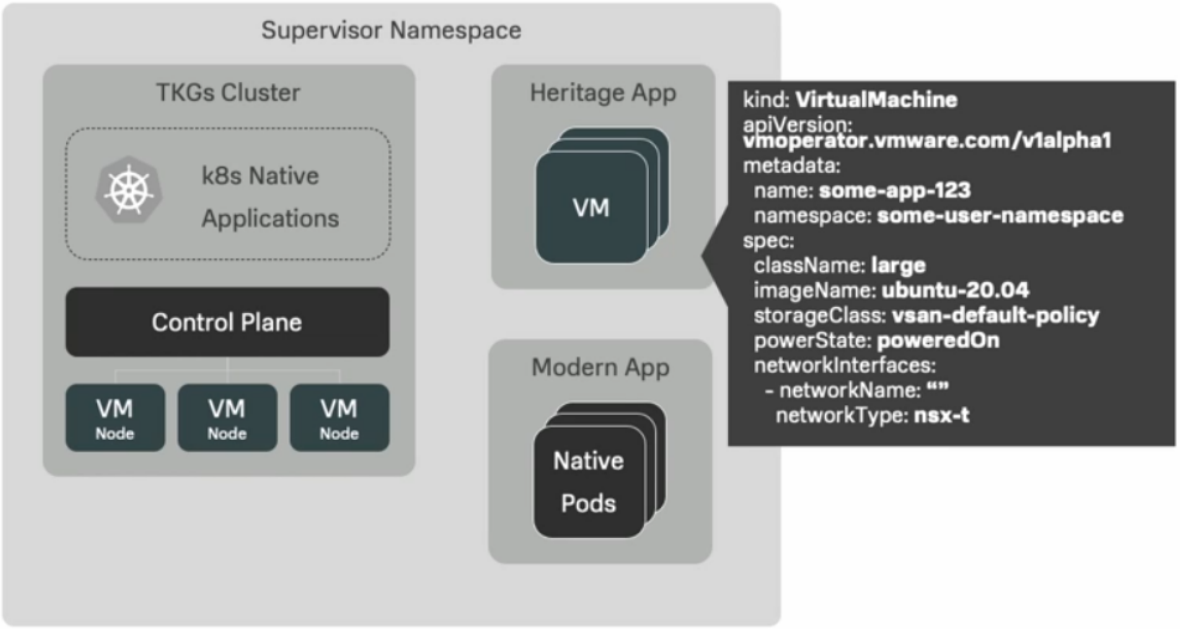
Вариант 1. Развернуть классические ВМ, но вместе с контейнерами и одними средствами

Де-факто это объединение подходов Pets и Cattles в рамках одной инфраструктуры.
Классические виртуальные машины хорошо ложатся на концепцию stateful-приложений. Для них есть понятные и зрелые инструменты хранения

данных, их резервного копирования и обработки. Но мы же изначально хотели использовать одинаковые инструменты? Поэтому давайте развертывать ВМ таким же способом, каким мы запускаем контейнеры, через те же инструменты и интеграции.

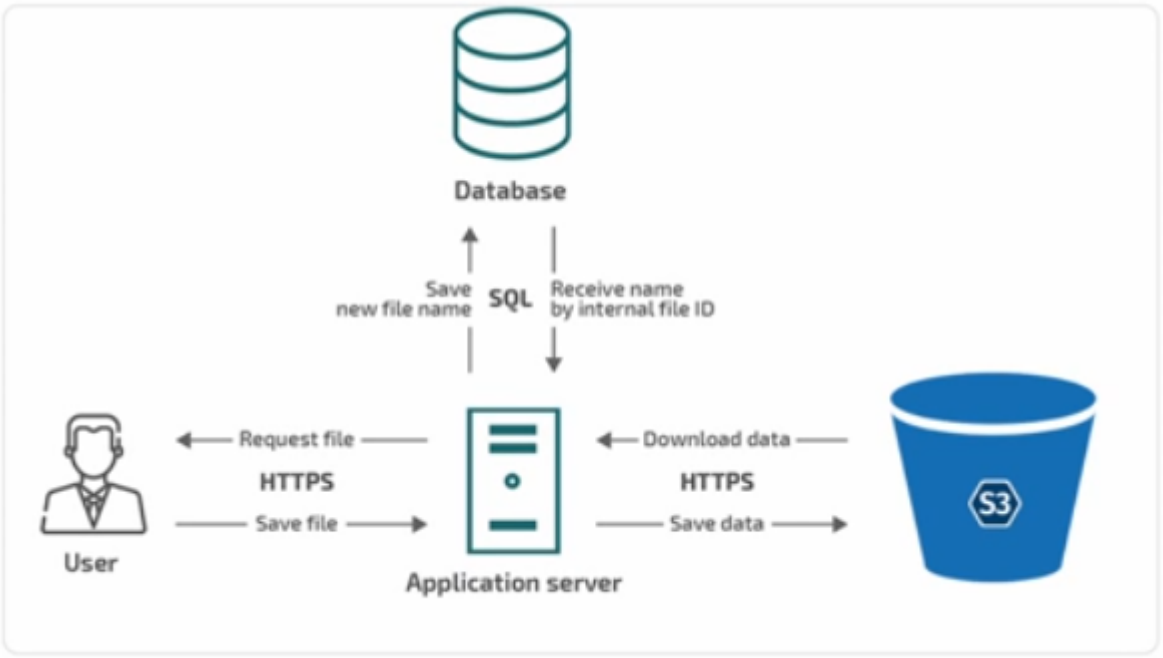
Это относительно новый подход и наиболее свежий из тех, о которых речь пойдет дальше. Он позволяет объединить эти противоречивые требования в рамках одной инфраструктуры.

- Для Dev-команды ничего не меняется — нужно только сделать другое описание для ВМ, в частности другие вызовы. Выглядеть это будет приблизительно так: в типе указываем виртуальную машину, некие метаданные и характеристики.



- Для Ops-команды сохраняются существующие средства обеспечения защиты данных, резервного копирования, управления.

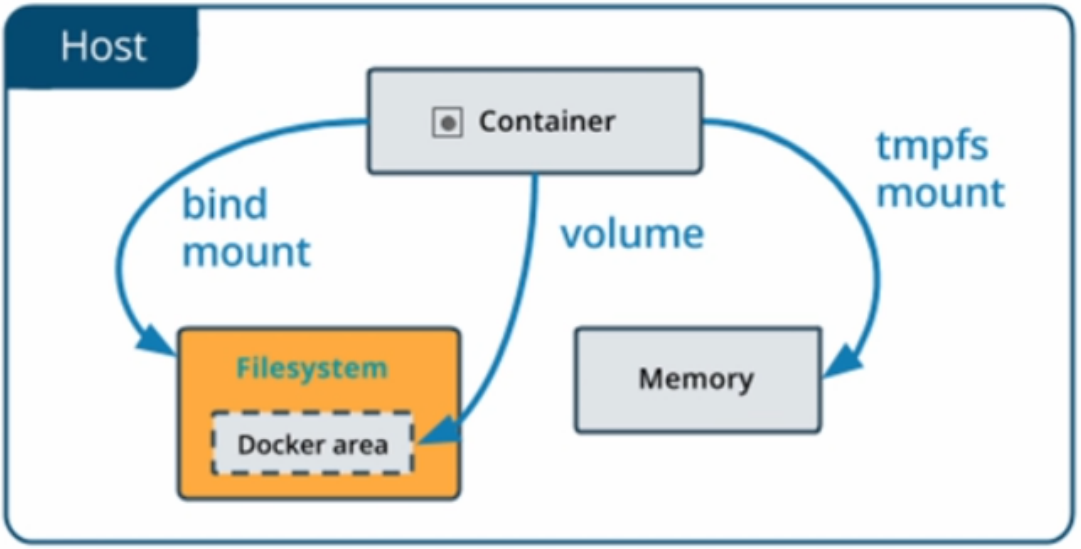
Вариант 2. Внешнее хранилище (обычно S3-совместимое)



Данные хранятся в виде объектов на внешнем постоянном хранилище. С одной стороны, это оптимальный способ: хранилища S3 очень гибко масштабируются и с ними достаточно просто работать. При этом приложение должно уметь работать с S3: обращение к данным происходит через HTTPS-вызовы и приложение должно с помощью этих вызовов «сходить» на S3, сделать put, get и так далее.

При этом S3, как правило, заточено на относительно «медленное» хранение с последовательным типом доступа, долгим сроком хранения и редким обращением. Поэтому класть базу данных в хранилище S3 — не самая здравая идея, а вот для хранения файлов, картинок и прочих пользовательских данных оно отлично подойдет.

Вариант 3. Docker Volumes



В качестве быстрого блочного или файлового персистентного хранилища, которое будет хранить состояние, причем вне жизненного цикла контейнера, подойдут **Volumes** в Docker.

Volumes — это способ замонтировать том/папку по ссылке или пути на внешнее относительно контейнера хранилище. Этот внешний том/папка может быть общим для нескольких контейнеров. Его можно свободно переподключать к заново созданным (после удаления) контейнерам — все изменения на внешнем Volume сохраняются.

Один Volume также можно подключить сразу к нескольким контейнерам как к отдельным инстансам. На стороне приложения стоит сразу решить, как с этим работать с точки зрения консистентности данных — чтобы не получилось блокировок или перезаписи данных.

Работа с Volumes выглядит следующим образом.

У Docker есть базовый компонент, управляющий хранилищем.

```
$ docker volume create myvol
```

Описание созданного Volume будет примерно таким:

```
$ docker volume inspect myvol
...
        "CreatedAt": "2021-09-10T09:25:30Z",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/myvol/_data",
        "Name": "myvol",
        "Options": {},
        "Scope": "local"
...
```

Просто так подключить созданный Volume к запущенному контейнеру не получится. Чтобы это сделать, необходимо остановить существующий контейнер и запустить новый, добавив Volume с внешнего хранилища.

```
$ docker run --name cont1 -v myvol:/mnt/volume -ti bitnami/minideb
```

Volume Name : Mount Point

```
$ docker run --name cont2 -v myvol:/mnt/volume -ti bitnami/minideb
```

```
$ docker stop cont1 cont2 && docker rm cont1 cont2
$ docker run --name cont4 -v myvol:/mnt/volume -ti bitnami/minideb
```

Контейнеризированные приложения должны уметь работать с Persistent Volume. К счастью, сейчас почти все популярные приложения, начиная с баз данных, уже умеют это делать. Однако не лишним будет изучить

документацию и узнать, как смонтировать директорию для сохранения состояния.