Лекция 13: Контейнеризация приложений с Docker. Решаем проблему "А у меня на машине все работает!"

Введение: Главная боль разработчиков

Здравствуйте! Представьте ситуацию, которая повторяется в IT-компаниях тысячи раз каждый день. Разработчик заканчивает работу над новой фичей, тестирует ее на своем ноутбуке и с уверенностью говорит: "Готово, у меня всё работает!". Он передает код тестировщику. Тестировщик запускает его у себя и... приложение падает с непонятной ошибкой. Разработчик в недоумении: "Странно, а у меня на машине все работает!".

Эта фраза — не просто мем. Это симптом глубокой и дорогостоящей проблемы — **проблемы расхождения окружений**. Код — это лишь верхушка айсберга. Для своей работы приложение зависит от огромного количества внешних факторов: версии языка программирования (Node.js), версии системных библиотек, настроек операционной системы, переменных окружения. Любое, даже самое незначительное различие в этих факторах между машиной разработчика, тестовым сервером и "боевым" сервером может привести к тому, что приложение будет вести себя совершенно по-разному.

Поиск причин таких проблем — это часы, а иногда и дни потраченного впустую времени, сорванные релизы и конфликты между отделами разработки и эксплуатации.

Сегодня мы изучим технологию, которая была создана, чтобы решить эту проблему раз и навсегда. Мы поговорим о **контейнеризации** и ее самой популярной реализации — **Docker**.

1. От Виртуальных Машин к Контейнерам

Исторически проблему окружений пытались решить с помощью **виртуальных машин (ВМ)**. Идея была в том, чтобы упаковать не только приложение, а **целую гостевую операционную систему** со всеми зависимостями в один большой файл-образ. Этот образ затем можно было запустить на любом физическом сервере с помощью специальной программы-**гипервизора** (VirtualBox, VMware). Это решало проблему идентичности окружений, но платой за это была огромная ресурсоемкость. Каждая ВМ — это полноценная ОС, которая весит десятки гигабайт и "отъедает" значительную часть процессорного времени и памяти на собственные нужды.

Контейнеризация предлагает более изящное и легковесное решение.

Ключевая идея: зачем нам эмулировать целую операционную систему для каждого приложения, если все они могут работать на серверах с Linux и **использовать ядро операционной системы хоста совместно**? Нам нужно изолировать только **процессы и файловую систему** самого приложения.

Контейнер — это и есть такое изолированное пространство. Он содержит только само приложение и его непосредственные зависимости (библиотеки, среду выполнения), но использует ядро хостовой ОС.

Аналогия:

- **Виртуальная машина** — это как **перевозить один станок, арендовав для него целый грузовой самолет**. Надежно, но безумно дорого и неэффективно.

- **Контейнер** — это как положить станок в **стандартный морской контейнер**. Его можно перевозить на любом корабле, поезде или грузовике, и он везде будет работать одинаково.

Благодаря этому контейнеры **легковесны** (образы весят мегабайты, а не гигабайты), **быстро запускаются** (за доли секунды, а не за минуты) и **эффективно используют ресурсы**.

2. Docker: Кит, который изменил IT-индустрию

Docker — это платформа, которая сделала технологию контейнеризации простой, доступной и невероятно популярной. Именно Docker стал тем самым "стандартом морского контейнера" для мира программного обеспечения.

Вся работа с Docker строится на трех основных концепциях:

1. **Image (Образ):** Это **шаблон** или "чертеж". Неизменяемый пакет, который включает в себя всё необходимое для запуска приложения: код, среду выполнения (Node.js), библиотеки, переменные окружения. Аналогия: **класс** в объектно-ориентированном программировании.
2. **Container (Контейнер):** Это **запущенный, работающий экземпляр образа**. Из одного образа можно запустить сколько угодно одинаковых, но полностью изолированных друг от друга контейнеров. Аналогия: **объект (экземпляр)** класса.
3. **Registry (Реестр):** Это **хранилище** для Docker-образов. Самый известный — **Docker Hub**, который является своего рода "GitHub'ом для Docker-образов". Там хранятся тысячи официальных образов для любого ПО (Node.js, PostgreSQL, Redis и т.д.).

3. Практика: "Контейнеризируем" наше To-Do API

Давайте возьмем наше приложение, которое мы создали в прошлом семестре, и "упакуем" его в Docker-контейнер. Для этого мы создадим специальный файл-инструкцию — Dockerfile.

Dockerfile — это текстовый файл, который содержит пошаговый "рецепт" для сборки Docker-образа. Каждая инструкция в нем создает новый, кэшируемый "слой" в образе.

Шаг 1: Выбор базового образа (FROM)
Мы не строим окружение с нуля. Мы берем за основу готовый официальный образ, в котором уже есть Linux и нужная нам версия Node.js. Мы будем использовать -alpine версию, так как она очень легковесная.

```
# Используем официальный образ Node.js v18 на базе Alpine Linux
FROM node:18-alpine
```

Шаг 2: Установка рабочей директории (WORKDIR)
Эта инструкция создает директорию внутри образа (например, /app) и делает ее текущей для всех последующих команд.

```
WORKDIR /app
```

Шаг 3: Установка зависимостей (COPY и RUN)
Это самый важный трюк для эффективной сборки. Чтобы Docker мог кэшировать node_modules, мы сначала копируем только package.json и package-lock.json, устанавливаем зависимости, и только **потом** копируем остальной код.

```
# Копируем только файлы с описанием зависимостей
COPY package*.json ./

# Выполняем установку production-зависимостей
RUN npm ci --only=production
```

Теперь этот слой будет пересобираться только тогда, когда мы изменим зависимости, а не при каждом изменении нашего кода.

Шаг 4: Копирование кода приложения (COPY)
Теперь копируем весь остальной код нашего приложения в рабочую директорию. Чтобы случайно не скопировать лишнее (например, локальные node_modules или папку .git), мы создаем файл .dockerignore.

```
# Копируем всё из текущей папки в /app внутри образа
COPY . .
```

Ша- 5: Открытие порта (EXPOSE)
Эта инструкция — своего рода документация. Она информирует Docker, что приложение внутри контейнера будет "слушать" порт 3000.

```
EXPOSE 3000
```

Шаг 6: Команда запуска (CMD)
Эта инструкция задает команду, которая будет выполнена при запуске контейнера.

```
# Запускаем приложение
CMD [ "node", "index.js" ]
```

4. Сборка и запуск

Теперь, когда у нас есть Dockerfile, мы можем собрать наш образ и запустить контейнер.

Сборка образа (docker build):
Открываем терминал в корне проекта и выполняем команду:

```
# -t (tag) - задает имя и тег (версию) для нашего образа
docker build -t my-todo-api:1.0 .
```

Запуск контейнера (docker run):

```
# -d - запустить в фоновом режиме
# -p 3000:3000 - "пробросить" порт 3000 изнутри контейнера на порт 3000 нашей машины
# --name my-api - дать контейнеру имя
docker run -d -p 3000:3000 --name my-api my-todo-api:1.0
```

Теперь, если вы откроете в браузере или Postman http://localhost:3000, вы обратитесь к нашему приложению, которое работает в **полностью изолированном контейнере**!

Заключение

Сегодня мы сделали огромный шаг в сторону современного DevOps. Мы поняли, как технология контейнеризации решает фундаментальную проблему "расхождения окружений". Мы познакомились с Docker, его ключевыми концепциями (Image, Container) и на практике научились писать Dockerfile для "упаковки" нашего Node.js-приложения.

Теперь у нас есть стандартный, переносимый "артефакт", который будет работать абсолютно одинаково на ноутбуке разработчика, на тестовом сервере и в продакшене в облаке.

На следующей лекции мы пойдем дальше. Наше приложение — это не только код, но и база данных, и кэш. Мы научимся управлять **системой из нескольких контейнеров** с помощью **Docker Compose**.