

# Docker

## Orchestration

# Agenda

Проблема

Основи Docker: Image, Container, Dockerfile

Docker Compose

Docker та мікросервіси

Проблема масштабування

Вступ до Kubernetes

# Problem

"На моїй  
машині все  
працювало!  
"

Повна ізоляція ОС

Високі накладні витрати (пам'ять, CPU)

Повільний запуск

Важкі образи (гігабайти)

Ізоляція на рівні процесів (спільне ядро  
ОС)

Мінімальні накладні витрати

Швидкий запуск (секунди)

Легкі образи (мегабайти)

Віртуальні  
машини

Контейнери

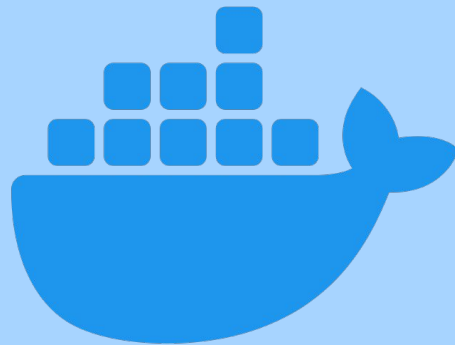
# Основи Docker

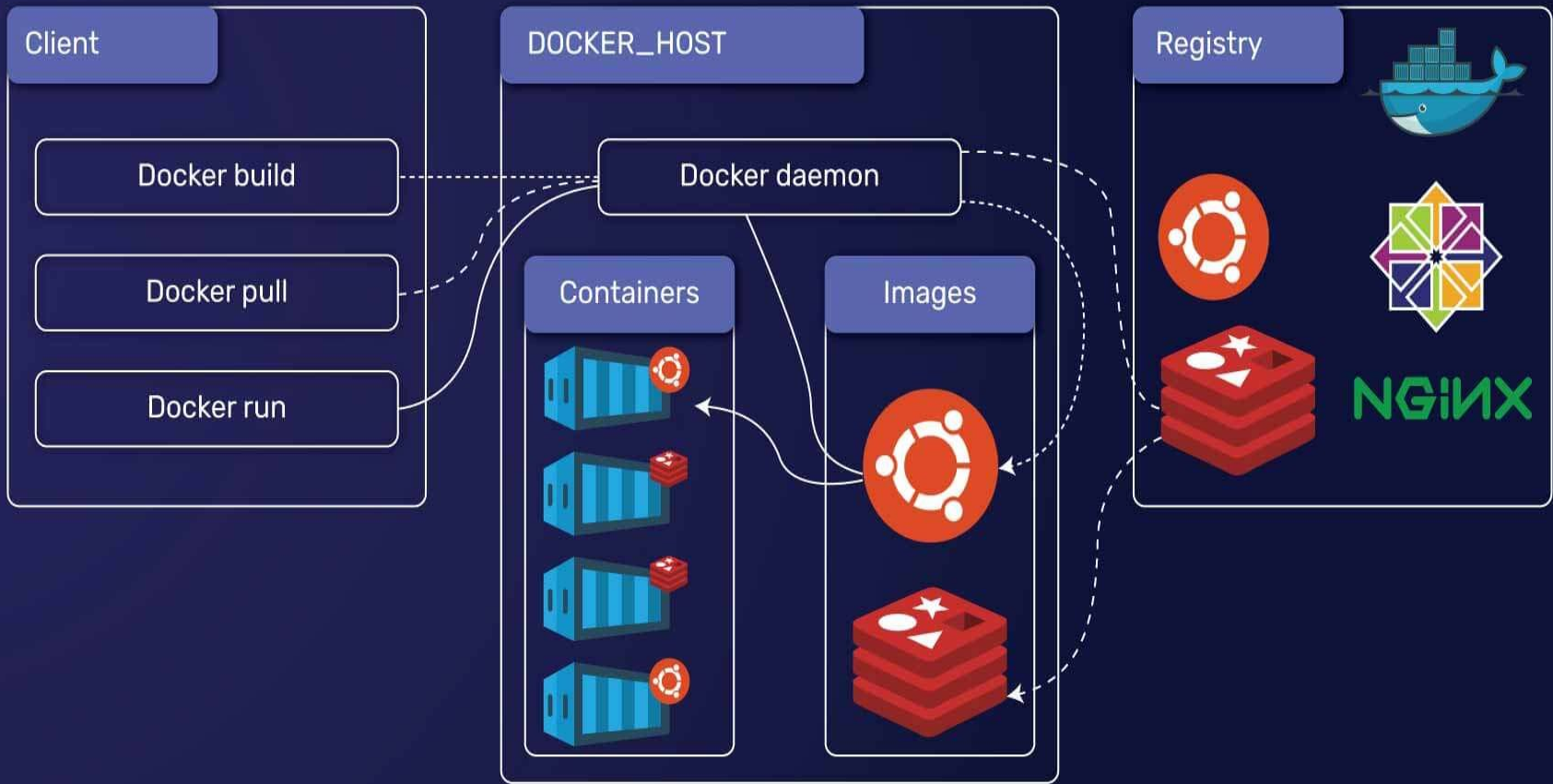
**Image (Образ):** Шаблон для створення контейнерів. Це як креслення або клас в ООП. Образ складається з шарів (layers), що робить його ефективним для зберігання та передачі.

**Container (Контейнер):** Запущений екземпляр образу. Це ізольоване середовище, де виконується ваш застосунок. Це об'єкт, створений з класу.

**Dockerfile:** Текстовий файл з інструкціями для збірки образу. Це рецепт, за яким Docker створює ваш образ крок за кроком.

**Registry (Реєстр):** Сховище для образів (напр., Docker Hub, AWS ECR).





# Dockerfile

```
# 1. Використовуємо офіційний базовий образ Node.js
FROM node:18-alpine

# 2. Встановлюємо робочу директорію всередині контейнера
WORKDIR /app

# 3. Копіюємо package.json та встановлюємо залежності
COPY package*.json ./
RUN npm install

# 4. Копіюємо решту коду застосунку
COPY . .

# 5. Відкриваємо порт, на якому працюватиме застосунок
EXPOSE 3000

# 6. Команда для запуску застосунку
CMD ["node", "server.js"]
```

# Docker Compose

```
version: '3.8'

services:
  webapp:
    build: .
    ports:
      - "8080:3000"
    environment:
      - MONGO_URL=mongodb://database:27017/mydb
    depends_on:
      - database

  database:
    image: "mongo:5.0"
    volumes:
      - mongo-data:/data/db

volumes:
  mongo-data:
```

# Docker та мікросервісна архітектура

Кожен мікросервіс працює у власному контейнері, зі своїми залежностями. Конфлікти версій бібліотек виключені.

Сервіс авторизації може бути на Go, сервіс замовлень на Java, а аналітика на Python. Docker це дозволяє.

Можна оновити лише один сервіс, не чіпаючи інші. Це прискорює цикл розробки.

Якщо сервіс платежів сильно навантажений, можна запустити 10 його екземплярів, залишивши один екземпляр сервісу профілю користувача.

Ізоляція

Поліглотність

Незалежне  
розгортання

Масштабування

Problem

# Зоопарк контейнерів

## Розгортання та оновлення

Як оновити 50 контейнерів сервісу без простою (zero-downtime)?

## Масштабування

Як автоматично додати контейнери при зростанні навантаження і прибрати при спаді?

## Service Discovery

Як контейнер А знайде контейнер Б, якщо їхні IP-адреси постійно змінюються?

## Self-Healing

Що робити, якщо контейнер або цілий сервер "впав"?

## Балансування навантаження

Як розподілити трафік між 10 екземплярами одного сервісу?

# Kubernetes [K8s]



Open-source платформа для автоматизації розгортання, масштабування та управління контейнеризованими застосунками. Створена в Google.

Декларативний підхід: Ви не кажете K8s, що робити (imperative), а описуєте бажаний стан (declarative). Наприклад: "Я хочу, щоб завжди було запущено 5 екземплярів мого веб-сервісу". Kubernetes сам подбає про те, щоб це було так.

## Ключові можливості

- Автоматичне масштабування (Horizontal Pod Autoscaling)
- Самовідновлення (Self-healing)
- Оновлення без простою (Rolling updates)
- Управління конфігурацією та секретами
- Балансування навантаження та service discovery

# Архітектура Kubernetes

## Control Plane (Мозок кластера):

API Server: "Вхідні ворота" до кластера. Все управління йде через нього.

etcd: Надійна база даних, де зберігається стан всього кластера.

Scheduler: Вирішує, на якому з Worker Node запустити новий контейнер.

Controller Manager: Слідкує за станом кластера і приводить його до бажаного.

## Worker Nodes (Робочі вузли):

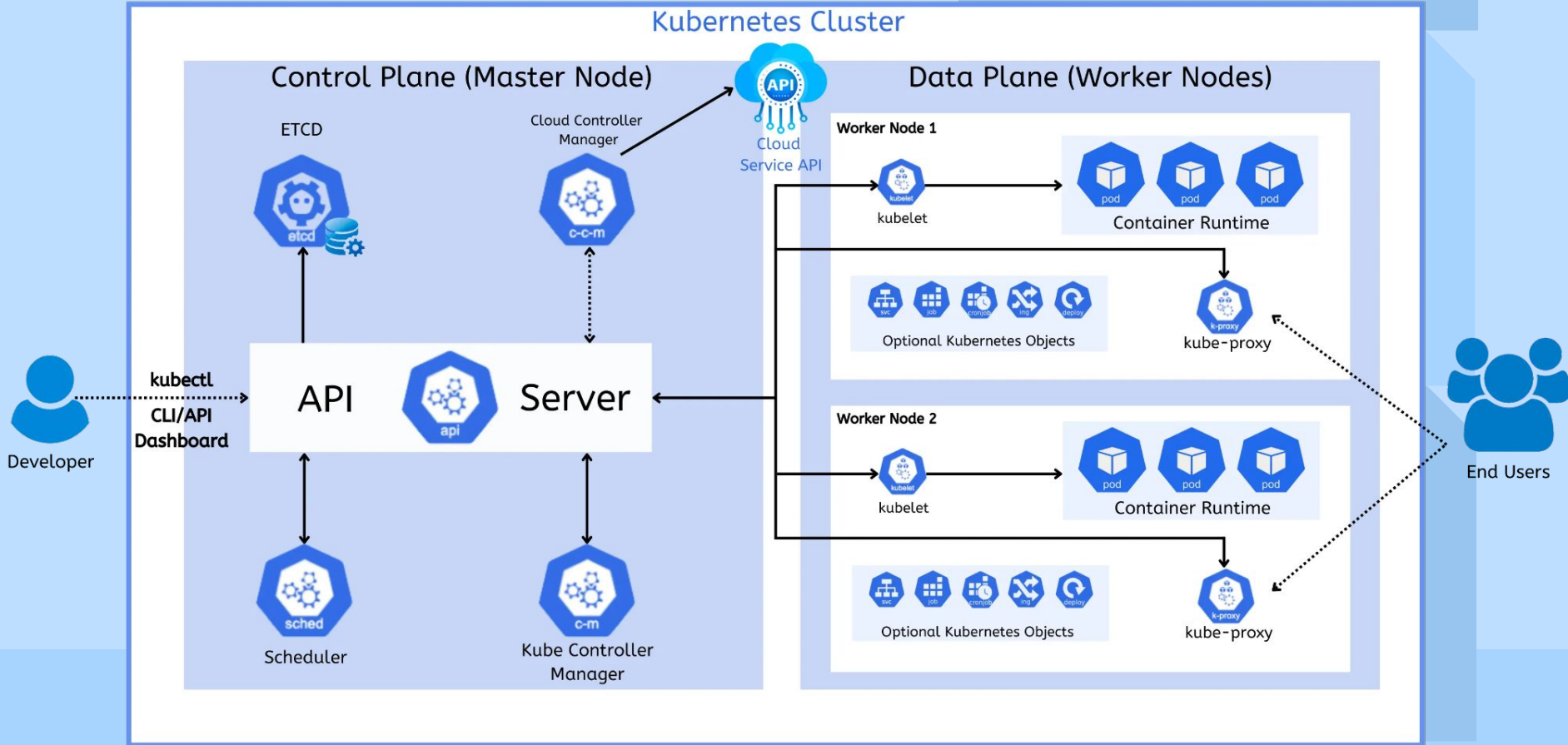
Kubelet: "Агент" K8s на кожному вузлі. Стежить, щоб контейнери на цьому вузлі працювали.

Kube-proxy: Відповідає за мережеву взаємодію.

Container Runtime: Програма, що безпосередньо запускає контейнери (напр., Docker).

# Ключові об'єкти Kubernetes

ress	Pod	Deployment	ReplicaSet	Service	ConfigMap / Secret	Ing
	Найменша одиниця розгортання в K8s. Це "обгортка" навколо одного або кількох тісно пов'язаних контейнерів. Вони ділять спільну мережу та сховище. Зазвичай, один под — один контейнер.	Декларативний опис для управління Подами. Ви кажете: "Хочу 3 репліки мого додатку". Deployment створює їх і стежить, щоб їх завжди було 3. Він також керує процесом оновлення.	Об'єкт, що забезпечує задану кількість однакових Подів. Зазвичай, ви не створюєте його напямую, ним керує Deployment.	Поды — ефемерні. Вони можуть вмирати, перезапущатися і отримувати нові IP-адреси. Service надає стабільну IP-адресу та DNS-ім'я для набору Подів. Він діє як внутрішній балансувальник навантаження.	Дозволяють відокремити конфігурацію та секретні дані (паролі, ключі API) від образу контейнера. Ці об'єкти монтуються в Поды як файли або змінні середовища.	



# Manifest Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nodejs-app-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-nodejs-app
  template:
    metadata:
      labels:
        app: my-nodejs-app
    spec:
      containers:
        - name: web-server
          image: your-dockerhub-username/my-nodejs-app:1.0
          ports:
            - containerPort: 3000
```

# Manifest Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-nodejs-service
spec:
  type: LoadBalancer
  selector:
    app: my-nodejs-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
```

# Переваги та виклики

**Портативність:** Працює однаково скрізь — на ноутбучі розробника, на тестовому сервері, в будь-якій хмарі.

**Масштабованість та еластичність:** Легко адаптується до будь-якого навантаження.

**Відмовостійкість:** Автоматичне відновлення після збоїв.

**Ефективне використання ресурсів:** "Упаковує" застосунки щільніше, ніж VM.

**Прискорення циклу розробки:** Стандартизація та автоматизація.

**Складність:** Kubernetes має високий поріг входження. Це комплексна система.

**Накладні витрати:** Сам Control Plane потребує ресурсів. Для дуже простих проектів це може бути надлишковим.

**Зміна парадигми:** Вимагає від розробників нового мислення (stateless-додатки, декларативна конфігурація).