**Архитектура, масштабирование и продвинутое управление Node.js**

В этой главе ты узнаешь:

* Как масштабировать Node.js-приложения
* Как использовать **PM2** и кластерный режим
* Как внедрить **Redis**, **load balancer**, **Kubernetes**
* Как проектировать **надёжную микросервисную архитектуру**

**📘 10.1 Почему Node.js нуждается в масштабировании?**

Node.js — это **однопоточный** рантайм. Один процесс = один поток.

📉 При большом трафике:

* Один процесс не справляется
* Нельзя использовать все ядра CPU

➡️ **Решение**: запускать несколько процессов или контейнеров  
(и распределять нагрузку между ними)

**⚙️ 10.2 PM2 — процесс-менеджер №1 для Node.js**

**📦 Установка:**

bash

КопироватьРедактировать

npm install -g pm2

**🚀 Запуск приложения:**

bash

КопироватьРедактировать

pm2 start index.js --name app

**🧠 Кластерный режим:**

bash

КопироватьРедактировать

pm2 start index.js -i max

🔹 -i max — запускает процесс на каждом ядре CPU  
🔹 Все процессы балансируют нагрузку между собой

**🗃️ 10.3 Конфигурация с eco-файлом**

📄 ecosystem.config.js

js

КопироватьРедактировать

module.exports = {

apps: [{

name: "api",

script: "./index.js",

instances: "max",

exec\_mode: "cluster",

env: {

NODE\_ENV: "production"

}

}]

}

📦 Запуск:

bash

КопироватьРедактировать

pm2 start ecosystem.config.js

**🧠 10.4 Redis как кэш и брокер**

Redis — супербыстрая in-memory БД, которую можно использовать:

* как кэш (ответы, токены)
* как message-broker (в pub/sub)
* как rate limiter

**Пример кэширования:**

js

КопироватьРедактировать

const redis = require("redis");

const client = redis.createClient();

client.get("users", async (err, data) => {

if (data) return res.send(JSON.parse(data));

const users = await User.find();

client.setex("users", 3600, JSON.stringify(users));

res.send(users);

});

**🔁 10.5 Load Balancer (HAProxy / Nginx)**

Для распределения трафика между **несколькими инстансами**:

* Используется перед контейнерами/сервисами
* Следит за healthcheck, пробрасывает порты

📦 Nginx как L7 Load Balancer:

nginx

КопироватьРедактировать

upstream node\_backend {

server app1:3000;

server app2:3000;

}

server {

listen 80;

location / {

proxy\_pass http://node\_backend;

}

}

**☸️ 10.6 Kubernetes — продвинутый оркестратор**

| **Объект** | **Описание** |
| --- | --- |
| Pod | Группа контейнеров |
| Deployment | Обновление и управление версиями |
| Service | Постоянный endpoint |
| ConfigMap | Настройки |
| Secret | Безопасные ключи |

**Минимальный deployment.yaml**

yaml

КопироватьРедактировать

apiVersion: apps/v1

kind: Deployment

metadata:

name: node-app

spec:

replicas: 3

selector:

matchLabels:

app: node

template:

metadata:

labels:

app: node

spec:

containers:

- name: app

image: username/my-node-app:latest

ports:

- containerPort: 3000

**📐 10.7 Архитектура микросервисов**

Каждый сервис:

* Имеет собственный Dockerfile
* Имеет собственную базу или schema
* Общается через HTTP или message queue (RabbitMQ, Kafka)

**Пример:**

sql

КопироватьРедактировать

📦 User Service

📦 Auth Service

📦 Product Service

📦 API Gateway

📦 MongoDB / Redis / RabbitMQ

**🛡️ 10.8 Защита и масштабирование API**

| **Подход** | **Объяснение** |
| --- | --- |
| Rate Limit | Защищает от спама (например, 100 req/min) |
| JWT | Авторизация |
| Nginx Reverse Proxy | Скрывает структуру |
| CORS Policy | Разрешённые домены |
| Logging & Alerting | Мониторинг |

**🧭 10.9 Стратегия масштабирования**

| **Уровень** | **Подход** |
| --- | --- |
| Горизонтальное | Несколько контейнеров или подов |
| Вертикальное | Больше CPU/RAM |
| Auto Scaling | K8s или AWS ECS |
| Кэширование | Redis или CDN |
| Разделение сервисов | Микросервисы |

**✅ 10.10 Вывод главы**

Теперь ты знаешь:

* Как запускать Node.js в кластере через PM2
* Как использовать Redis и балансировку
* Как перейти от Docker Compose к Kubernetes
* Как проектировать масштабируемые и отказоустойчивые API