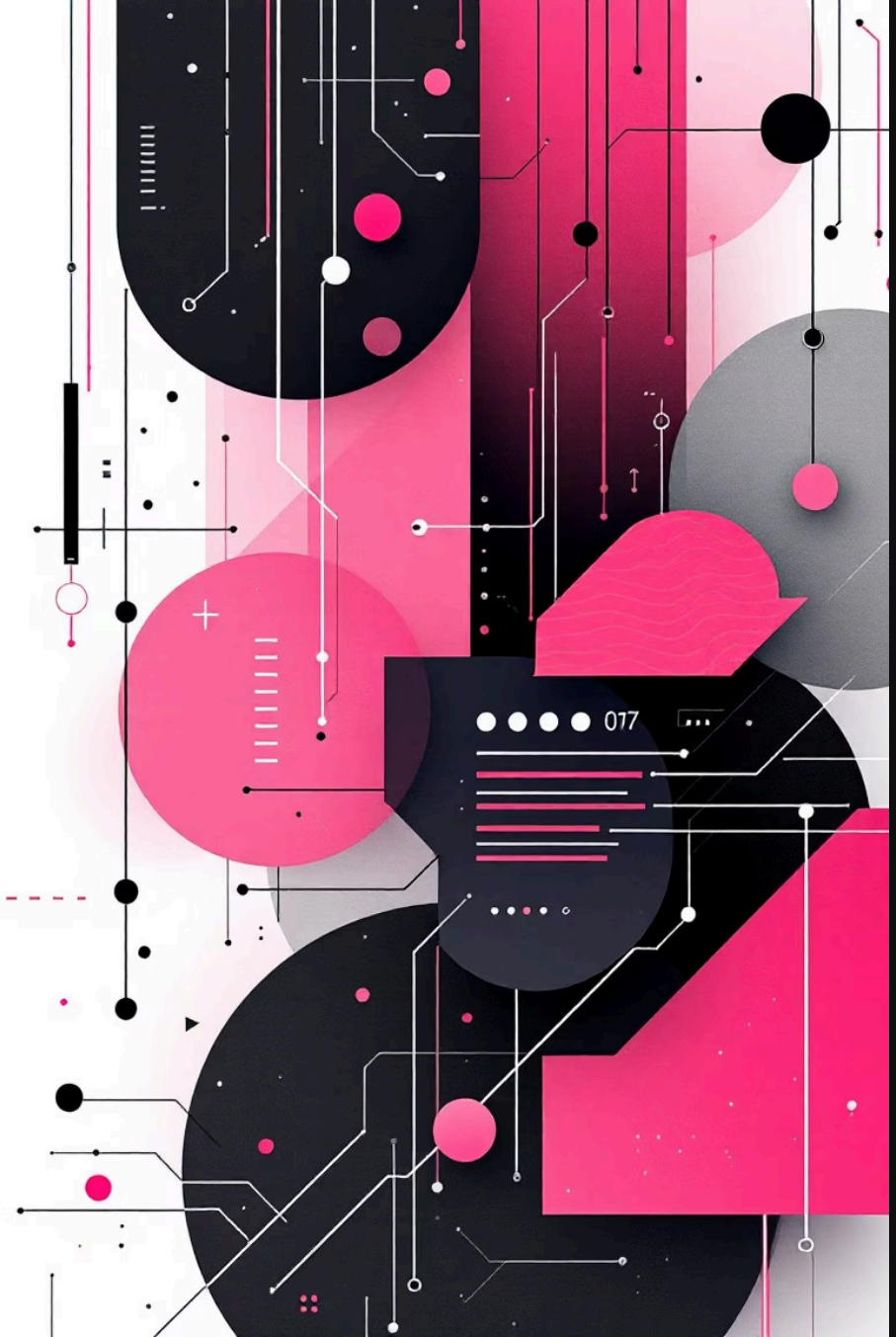


Стратегии тестирования микросервисов

Переход от ручного подхода к автоматизированным стратегиям

Презентация для студентов и практикующих разработчиков ПО.

Глава I

Проблемы традиционного подхода к тестированию

Традиционный подход, при котором тестирование проводится только после завершения разработки, неэффективен и замедляет доставку ПО, особенно в динамичной микросервисной среде.

Ручное тестирование неэффективно

Люди медленны и не могут работать круглосуточно. Полагаясь на ручное тестирование, невозможно обеспечить быструю и безопасную доставку кода.

Тестирование происходит слишком поздно

Тестирование уже написанного приложения менее продуктивно. Тесты должны быть включены в процесс разработки, чтобы разработчики видели результаты во время редактирования кода.

Культурные барьеры автоматизации

Несмотря на доступность инструментов (например, JUnit с 1998 года), уровень автоматизации тестов остается низким. Это вызвано прежде всего культурными, а не техническими причинами.

26%

Частичная автоматизация

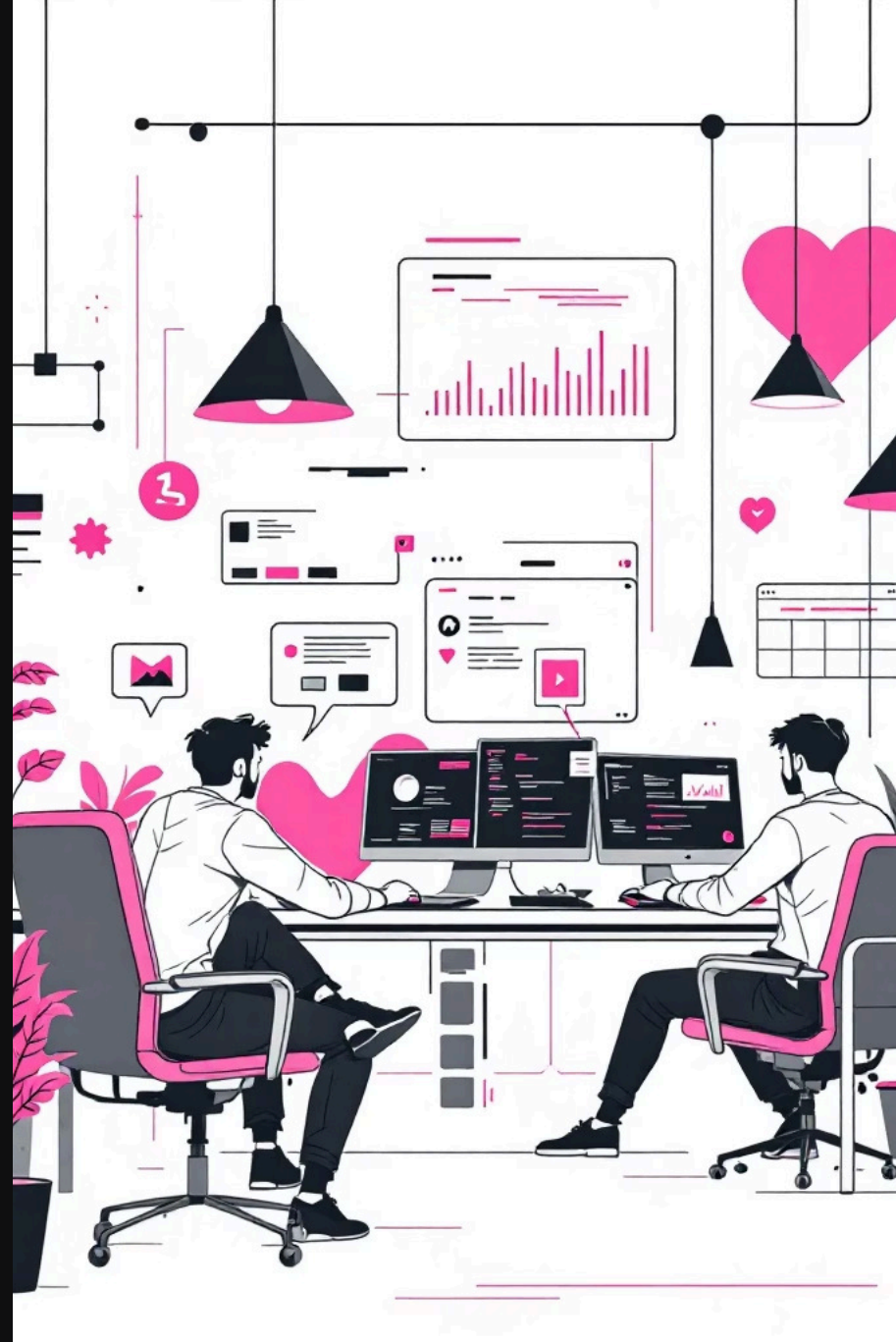
Доля организаций с автоматизированным тестированием (по отчету Sauce Labs, 2018).

3%

Полная автоматизация

Доля организаций, достигших полной автоматизации тестирования.

> «Тестированием должен заниматься отдел QA», «У разработчиков есть более важные задачи» — такие убеждения препятствуют внедрению автоматизации.



Глава II

Основы автоматического тестирования

Автоматические тесты — это основа успешной микросервисной архитектуры. Их цель — проверить поведение тестируемой системы, будь то отдельный класс или целый сервис.

01

1. Подготовка (Setup)

Инициализация среды, настройка зависимостей и приведение системы в требуемое начальное состояние.

02

2. Выполнение (Execution)

Запуск тестируемой системы, например, вызов конкретного метода или отправка запроса.

03

3. Проверка (Verification)

Оценка результатов выполнения и нового состояния системы на соответствие ожидаемым значениям.

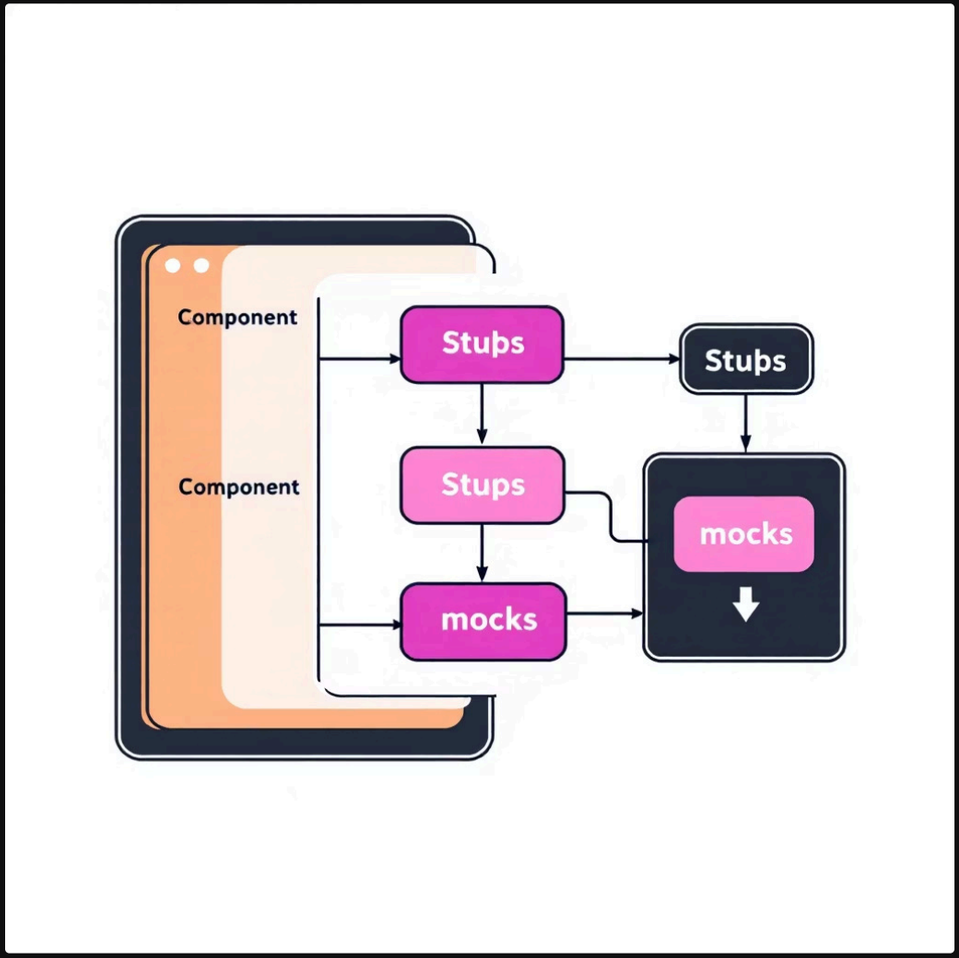
04

4. Очистка (Cleanup)

Удаление тестовой среды и откат изменений (например, транзакций БД), если это необходимо.

Тестирование в изоляции с использованием дублеров

Зависимости могут сделать тесты медленными, сложными и ненадежными. Для тестирования системы в изоляции используются специальные объекты-дублеры, симулирующие поведение реальных зависимостей.



Дублеры (Doubles)

Объекты, симулирующие поведение зависимостей для изоляции тестируемой системы (например, OrderController с дублером OrderService).

Заглушки (Stubs)

Дублеры, которые просто возвращают определенные, заранее заданные значения тестируемой системе.

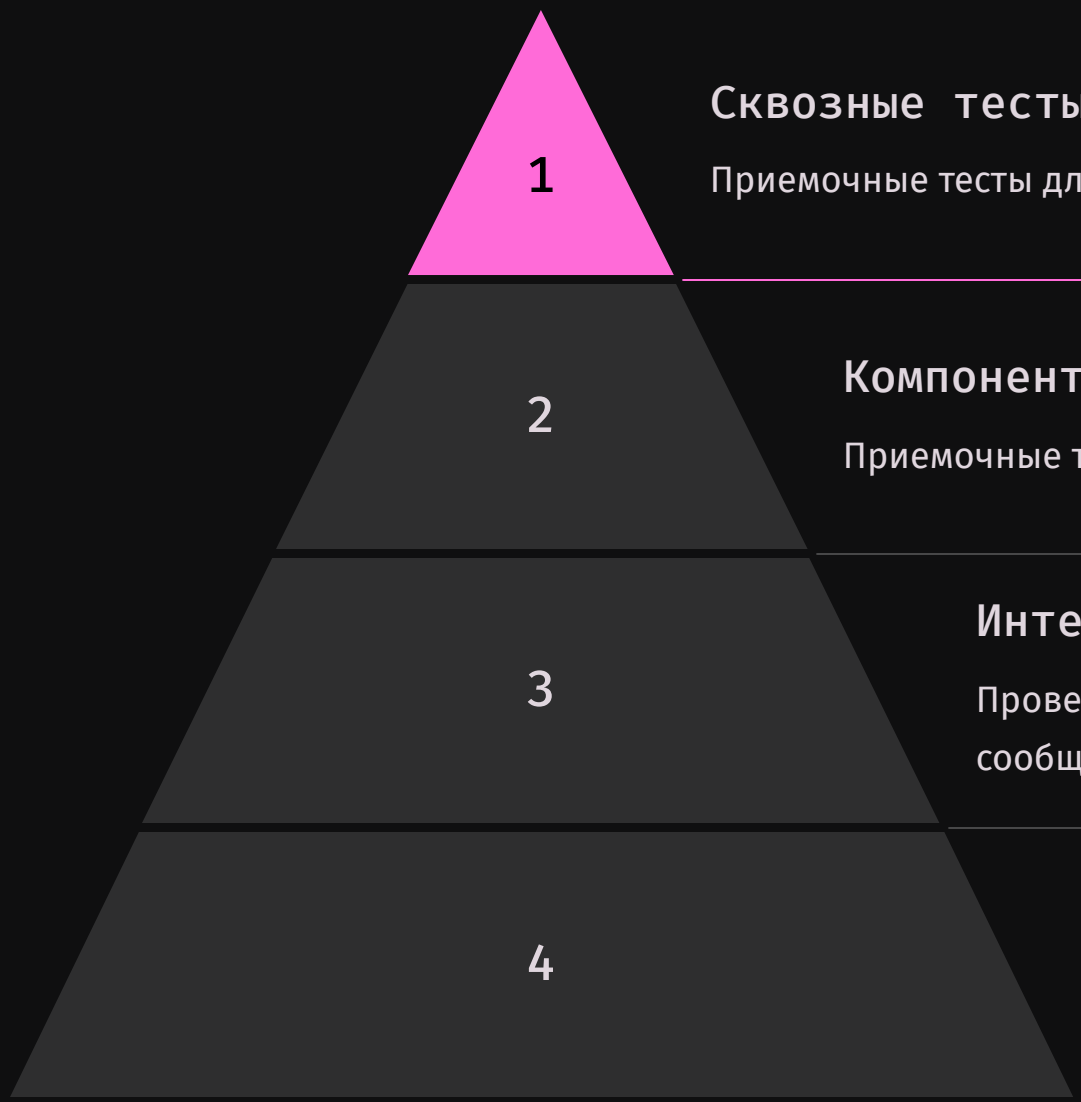
Макеты (Mocks)

Дублеры, используемые для проверки, что тестируемая система корректно взаимодействует с ними (например, вызывала нужный метод).

Глава III

Классификация и приоритизация тестов: Пирамида

Тестовая пирамида описывает относительные пропорции различных типов функциональных тестов по охвату. Чем выше тест в пирамиде, тем медленнее и сложнее он должен быть.



Сквозные тесты (End-to-End)

Приемочные тесты для всего приложения. **Мало.**

Компонентные тесты

Приемочные тесты для отдельного сервиса.

Интеграционные тесты

Проверка взаимодействия с инфраструктурой (БД, брокеры сообщений).

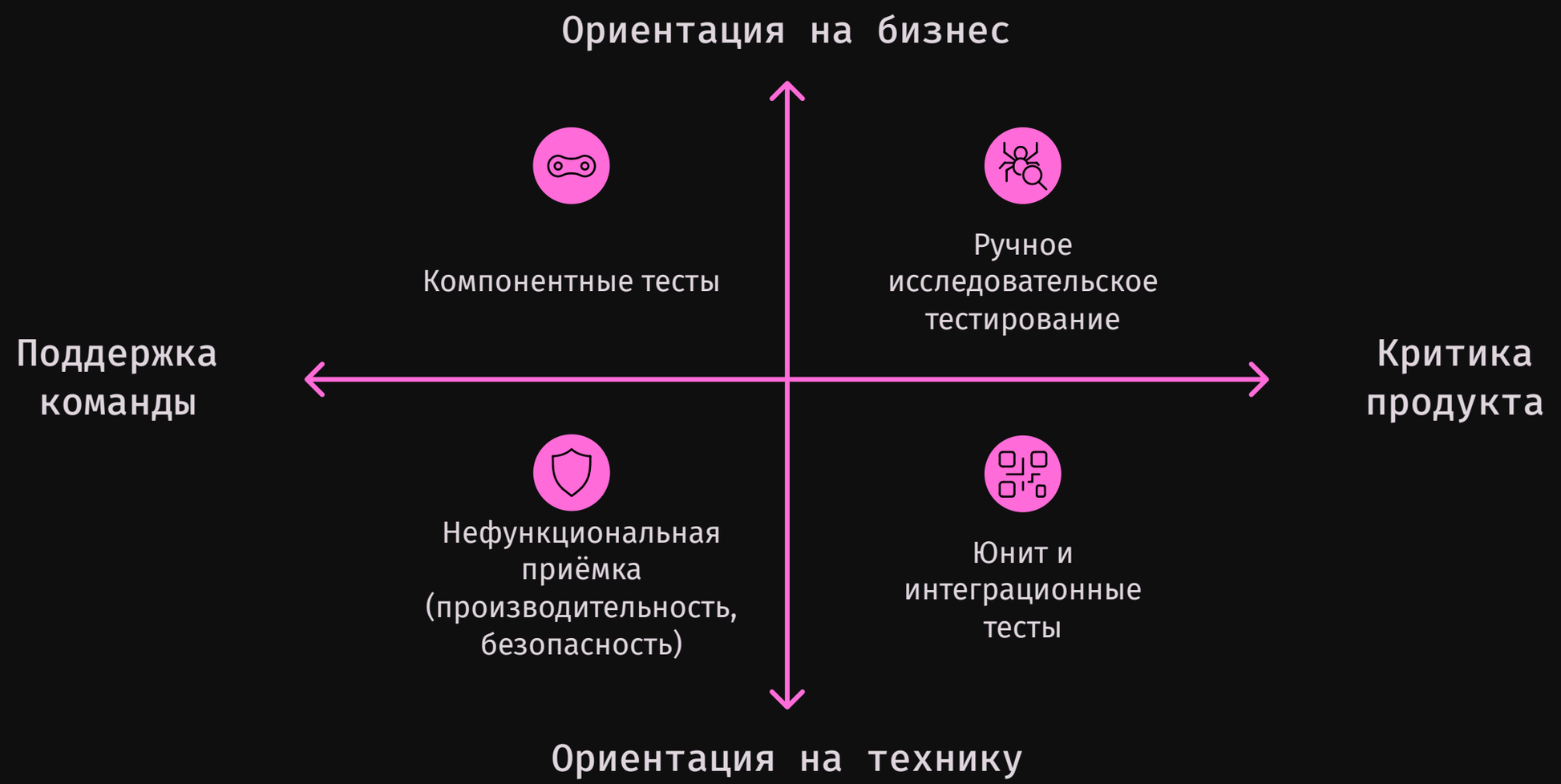
Модульные тесты (Unit)

Тестирование небольших частей кода (классов). **Много.**

Модульные тесты должны быть быстрыми, простыми и надежными, составляя основу всего тестового набора.

Тестовый квадрант

Квадрант Брайана Марика классифицирует тесты по целям (помощь в программировании vs. оценка приложения) и ориентации (бизнес-терминология vs. техническая терминология).



Глава IV

Тестирование контрактов (CDC)

Сложность тестирования микросервисов обусловлена необходимостью обеспечить корректное межсервисное взаимодействие (IPC). Тестирование контрактов позволяет проверить это взаимодействие без использования медленных сквозных тестов.

Контракт как соглашение

Каждое взаимодействие (потребитель — провайдер) является соглашением о структуре API или событий.

Суть CDC

Интеграционный тест провайдера, который гарантирует, что его API отвечает ожиданиям потребителя.

Проверка

Контрактные тесты проверяют HTTP-метод, путь, заголовки, тело запроса/ответа и код состояния.

Если команда провайдера нарушает контракт, тест в их конвейере CI/CD немедленно падает.

Реализация контрактов и асинхронное взаимодействие

Инструменты вроде Spring Cloud Contract или Pact облегчают внедрение Consumer-Driven Contracts. Контракт описывается с помощью DSL и используется для генерации тестов.

Инструментарий

- Spring Cloud Contract
- Pact Framework

Контракт DSL

Контракт (например, HTTP-запрос и ответ) используется для:

- Генерации тестов провайдера.
- Настройки макетов (заглушек) для тестов потребителя.

Контракты для обмена сообщениями

CDC также применим к асинхронному взаимодействию (издатель/подписчик). Контракт представляет собой пример доменного события.

→ Тест провайдера

Проверяет, что сгенерированное событие соответствует контракту.

→ Тест потребителя

Проверяет, что потребитель может корректно обработать данное событие.

Глава V & VI

Интеграция тестов в CI/CD конвейер

Автоматическая доставка кода требует поэтапного выполнения тестов, где время выполнения каждого набора увеличивается по мере приближения к промышленной среде.

1. Этап Pre-commit

Запуск модульных тестов разработчиком перед фиксацией кода.

2. Этап Commit

Компиляция, модульные тесты, статический анализ (должен быть очень быстрым).

3. Интеграционный этап

Выполнение более медленных интеграционных тестов, включая тесты контрактов.

4. Компонентный этап

Выполнение приемочных тестов для отдельного микросервиса.

5. Этап развертывания

Развертывание сервиса в промышленной среде.

> Ключевой приоритет: **Сообщить о непройденных тестах как можно скорее**, чтобы минимизировать стоимость исправления ошибки.