САГА О МИКРОСЕРВИСАХ

Издание в двух томах

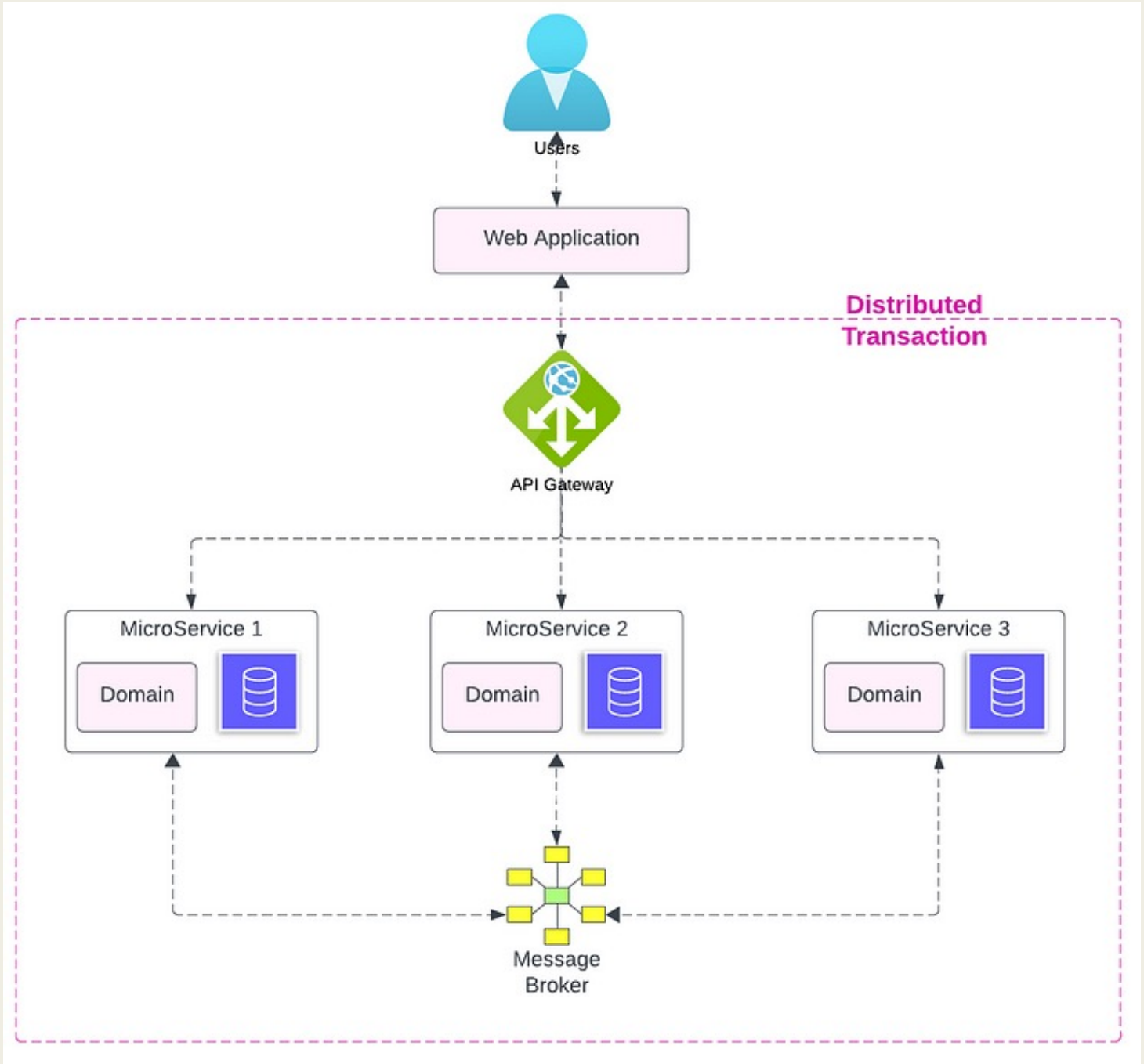
Обо мне

- Занимаюсь backend-разработкой на Java, Kotlin и Spring
- Люблю DevOps и задачи инфраструктуры
- Интересуюсь Linux, Docker, Kubernetes и сетями
- Спикер java-конференций и митапов

Рустам Курамшин
Senior Java Developer
Team/Tech Lead

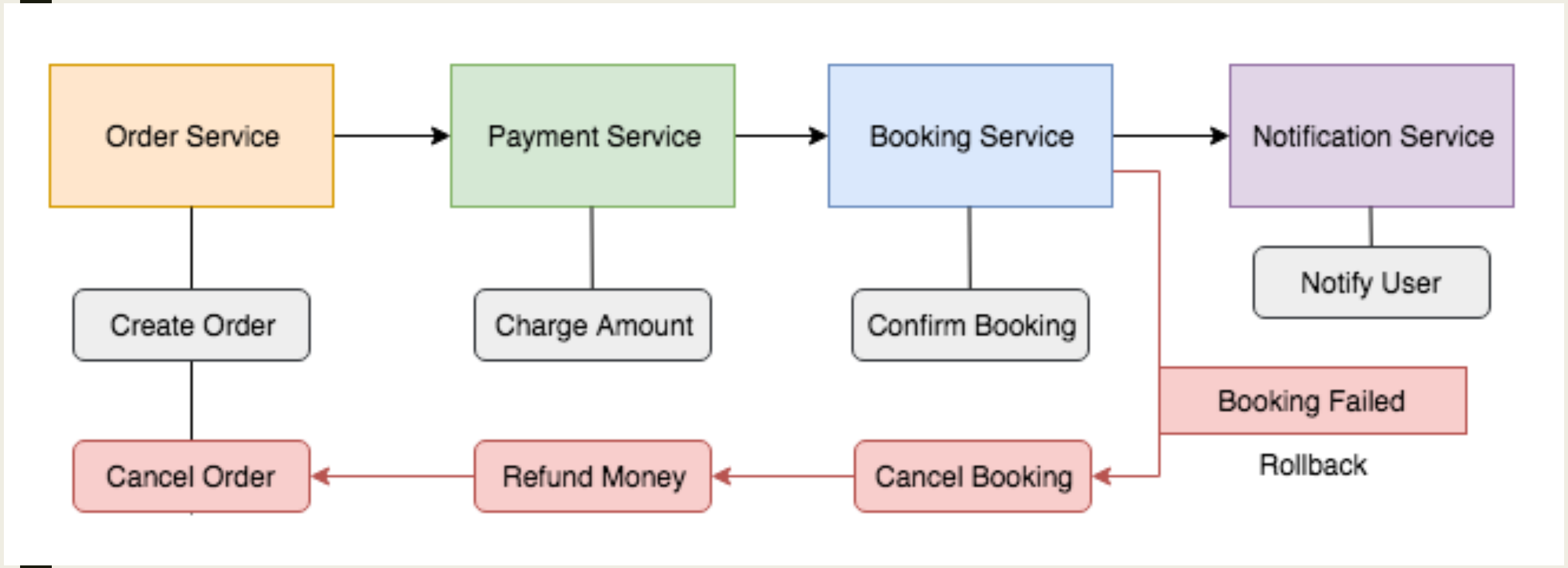
Проблемы распределённых транзакций

- Распределённая транзакция — это последовательность операций над **разными сервисами или БД**, которые должны выполняться как единое целое
- Классический @Transactional **не работает между микросервисами** — нет общего контекста или менеджера транзакций
- Примеры: **оформление заказа**, **бронирование билетов**, **банковский перевод между банками**
- Частичные сбои = **грязные данные** (оплата прошла, а товар не зарезервирован)
- Нет **глобального rollback** → нельзя откатить изменения во всех системах сразу


Users
Web Application
Distributed Transaction
API Gateway
MicroService 1
Domain
MicroService 2
Domain
MicroService 3
Domain
Message Broker

Что такое паттерн Saga

- **Saga** — это последовательность **локальных транзакций**, распределённых по сервисам
- Вместо глобального rollback выполняются **компенсационные действия**
- Каждая успешная операция сопровождается "откатным" действием
- Если один шаг провалился — запускаются компенсации всех предыдущих
- Подходит для **долгоживущих бизнес-процессов**


Order Service
Payment Service
Booking Service
Notification Service
Create Order
Charge Amount
Confirm Booking
Notify User
Booking Failed
Rollback
Cancel Order
Refund Money
Cancel Booking

Saga vs XA / 2PC

- **XA / 2PC** = глобальный коммит между сервисами (тяжёлый и не масштабируется)
- Требует блокировок и координации — плохо работает в распределённых системах
- **Saga** = проще, нет блокировок, нет глобального транзакционного менеджера
- Работает асинхронно, с компенсациями — **eventual consistency**
- Лучше подходит для **микросервисов и облачной архитектуры**

Варианты реализации Saga

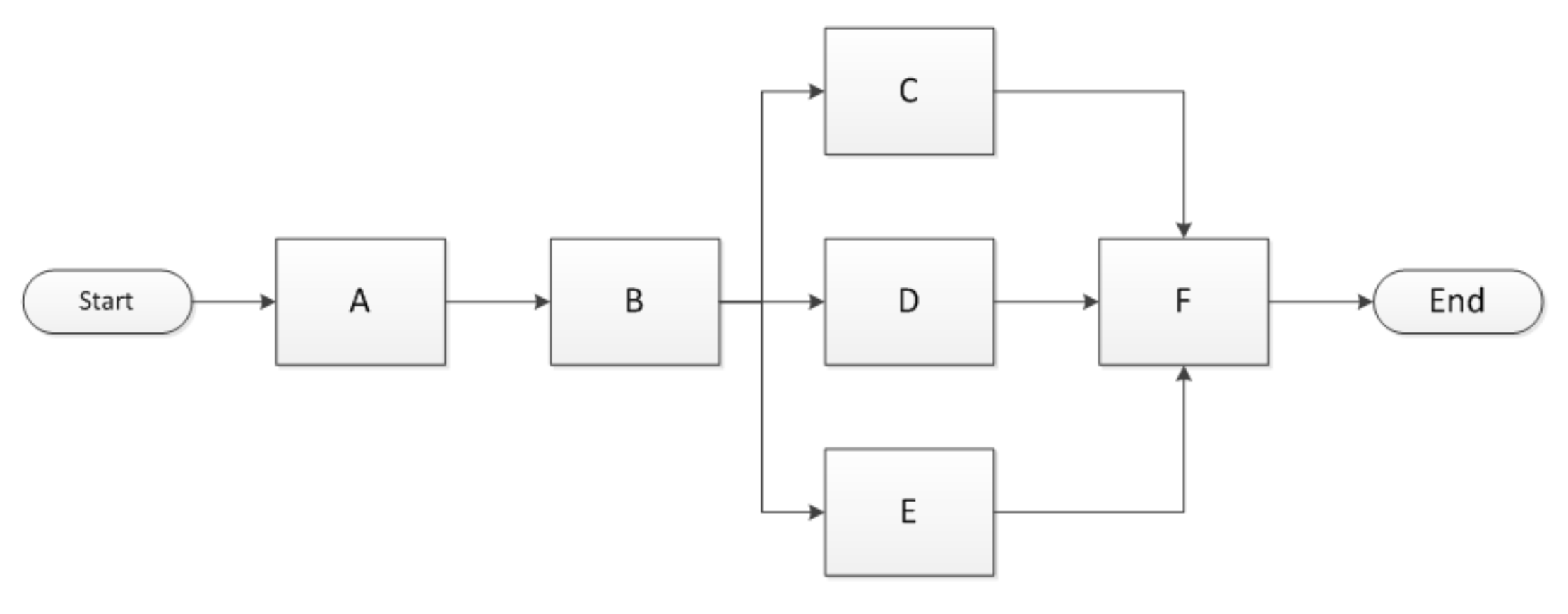
- **Оркестрация:** централизованный компонент (оркестратор) вызывает шаги саги по порядку
- **Хореография:** каждый участник сам слушает события и реагирует на них
- Оба подхода реализуют **компенсации** для откатов
- Выбор зависит от требований к управляемости, изолированности и масштабируемости
- Temporal, Camunda — примеры **оркестрации**, Kafka + event bus — примеры **хореографии**

Оркестрация vs Хореография

	Оркестрация	Хореография
Контроль	Централизованный	Распределённый
Читаемость	Высокая (код в одном месте)	Сложно проследить
Масштабируемость	Умеренная	Отличная
Изоляция сервисов	Слабо изолированы	Высокая
Тестируемость	Отличная (workflow как код)	Сложнее

Базовые понятия workflow-систем

- **Workflow** — описание бизнес-процесса: шаги, условия, ветвления
- **Activity** — отдельное действие: вызов внешнего API, отправка письма, запись в БД
- **DAG** (Directed Acyclic Graph) — порядок выполнения шагов без циклов
- **Состояние и идемпотентность** — важны для повторов, откатов и гарантий консистентности


Start
A
B
C
D
E
F
End

Инструменты для реализации на Java

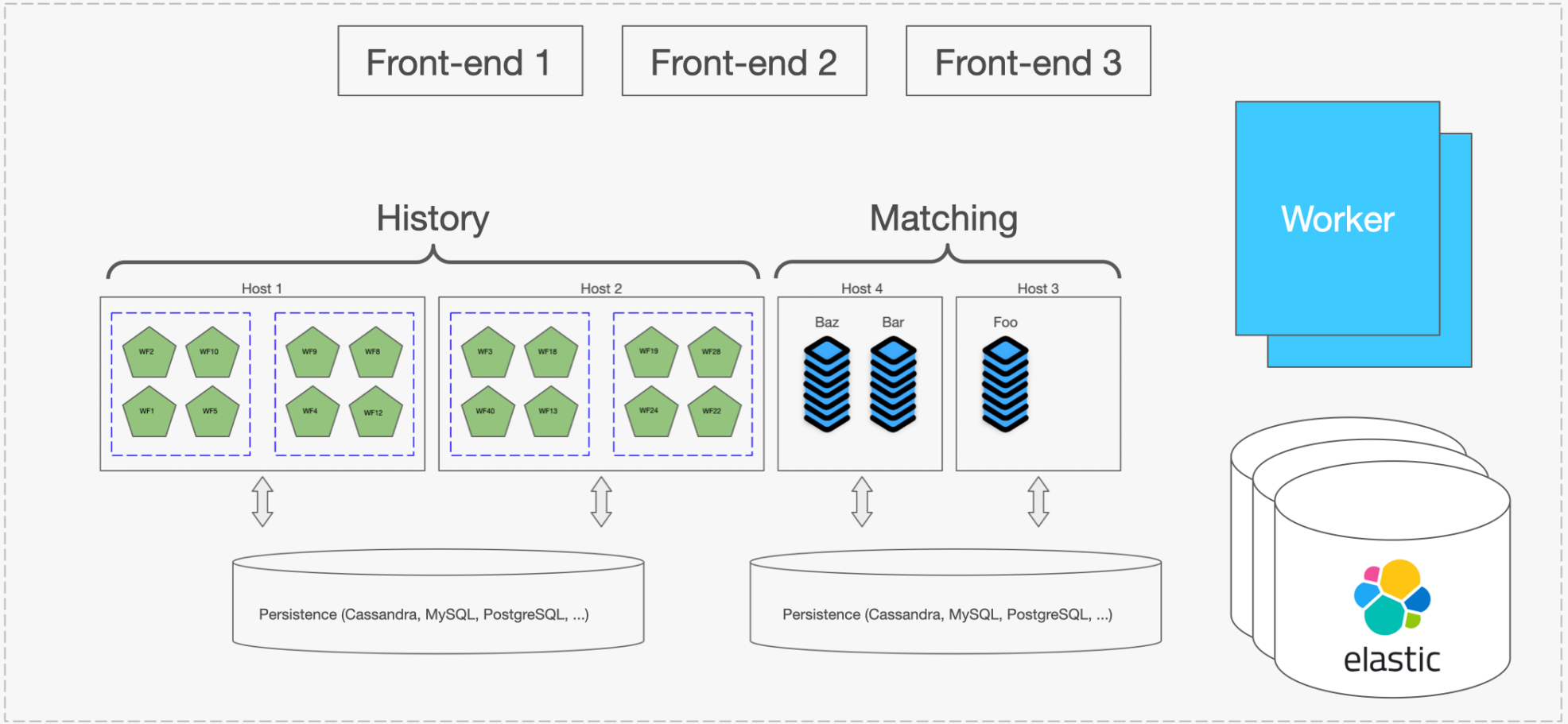
- **Camunda, Apache Airflow** — BPMN и pipeline-ориентированные движки, подходят для визуального моделирования процессов
- **Axon Framework, Eventuate** — ориентированы на event sourcing и CQRS, могут реализовывать Saga через события
- **Orkes / Netflix Conductor** — orchestration engine от Netflix, поддерживает REST и JSON workflow-описания
- **Temporal** — надёжный workflow engine с полноценной Java SDK, управление состоянием и retry "из коробки"
- Выбор зависит от архитектуры, требований к отказоустойчивости и удобству разработки

Архитектура Temporal

- **Temporal Server + Worker:** сервер управляет состоянием, воркер выполняет бизнес-логику
- **Workflow — это обычный Java-код**, описывающий последовательность действий
- Workflow **не исполняется напрямую**, а восстанавливается по событиям (**Event Sourcing**)
- Сервер хранит все события, воркер может быть перезапущен в любой момент
- Встроенные механизмы: **retry, timeout, compensation, monitoring**

Temporal

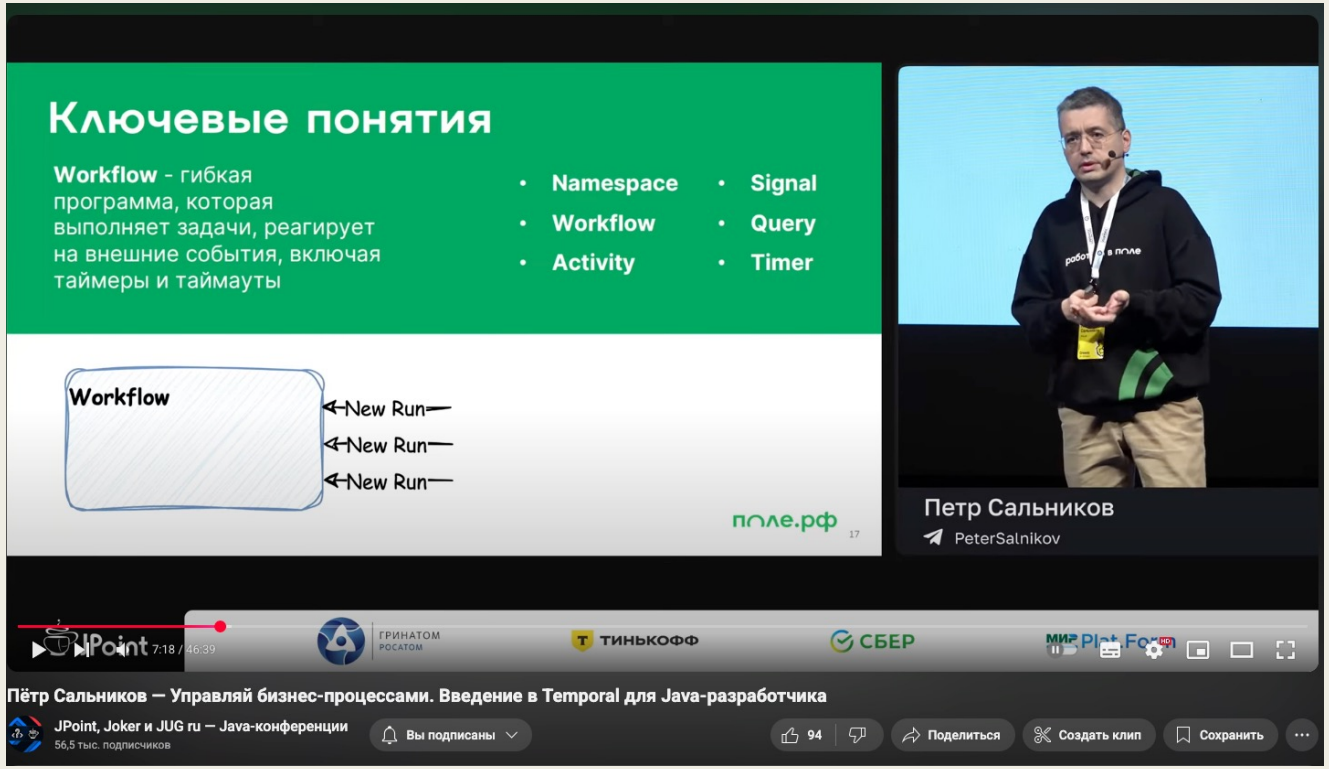
Temporal



Как Temporal выглядит в коде

- **Workflow-интерфейс** описывает бизнес-процесс (@WorkflowInterface)
- **Activity-интерфейсы** определяют действия (@ActivityInterface)
- **Worker** регистрирует workflow и activities, слушает задачи из task queue
- Всё взаимодействие через **обычный Java-код** — нет скриптов, DSL или XML
- Temporal сам обрабатывает retries, timeouts и сохранение состояния

Доклад Петра Сальникова на JPoint



Исходники проекта

Java hub