1. Введение в микросервисную архитектуру

Монолит vs. микросервисы. Монолитная архитектура – это когда всё приложение построено как единый целостный блок (все модули в одном приложении), который разворачивается и масштабируется целиком. В монолите любые изменения требуют перекомпиляции и деплоя всего приложения. Противоположно этому, микросервисная архитектура разбивает систему на множество небольших независимых сервисов, **каждый со своей задачей, логикой и, как правило, собственной базой данных** (Microservices vs. monolithic architecture | Atlassian) (Microservices vs. monolithic architecture | Atlassian). Эти микросервисы запускаются и деплоятся отдельно, взаимодействуя друг с другом через сетевые API. Например, компания Netflix одной из первых перешла от монолита к более чем тысяче микросервисов ради лучшей масштабируемости (Microservices vs. monolithic architecture | Atlassian).

(Microservices vs. monolithic architecture | Atlassian) *Принципиальная разница: при монолитной архитектуре все компоненты (например, модули оплаты, корзины, склада) работают в одном приложении на одном сервере (едином экземпляре), тогда как в микросервисной архитектуре каждый компонент вынесен в отдельный сервис, и клиент общается с ними через единую точку входа или напрямую по сети* (Microservices vs. monolithic architecture | Atlassian) (Microservices vs. monolithic architecture | Atlassian).

Преимущества микросервисов. Микросервисы позволяют отдельным командам работать автономно над разными сервисами и развёртывать изменения независимо друг от друга, не нарушая работу всей системы (Microservices vs. monolithic architecture | Atlassian) (Microservices vs. monolithic architecture | Atlassian). Это повышает **гибкость и скорость разработки** – обновления отдельных сервисов могут выкатываться часто (в крупных компаниях деплой может происходить ежедневно или даже несколько раз в день) (Microservices vs. monolithic architecture | Atlassian). **Масштабирование** упрощается: можно масштабировать только узкие места – запустить больше экземпляров нужного сервиса под повышенную нагрузку, не трогая остальные (Microservices vs. monolithic architecture | Atlassian). Также повышается **устойчивость системы**: сбой в одном микросервисе не «роняет» всё приложение целиком, остальные сервисы продолжают работу (Microservices vs. monolithic architecture | Atlassian). Команды разработчиков становятся более самостоятельными и могут выбирать оптимальный стек технологий для своего сервиса, что даёт **гибкость в выборе технологий** под каждую задачу (Microservices vs. monolithic architecture | Atlassian).

Недостатки микросервисов. За преимущества приходится платить увеличением **сложности инфраструктуры**. Когда приложение разрезано на десятки и сотни сервисов, сложнее отследить взаимосвязи, обеспечить согласованность данных и управлять всеми компонентами (Microservices vs. monolithic architecture | Atlassian). Появляются накладные расходы: нужно настроить **сеть и взаимодействие** между сервисами, контейнеризацию, оркестрацию, мониторинг каждого сервиса – всё это усложняет поддержку. Без должного управления микросервисы могут привести к «сползанию» сложности (development sprawl), когда рост числа сервисов замедляет разработку (Microservices vs. monolithic architecture | Atlassian). **Отладка и тестирование** распределённой системы тоже сложнее: запрос проходит через несколько сервисов, и нужно собирать логи с разных узлов, чтобы найти проблему (Microservices vs. monolithic architecture | Atlassian). Кроме того, дублирование инфраструктуры для каждого микросервиса (базы данных, пайплайны деплоя, мониторинг) может увеличить расходы на инфраструктуру (Microservices vs. monolithic architecture | Atlassian). Таким

образом, микросервисы подходят для сложных и масштабируемых систем, но требуют зрелых DevOps-практик и инфраструктуры.

2. Основные паттерны микросервисов

Микросервисная архитектура породила ряд типовых **паттернов (шаблонов проектирования)**, которые помогают организовать взаимодействие между сервисами. Рассмотрим ключевые паттерны: API Gateway, Aggregator, Service Discovery, Event-Driven (событийно-ориентированная архитектура) и Saga.

API Gateway (шлюз API)

API Gateway – это единая точка входа для всех клиентских запросов в систему микросервисов. Вместо того чтобы клиенты напрямую обращались к каждому сервису, они обращаются к шлюзу, а уже **API Gateway маршрутизирует запросы к нужным микросервисам** или даже объединяет ответы нескольких сервисов (Pattern: API Gateway / Backends for Frontends). Шлюз может выполнять дополнительные функции: аутентификацию, ограничение частоты запросов, кэширование, преобразование протоколов. По сути, он скрывает внутреннюю структуру микросервисов от внешних клиентов, предоставляя им единый интерфейс.

(The API gateway pattern versus the direct client-to-microservice communication - .NET | Microsoft Learn) *Схема с API Gateway: клиенты (мобильное приложение, SPA веб-приложение, браузер) посылают запросы в единый шлюз (в центре), который затем перенаправляет их на внутренние микросервисы (справа). Шлюз выступает в роли обратного прокси и разграничителя, позволяя добавить аутентификацию, балансировку нагрузки и другие кросс-срезочные функции* (The API gateway pattern versus the direct client-to-microservice communication - .NET | Microsoft Learn) (The API gateway pattern versus the direct client-to-microservice communication - .NET | Microsoft Learn).

Преимущество API Gateway – уменьшение «чата» между клиентом и множеством сервисов. Клиент делает **один запрос** на шлюз, вместо нескольких к разным сервисам (The API gateway pattern versus the direct client-to-microservice communication - .NET | Microsoft Learn). Шлюз может **агрегировать данные**: например, собрать информацию с трех разных сервисов и вернуть клиенту одним ответом (см. паттерн Aggregator ниже). Возможна и вариация **Backend for Frontend (BFF)** – отдельные API Gateway для разных видов клиентов (отдельно для веб, мобильного и пр.), чтобы адаптировать ответы под их нужды (The API gateway pattern versus the direct client-to-microservice communication - .NET | Microsoft Learn).

Важно правильно внедрить шлюз, чтобы он не превратился в узкое место или монолит. Иногда вместо одного шлюза делят на несколько по доменам или типам клиентов, чтобы избежать единой точки отказа и улучшить масштабируемость (The API gateway pattern versus the direct client-to-microservice communication - .NET | Microsoft Learn).

Aggregator (агрегатор)

Паттерн **Aggregator** (агрегатор) предполагает, что существует отдельный сервис-компонент, который получает запрос от клиента и **вызывает несколько микросервисов**, собирая (агрегируя) результаты, чтобы вернуть единый ответ. По сути, это композиция нескольких вызовов в одном месте. Часто такой подход используется внутри API Gateway или BFF: один запрос пользователя требует данных из разных сервисов, и агрегатор берет на себя координацию этих вызовов.

Зачем нужен агрегатор? Он снижает количество сетевых вызовов от клиента. Вместо того чтобы клиент делал подряд, скажем, три запроса (к сервису товаров, сервису цен и сервису отзывов), агрегатор сделает эти три вызова внутри сети и вернёт клиенту сразу объединённый результат. Это особенно важно при медленных или дорогих соединениях (например, мобильный интернет) (Gateway Aggregation pattern - Azure Architecture Center | Microsoft Learn) (Gateway Aggregation pattern - Azure Architecture Center | Microsoft Learn). В агрегатор можно заложить параллельное выполнение запросов к сервисам, что ускоряет получение ответа.

(Gateway Aggregation pattern - Azure Architecture Center | Microsoft Learn) *Принцип работы паттерна Aggregator: приложение (снизу) делает один запрос (1) к шлюзу/агрегатору. Агрегатор разбивает его на несколько запросов к разным сервисам (стрелки 2 -> Service 1, Service 2, Service 3), получает от каждого ответ (3 <-) и формирует единый ответ, который возвращает приложению (4)* (Gateway Aggregation pattern - Azure Architecture Center | Microsoft Learn).

Aggregator-паттерн часто реализуется как часть API Gateway. Например, при открытии страницы профиля пользователя, шлюз может сходить и в сервис пользователей, и в сервис уведомлений, и в сервис статистики, а затем вернуть объединённые данные. **Важно** при этом не нарушать независимость сервисов – агрегатор должен вызывать их через публичные API, не вмешиваясь во внутреннюю логику, и не создавать избыточной связности.

Service Discovery (обнаружение сервисов)

В динамической среде, где микросервисы могут масштабироваться (добавляться/удаляться) и менять свои адреса (например, при деплое в контейнерах), возникает задача – **найти нужный сервис** по имени. *Service Discovery* – это паттерн, обеспечивающий регистрацию сервисов и поиск их местоположения (адресов) другими компонентами.

Существует две модели **обнаружения сервисов**: клиентская и серверная. В **клиентской модели** сам клиент (или шлюз) знает, где находится сервис: он обращается к специальному *реестру сервисов* (service registry), получает актуальный адрес экземпляра и напрямую вызывает сервис. Во **встроенной в сеть (серверной) модели** клиент посылает запрос на фиксированный адрес (*маршрутизатор* или балансировщик), а тот сам смотрит в реестр и перенаправляет запрос на один из доступных экземпляров сервиса (Pattern: Server-side service discovery). Оба подхода требуют компонента **Service Registry** – базы данных, где сервисы регистрируются при запуске (со своим адресом и портом) и откуда удаляются при отключении.

Например, сервис авторизации поднялся на контейнере и зарегистрировался (через консоль или API) в реестре (например, Consul, Eureka) под именем "AuthService" с адресом `10.0.1.5:5000`. Другой сервис, желая обратиться к AuthService, спросит реестр: "дай адрес AuthService" и получит `10.0.1.5:5000`, после чего выполнит запрос. В случае серверного discovery эту роль может выполнять балансировщик (или API Gateway), скрывая от клиента детали. **Итог** – discovery позволяет сервисам находить друг друга **динамически** в среде, где их адреса не фиксированы, что необходимо для автоматического масштабирования и устойчивости системы (Pattern: Server-side service discovery) (Pattern: Server-side service discovery).

(Pattern: Server-side service discovery) *Схема Server-side Service Discovery: клиент (слева) обращается с запросом к роутеру/балансировщику, который выступает в роли посредника. Роутер сначала делает запрос к реестру сервисов (внизу) чтобы узнать адреса доступных экземпляров нужного сервиса (query -> Service Registry), затем перенаправляет запрос клиента на один из экземпляров (load balance ->*

Service Instance A/B/C). Сами сервисы при запуске регистрируются в реестре (register -> Service Registry). *Примечание:* В клиентской модели роутер может отсутствовать – тогда Discovery-клиент (вшитый в приложение) сам запрашивает реестр и идёт прямо на сервис.

Event-Driven Architecture (событийно-ориентированная архитектура)

Событийно-ориентированная архитектура – это подход, при котором сервисы обмениваются сообщениями о наступлении событий асинхронно, вместо прямых синхронных запросов. Когда в одном микросервисе происходит что-то значимое (изменение данных, действие пользователя), он публикует **событие** (event) – сообщение в некий канал (например, брокер сообщений). Другие микросервисы, подписанные на этот тип событий, получают уведомление и реагируют (например, обновляют свои данные или выполняют какое-то действие) (Pattern: Event-driven architecture).

Ключевые компоненты EDA (Event-Driven Architecture): **поставщики событий** (event producers), **потребители событий** (event consumers) и **канал доставки событий** (event bus/message broker) (Event-driven architecture style - Azure Architecture Center | Microsoft Learn). В роли канала обычно выступают очереди сообщений или системы потоковой обработки (Kafka, RabbitMQ, NATS и пр.). Публикация события обычно *не ждёт* ответа – сервис-отправитель просто уведомляет, что событие случилось, и продолжает работу. Потребители обрабатывают событие в своём темпе, обеспечивая **слабую связанность**: отправитель не знает, кто его слушает (или есть ли слушатели вообще) (Event-driven architecture style - Azure Architecture Center | Microsoft Learn).

Плюсы событийной архитектуры: высокая асинхронность и разгрузка – сервисы не блокируются в ожидании ответа друг от друга, что повышает производительность и устойчивость. Также это позволяет достичь **eventual consistency** (конечной согласованности) данных между сервисами вместо глобальных транзакций. Например, при оформлении заказа Order-сервис публикует событие "Order Created". Подписанные на него Inventory-сервис и Billing-сервис получают событие: первый резервирует товары на складе, второй блокирует оплату. Если какой-то шаг не прошёл, соответствующие сервисы могут опубликовать событие об ошибке (или непрохождении), на которое другие отреагируют, отменяя свои действия. Таким образом обеспечивается согласованность через обмен событиями (паттерн Saga, см. ниже) без двухфазных коммитов.

(Implementing event-based communication between microservices (integration events) - .NET | Microsoft Learn) *Иллюстрация событийно-ориентированной архитектуры: один сервис (Catalog) публикует событие* `PriceUpdated` *(после изменения цены товара) в общий* ***event bus*** *(серый прямоугольник по центру), и сразу завершает свою локальную транзакцию. Другие сервисы (Basket, и любые дополнительные) подписаны на событие* `PriceUpdated` *– брокер доставляет им копии сообщения. Эти потребители независимо друг от друга выполняют свои действия (например, Basket-сервис пересчитывает цены в корзинах клиентов). Все взаимодействие происходит асинхронно через посредника, сервисы не вызывают напрямую API друг друга* (Pattern: Event-driven architecture) (Implementing event-based communication between microservices (integration events) - .NET | Microsoft Learn).

Такой подход значительно **декуплирует** (развязывает) сервисы: отправитель события не зависит от того, сколько получателей и кто они – можно добавлять новых подписчиков без изменений в коде источника. Минус – усложняется обработка ошибок и отладка, так как поток управления распределён. Тем не менее, событийная архитектура лежит в основе паттерна Saga и масштабируемых систем, где важно реагировать на изменения в режиме реального времени.

Saga (шаблон саги: хореография vs оркестрация)

Проблема: В монолите транзакции охватывают сразу всё приложение (ACID-транзакция в одной базе). В микросервисах данные распределены по разным сервисам, и выполнить одну классическую транзакцию на весь бизнес-процесс невозможно. Например, оформление заказа затрагивает сервис заказа, сервис платежа и сервис товара – как обеспечить **атомарность** и согласованность, если каждый сервис имеет свою БД? Паттерн **Saga** решает эту задачу.

Saga – это **набор локальных транзакций, выполняемых последовательно в разных сервисах, связанных единым бизнес-процессом, с компенсациями в случае неудач** (Pattern: Saga). Вместо глобального «коммита» все участники саги выполняют свои операции по очереди, передавая эстафету следующему шагу. Если какой-то шаг не удался (нарушены бизнес-правила, недостаточно средств и т.п.), запускается последовательность обратных действий (compensating transactions), которые отменяют уже выполненные предыдущие шаги, возвращая систему в консистентное состояние (Pattern: Saga).

Есть два основных способа координации шагов саги: **хореография (choreography)** и **оркестрация (orchestration)** (Pattern: Saga). При **хореографии** нет центрального контроллера – каждый сервис после своей локальной транзакции просто публикует **событие** (доменное событие) о результате, и другие сервисы, подписанные на это событие, сами решают, что делать дальше. По цепочке событий процесс протекает, пока все шаги не завершатся. Это распределенный, событийный подход – продолжение идеи event-driven. Пример: Order-сервис создает заказ в статусе "PENDING" и выдает событие "Order Created". Payment-сервис услышал это событие – пытается снять оплату, затем публикует событие "Payment Approved" или "Payment Declined". Order-сервис ловит второе событие и либо помечает заказ "APPROVED", либо "CANCELED" – сага завершена (Pattern: Saga). Здесь каждый участник «танцует» без дирижера, отсюда термин "хореография". Он прост в реализации (использует уже существующее событие взаимодействие), но сложен в сопровождении при большом количестве участников – сложно отследить общий поток, возможны циклические вызовы событий.

При **оркестрации** вводится специальный компонент – **оркестратор саги** (saga orchestrator), который знает сценарий и дирижирует выполнением. Оркестратор (может быть реализован как отдельный сервис или даже как часть одного из существующих) отправляет **команды** каждому сервису по очереди и ждёт ответа/результата, решая, что делать дальше. Он как централизованный менеджер: стартует процесс, вызывает Service A (тот делает локальную транзакцию), потом оркестратор вызывает Service B и т.д. Если какой-то сервис вернул ошибку, оркестратор запускает команды компенсации в обратном порядке для тех, кто уже выполнил работу (Pattern: Saga). Преимущество – полный контроль и видимость последовательности, легче реализовать сложную логику ветвлений, таймаутов, повторов. Недостаток – оркестратор становится точкой отказа и требующим внимания компонентом; к тому же логика распределённой транзакции концентрируется в одном месте (может напоминать монолит по замыслу, хотя и распределённый по исполнению).

(Saga orchestration pattern - AWS Prescriptive Guidance) *Сага с оркестратором: слева компонент "Saga orchestrator" содержит сценарий (шаги T1, T2, T3 – зеленые для успешных действий, и шаги C1, C2 – желтые для компенсирующих действий при отмене). Оркестратор последовательно инициирует: T1 – создание заказа в Order Service (синий), T2 – резервирование товара в Inventory Service (зелёный), T3 – списание средств в Payment Service (жёлтый). Если на шаге T3 происходит сбой (красный крестик на "Make payment"), оркестратор запускает компенсации: C1 отменяет резерв товара, C2 отменяет заказ. Таким образом все локальные транзакции, выполненные до сбоя, откатываются компенсирующими транзакциями* (Pattern: Saga) (Pattern: Saga).

В итоге, паттерн Saga обеспечивает **согласованность данных в распределённой системе** без блокирующих глобальных транзакций. Хореография проще, когда участников мало и их взаимодействие несложное. Оркестрация лучше контролируется, но требует выделенного управляющего компонента. В реальных системах выбор подхода зависит от конкретных требований и масштаба – иногда комбинируют оба (например, несколько мелких саг-хореографий оркестрируются более крупным процессом).

3. Разбор архитектуры проекта

Рассмотрим архитектуру учебного проекта, состоящего из нескольких микросервисов. В нашем примере есть два основных сервиса и API Gateway на базе Nginx:

- **LostFoundService** – сервис объявлений о пропаже и находке вещей (условно, сервис для публикации потерянных и найденных предметов).
- **AuthService** – сервис аутентификации и управления пользователями.
- **Nginx** – веб-сервер, выполняющий роль API Gateway (reverse proxy) для маршрутизации запросов клиентов к микросервисам.

Структура проекта. Каждый сервис – это отдельное веб-приложение (в нашем случае на FastAPI), со своим REST API. Предположим, LostFoundService содержит эндпойнты для создания объявления о потерянной вещи, отметки о найденной вещи, поиска по объявлениям и т.д. AuthService обеспечивает регистрацию пользователей, вход (логин) и выдачу токенов (например, JWT) для аутентификации. Оба сервиса запускаются независимо (например, каждый на своём порту). Nginx сконфигурирован так, чтобы принимать все входящие HTTP-запросы и проксировать их дальше: запросы на адреса, связанные с авторизацией, пересылаются на AuthService, а запросы к функционалу потерянных вещей – на LostFoundService.

Взаимодействие сервисов. Сервисы в этой архитектуре общаются в основном через **HTTP-запросы** через Nginx. Например, клиент (скажем, браузер фронтенд-приложения) делает POST-запрос на `/api/auth/login` с учетными данными. Nginx принимает этот запрос на внешнем порту (например, 80) и проксирует его на внутренний адрес AuthService (например, `http://auth-service:8000/login`). AuthService проверяет логин/пароль, генерирует JWT-токен и возвращает его. Далее, когда клиент хочет добавить объявление о пропаже (`POST /api/lostfound/items`), он включает полученный JWT в заголовок Authorization. Nginx перенаправляет запрос на LostFoundService (`http://lostfound-service:8000/items`). LostFoundService самостоятельно проверяет JWT – либо путем его верификации (если он знает секрет подписи токена), либо запрашивая AuthService (реже, чтобы не создавать сильную связность, чаще каждый сервис валидирует токен локально). После проверки авторизации LostFoundService выполняет свою логику (сохраняет запись о пропаже в своей базе) и отвечает клиенту.

Важно, что **между сервисами нет прямых вызовов к базам друг друга** – каждый работает только со своей базой данных, а обмен данными – через публичные API. Это соответствует принципу “Database per service” и обеспечивает слабую связанность. Если LostFoundService нужно узнать данные о пользователе, он должен запросить их у AuthService через API (или кешировать после получения при аутентификации).

Таким образом, Nginx в этой архитектуре выполняет роль **точки входа** (как API Gateway): клиенты обращаются всегда к нему, а он уже разруливает на нужный микросервис. Сами микросервисы могут также взаимодействовать друг с другом через HTTP (в обход шлюза, по внутренней сети Docker, если

настроить), но в нашем простом проекте такой сценарий не обязателен – достаточно того, что клиент координирует взаимодействие, дергая нужные сервисы через gateway.

Логически, можно изобразить эту архитектуру так: **Клиент** → (HTTP) → **Nginx (API Gateway)** → **AuthService** (для auth-запросов) **или** → **LostFoundService** (для функциональных запросов). Также, AuthService и LostFoundService могут посылать события друг другу (например, по Kafka) или прямые запросы при необходимости, но в базовой реализации они отделены.

4. Настройка API Gateway с Nginx

В проекте в качестве API Gateway используется **Nginx** – высокопроизводительный веб-сервер и реверс-прокси. Он слушает HTTP-запросы от клиентов на внешнем порту (например, 80) и на основе URL определяет, в какой сервис проксировать конкретный запрос.

Роль Nginx. Nginx действует как *обратный прокси* перед микросервисами. Он может обеспечивать единый домен для всех сервисов (чтобы не было необходимости обращаться на разные порты или хосты для разных частей API). Кроме того, Nginx может выполнять балансировку нагрузки, терминирование SSL (принимать HTTPS и слать внутрь HTTP), кэшировать ответы, внедрять заголовки и пр. В контексте нашего учебного проекта основная роль – маршрутизация: раздать запросы по нужным сервисам.

Конфигурация `nginx.conf`. В конфигурации Nginx настраиваются **location**-блоки для проксирования. Примерно это выглядит так (упрощенный вариант):

```
http {
    server {
        listen 80;
        server_name localhost;

        location /api/auth/ {
            proxy_pass http://auth-service:8000/;
        }

        location /api/lostfound/ {
            proxy_pass http://lostfound-service:8000/;
        }
    }
}
```

Здесь `auth-service` и `lostfound-service` – DNS-имена сервисов в сети Docker (или хосты, если на отдельных серверах). Так, запрос к `http://localhost/api/auth/login` пойдет на `http://auth-service:8000/login` внутри Docker-сети. Если в проекте предусмотрена документация Swagger/UI для каждого сервиса, можно проксировать и её (например, `/api/lostfound/docs` → `lostfound-service:8000/docs`).

Особенности проксирования FastAPI с подпенями (prefix). При использовании FastAPI (а оба сервиса, предположим, на нём), возникает нюанс: когда приложение размещено не в корне домена, а по пути (например, `/api/lostfound`), нужно сообщить FastAPI об этом "префиксе". Это делается с помощью параметра `root_path` при создании приложения FastAPI. Например: `app =`

`FastAPI(root_path="/api/lostfound")`. Также, если используется Uvicorn/Hypercorn, ему тоже может потребоваться указать `--root-path`. Если этого не сделать, то при открытии документации Swagger через шлюз, ссылки на эндпойнты могут строиться неправильно (они будут от корня, без `/api/lostfound`, и Nginx может не найти такие пути).

В нашем проекте можно настроить FastAPI так, чтобы он знал о префиксе. Как указывают разработчики, и приложение FastAPI, и сервер Uvicorn должны получить параметр root_path для корректной работы под прокси (Root path is applied 2 times when using root_path · fastapi fastapi · Discussion #9018 · GitHub). Например:

```
app = FastAPI(title="Lost&Found Service", root_path="/api/lostfound")
...
uvicorn.run(app, host="0.0.0.0", port=8000, root_path="/api/lostfound")
```

Тогда FastAPI будет генерировать правильные пути в `/docs` и при редиректах, учитывая префикс `/api/lostfound`.

Проблемы и решения. Одна из распространённых проблем – **несовпадение путей**. Если Nginx проксирует на сервис, добавляя или убирая часть пути, может нарушиться разрешение URL в самом сервисе. Решение – использовать директивы `sub_filter` или настроить корректно `proxy_pass` (например, ставить слэш в конце, как в примере выше, чтобы путь после `/api/lostfound/` передавался внутрь). Также важно настроить заголовки Host, X-Forwarded-For и т.д. Nginx обычно автоматически прокидывает Host, но если нужны защищенные маршруты, может пригодиться `X-Forwarded-Prefix` для информирования сервиса о префиксе. Однако в случае FastAPI это избыточно, достаточно `root_path`.

Второй момент – **раздача статики или файлов**. Если микросервис должен отдавать, скажем, загруженные изображения, можно либо настроить Nginx на раздачу этих файлов напрямую (если они доступны ему через volume), либо проксировать и их. В учебном проекте, вероятно, статика минимальна, поэтому Nginx просто проксирует всё под определённым префиксом.

Итак, Nginx значительно упрощает клиентам работу с нашей системой: все запросы идут на единый хост/порт, а микросервисы могут свободно меняться за шлюзом. Мы получаем гибкость в передаче запросов и возможность центрально реализовать некоторые аспекты безопасности и логирования.

5. Использование Docker Compose

Для запуска всех компонентов системы вместе используется **Docker Compose** – инструмент, позволяющий описать многосоставное приложение (сервисы, сети, тома) в одном YAML-файле и запускать его одной командой.

В `docker-compose.yml` нашего проекта, вероятно, определены три сервиса: `lostfound-service`, `auth-service` и `nginx` (gateway). Простейший пример такого файла:

```
version: "3.8"
services:
  auth-service:
```

```
    build: ./AuthService  # путь к Dockerfile AuthService
    ports:
      - "8001:8000"
    networks:
      - backend

  lostfound-service:
    build: ./LostFoundService
    ports:
      - "8002:8000"
    networks:
      - backend

  nginx:
    image: nginx:1.21
    ports:
      - "80:80"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - auth-service
      - lostfound-service
    networks:
      - backend

networks:
  backend:
    driver: bridge
```

Как это работает. Compose создаёт общую *сеть* (в данном случае с именем `backend`), к которой подключаются все три контейнера. Благодаря этому они могут обращаться друг к другу по сервисным именам как по DNS. Например, внутри контейнера Nginx hostname `auth-service` будет резолвиться в IP-адрес контейнера AuthService (Precedence of DNS entry vs. Compose service name). Это избавляет от необходимости хардкодить IP – Docker Compose и его внутренняя DNS-служба управляют именами. Если в конфиге Nginx мы указали `proxy_pass http://auth-service:8000`, то в рантайме Nginx правильно найдёт контейнер AuthService.

Compose-файл также монтирует конфиг Nginx из нашего проекта в контейнер (`volumes:` строка) и пробрасывает порты: 80 наружу. Аналогично, для отладки, мы открыли порты 8001 и 8002 для сервисов, чтобы можно было обращаться к ним напрямую во время разработки (необязательно в продакшене). Опция `depends_on` указывает, что Nginx должен запускаться после того, как поднимутся указанные сервисы (Auth и LostFound), хотя это не даёт гарантии готовности сервисов, но в нашем случае это ок (они поднимутся быстро).

Запуск и взаимодействие. Разработчик (или автоматизация CI/CD) запускает: `docker-compose up -d --build`. Docker Compose билдит образы для наших сервисов (если указана директива build) и запускает контейнеры. Все сервисы подключаются к сети `backend`. Теперь, когда Nginx внутри контейнера начнёт работать, он сразу сможет проксировать на `http://auth-service:8000` и `http://lostfound-service:8000` – эти имена ему известны через DNS. Клиентские приложения могут обращаться к `http://<host>:80/api/...` – и получать ответы через gateway.

Compose также облегчает локальное тестирование: разработчик может поднять всю систему на своей машине ровно так же, как она будет работать в продакшене. Отдельные сервисы можно перезапускать без остановки всей системы (`docker-compose restart auth-service`). Логи всех сервисов можно собрать через `docker-compose logs`.

Стоит отметить, что Compose в продакшене часто заменяется более продвинутыми оркестраторами (Kubernetes, Swarm), но принципы остаются теми же: сервисы объявлены, сети созданы, имена резолвятся, можно масштабировать (в Compose масштабирование ограничено, но есть `docker-compose up --scale` для увеличения числа экземпляров сервиса).

В нашем небольшом проекте Docker Compose обеспечивает простую **контейнеризацию**: каждый микросервис упакован в Docker-образ с нужными зависимостями, а Compose связывает их в единое целое. Это устраняет проблемы «на моей машине не работает», поскольку окружение унифицировано, и упрощает деплой – один файл описывает всю систему.

6. Контрольные вопросы и ответы

1. **В чём разница между монолитной и микросервисной архитектурой?**
   Ответ: Монолитная архитектура представляет собой единое приложение, где все модули связаны и разворачиваются вместе. Микросервисная архитектура разбивает систему на набор мелких независимых сервисов, каждый из которых выполняет свою функцию и может развёртываться отдельно. В микросервисах взаимодействие происходит по сети через API, тогда как в монолите модули общаются через вызовы внутри одного приложения.

2. **Назовите три преимущества микросервисной архитектуры и три её недостатка.**
   Ответ: Преимущества: (1) **Масштабируемость** – каждый сервис можно масштабировать отдельно (горизонтально увеличивая нужные мощности). (2) **Быстрая разработка и деплой** – небольшие команды могут независимо развёртывать обновления своих сервисов, не дожидаясь релиза всего приложения. (3) **Устойчивость** – сбой одного микросервиса не обрушивает всю систему, остальные сервисы продолжают работать. Недостатки: (1) **Сложность инфраструктуры** – требуется настроить взаимодействие множества сервисов (оркестрация, мониторинг, логирование, сеть). (2) **Повышенные накладные расходы** – больше ресурсов уходит на поддержание контейнеров, сетевого взаимодействия, дублирование механизмов (каждый сервис имеет свое хранение, тесты, deployment pipeline). (3) **Трудность отладки и согласованности** – сложнее отслеживать поток выполнения, реализовывать транзакции, а баги диагностировать, собирая информацию с разных сервисов.

3. **Что такое паттерн API Gateway и зачем он нужен?**
   Ответ: API Gateway – это шлюз, единая точка входа в систему микросервисов. Клиенты посылают все запросы на API Gateway, а он уже маршрутизирует их к нужному микросервису. Это упрощает клиентам взаимодействие (не нужно знать адрес каждого сервиса) и позволяет реализовать общие функции – например, проверку токена, лимитирование запросов, кэш – в одном месте. Также API Gateway может агрегировать данные от нескольких сервисов и возвращать их как единый ответ.

4. **Как работает паттерн Aggregator в микросервисах?**
   Ответ: Паттерн Aggregator предполагает, что специальный сервис (или компонент) получает запрос, затем вызывает несколько других микросервисов, собирает результаты и формирует единый ответ. По сути, агрегатор инкапсулирует логику обращения к нескольким сервисам.

Например, запрос на страницу продукта может вызвать сервис товаров, сервис цен и сервис отзывов, собрать данные и вернуть клиенту за один раз. Это снижает количество запросов от клиента и оптимизирует трафик, особенно в условиях высокой задержки.

5. **Для чего нужен механизм Service Discovery (обнаружение сервисов) в микросервисной архитектуре?**
   Ответ: Service Discovery нужен, чтобы микросервисы могли находить друг друга динамически. В среде, где сервисы постоянно поднимаются, перезапускаются или масштабируются (меняя IP-адреса или порты), реестр сервисов хранит актуальные адреса. Клиенты или API Gateway могут запрашивать реестр, чтобы узнать, куда слать запрос для данного сервисного имени. Это автоматизирует конфигурацию – не нужно вручную прописывать адреса каждого экземпляра. Проще говоря, discovery – это “телефонный справочник” сервисов в инфраструктуре.

6. **Что означает событийно-ориентированная архитектура (Event-Driven Architecture)?**
   Ответ: В событийно-ориентированной архитектуре компоненты общаются посредством событий – сообщения рассылаются при наступлении определённых ситуаций. Вместо прямых запросов “вызови сервис B”, сервис A публикует событие (например, "OrderCreated") на общий *шиной* или брокере сообщений. Заинтересованные сервисы (подписчики) получают это событие и реагируют (например, один сервис резервирует товары, другой отправляет email). Такая архитектура позволяет асинхронно обновлять состояние системы и добиться согласованности без жёстких связей: отправитель не ждёт ответа и не знает, кто обработал событие, всё происходит через посредника – очередь или поток событий.

7. **Объясните паттерн Saga и разницу между подходами хореографии и оркестрации.**
   Ответ: Saga – это способ выполнить распределённую транзакцию без двухфазного коммита, разделив её на серию локальных транзакций в разных сервисах. Если все шаги проходят успешно, сага завершается, если какой-то шаг не удался – выполняются компенсирующие действия для отмены уже выполненных операций. *Хореография* – когда нет центрального контроллера: каждый сервис просто публикует событие, что он сделал, и следующий сервис слушает эти события, решая, что делать (цепочка событий). Это распределённое управление, каждая служба сама знает логику следующего шага. *Оркестрация* – когда есть единый оркестратор (сервис или компонент), который по заранее прописанному сценарию рассылает команды сервисам: "сделай это", "а теперь следующий шаг тому сервису" и т.д., и собирает ответы. При оркестрации вся логика последовательности сконцентрирована в одном месте (у оркестратора), а при хореографии – размазана по подпискам сервисов на события друг друга.

8. **Какие сервисы входят в пример проекта и как они взаимодействуют между собой?**
   Ответ: В учебном проекте три основных компонента: AuthService (сервис аутентификации), LostFoundService (сервис объявлений потерянных/найденных вещей) и Nginx (API Gateway). Клиенты посылают HTTP-запросы к Nginx. Nginx по URL определяет, куда отправить запрос: запросы, связанные с авторизацией (регистрация, логин), он перенаправляет на AuthService; запросы к функционалу объявлений – на LostFoundService. AuthService при успешном логине выдает JWT-токен клиенту. Клиент затем использует этот токен при обращении к LostFoundService (через Nginx), а LostFoundService проверяет токен (сам или через AuthService) для разрешения операции. Прямых вызовов между LostFoundService и AuthService в данной архитектуре может не быть – они связаны через общий шлюз и через токены (т.е. через уровень аутентификации клиента). Таким образом, взаимодействие происходит по принципу «клиент -> gateway -> нужный сервис» в соответствии с маршрутизацией.

9. **Какова роль Nginx в этой микросервисной архитектуре?**
   Ответ: Nginx выступает в роли API Gateway (шлюза) и реверс-прокси. Он обеспечивает единый входной адрес для всех запросов к системе. Роль Nginx – принять запрос от внешнего клиента и на основании правила (префикса пути, хоста и т.д.) переслать этот запрос к соответствующему микросервису. Помимо маршрутизации, Nginx может выполнять дополнительные задачи: например, терминировать SSL (принимать HTTPS), кешировать некоторые ответы, сжимать их, добавлять заголовки безопасности. В контексте проекта основная функция – раздать трафик: отделить `/api/auth` на AuthService, `/api/lostfound` на LostFoundService. Без Nginx потребовалось бы экспонировать каждый сервис на отдельном порту и клиенту знать все эти адреса; с Nginx система выглядит для клиента как единое целое.

10. **Что такое параметр `root_path` в FastAPI и почему он понадобился в настройке?**
    Ответ: `root_path` сообщает приложению FastAPI, под каким префиксом пути оно "смонтировано" за прокси. Например, если сервис фактически доступен внешне по адресу `http://example.com/api/lostfound/...` (т.е. префикс `/api/lostfound`), то внутри FastAPI нужно указать `root_path="/api/lostfound"`. Тогда автоматически документация Swagger и ссылки будут корректно включать этот префикс. В нашем проекте, где Nginx проксирует с добавлением `/api/auth` и `/api/lostfound`, без установки root_path у сервисов были бы проблемы: Swagger-документация, например, могла бы указывать пути без `/api/lostfound`, и через шлюз они не работали бы. Поэтому мы настроили FastAPI приложений с соответствующим root_path, чтобы они правильно формировали URL, учитывая, что находятся за прокси. Также Uvicorn при запуске передаётся этот же параметр. В итоге, Nginx + FastAPI работают согласованно: Nginx лишь переадресует пути, а FastAPI знает свой префикс и обслуживает запросы нормально.

11. **Как Docker Compose помогает в запуске и работе микросервисов?**
    Ответ: Docker Compose позволяет описать все микросервисы (контейнеры) в одном конфигурационном файле и запустить их одной командой. В Compose-файле задаются образы/настройки для каждого сервиса и общие сети. Он автоматически создаёт сеть, где все контейнеры видят друг друга по именам сервисов – благодаря этому, например, Nginx может обращаться к `auth-service` по имени, без ручной настройки IP. Compose упрощает управление: можно поднять или остановить всю группу сервисов, или масштабировать какой-то сервис (запустить N копий) для тестов. По сути, Compose обеспечивает локальную оркестрацию: разработчику не нужно вручную запускать каждый сервис и настраивать соединения – всё описано декларативно. Это ускоряет развертывание окружения разработки/тестирования и гарантирует, что сервисы запущены с правильными параметрами (переменные окружения, порты, тома и т.д.). In summary, Docker Compose сводит многокомпонентную систему в единое целое, которым легко управлять.