**2. Подготовка среды разработки**

**Инструменты и окружение**

* **Mac как рабочая станция:**
  + Убедитесь, что macOS обновлён до последней версии.
  + Установите [Homebrew](https://brew.sh/) (если еще не установлен) для удобного управления пакетами.
* **Java и IDE:**
  + Установите JDK 11 или выше (например, через Homebrew: brew install openjdk@11).
  + Выберите IDE для разработки (IntelliJ IDEA является отличным выбором для Spring Boot).
* **Docker:**
  + Установите Docker Desktop для macOS, чтобы легко создавать и управлять контейнерами.
  + С помощью Docker создайте базовые контейнеры для:
    - **Keycloak:** для управления пользователями и аутентификацией.
    - **RabbitMQ:** для асинхронной коммуникации между микросервисами.
    - **Redis:** для кэширования и сессионного хранения.
    - **Базы данных:** например, PostgreSQL или MySQL для хранения данных приложения.
* **Система контроля версий:**
  + Настройте Git и создайте репозиторий (например, на GitHub или GitLab).
  + Подумайте о настройке CI/CD, чтобы автоматизировать сборку и тестирование проекта.

**Библиотеки и зависимости**

* **Spring Boot и связанные модули:**
  + Подготовьте проект на Spring Boot с необходимыми зависимостями:
    - Spring Web (для создания REST API)
    - Spring Security (для защиты API, интеграция с Keycloak)
    - Spring Data JPA (или другой подходящий модуль для работы с базой данных)
    - Другие модули по мере необходимости (например, Spring Cloud для микросервисной архитектуры)
* **Конфигурация подключения:**
  + Настройте application.properties или application.yml для подключения к базе данных, а также для настройки Keycloak и других интеграций.
  + Создайте профили для разработки и продакшена, чтобы облегчить тестирование и развертывание.

**3. Реализация бэкенд-сервисов**

**Архитектура и структура проекта**

* **Модульная организация кода:**
  + Рассмотрите возможность создания отдельных Spring Boot проектов или модулей для каждого микросервиса (например, сервис пользователей, сервис заказов, сервис уведомлений).
  + Определите четкие API-контракты между сервисами.

**Сервис аутентификации и авторизации**

* **Интеграция с Keycloak:**
  + Настройте Keycloak как внешний провайдер аутентификации.
  + Используйте адаптер Keycloak для Spring Security, чтобы защитить свои REST API через JWT и OAuth2.
  + Создайте базовые роли (например, ROLE\_USER, ROLE\_CONFEITER и т.д.) и настройте маппинг в Keycloak.

**Сервис пользователей**

* **Функционал регистрации и профиля:**
  + Реализуйте API для регистрации пользователей, как кондитеров, так и заказчиков.
  + Продумайте валидацию входных данных и обработку ошибок.
  + Создайте сущность пользователя с необходимыми полями (имя, email, тип пользователя, портфолио для кондитеров и т.д.).

**Сервис заказов**

* **Основные функции:**
  + Разработайте REST API для создания, редактирования и отслеживания заказов.
  + Определите логику распределения заказов – например, как заказы будут назначаться ближайшим кондитерам (на базе геолокации).
  + Реализуйте механизм обновления статусов заказов (создан, в процессе, выполнен, отменён).

**Сервис уведомлений и асинхронной обработки**

* **Интеграция с RabbitMQ:**
  + Настройте RabbitMQ для обмена сообщениями между микросервисами.
  + Реализуйте отправку уведомлений (например, об изменении статуса заказа) через очереди RabbitMQ.
  + Создайте слушателей (listeners) для обработки сообщений и обновления статуса в базе данных.

**Кэширование и оптимизация**

* **Использование Redis:**
  + Интегрируйте Redis для кэширования часто используемых данных (например, сессий пользователей или результатов запросов).
  + Настройте время жизни кэша и механизмы обновления данных для повышения производительности.

**Дополнительные рекомендации**

* **Тестирование:**
  + С самого начала пишите юнит-тесты для ключевых компонентов.
  + Организуйте интеграционные тесты для проверки взаимодействия между микросервисами.
* **Документация API:**
  + Используйте Swagger/OpenAPI для документирования ваших REST API.
  + Это поможет вам и команде быстрее понять и протестировать функциональность.
* **Контейнеризация:**
  + Создайте Dockerfile для каждого микросервиса.
  + Настройте Docker Compose для удобного запуска всех необходимых контейнеров локально.

**1. User (Пользователь)**

Поскольку аутентификацию и базовые данные вы, вероятно, будете хранить в Keycloak, в вашей БД могут храниться лишь дополнительные сведения, относящиеся к профилю и специфике работы с заказами.

**Поля:**

* **id** (UUID или Long) – уникальный идентификатор пользователя.
* **email** – контактный адрес (если не берётся из Keycloak, либо для локальных расширенных данных).
* **fullName** – полное имя пользователя.
* **role** – тип пользователя (например, CUSTOMER для заказчика и CONFECTIONER для кондитера).  
  *Замечание:* Возможно, роль определится и через Keycloak, а здесь — дополнительные атрибуты.
* **phone** – контактный номер.
* **location** – данные для геолокации (можно хранить как объект с latitude и longitude или строковое представление адреса).
* **rating** – средняя оценка (для кондитеров) или отзыв.
* **portfolioUrl** – URL на портфолио (актуально для кондитеров).
* **storeName** – если пользователь представляет магазин, можно указать название магазина.
* **createdAt/updatedAt** – метки времени создания/обновления профиля.

**2. Order (Заказ)**

Заказ может быть как готовым товаром из каталога магазина, так и индивидуальным заказом, где клиент описывает пожелания. В комбинированном подходе важно учитывать, что после создания заказа он сначала проходит через этап push-уведомления, а затем (при отсутствии отклика) переходит в пул активных заказов для получения заявок.

**Поля:**

* **id** (UUID или Long) – уникальный идентификатор заказа.
* **orderType** – тип заказа, например, STORE\_PRODUCT или CUSTOM\_ORDER.
* **description** – подробное описание заказа (требования, пожелания).
* **price** – ориентировочная или фиксированная цена (если применимо).
* **status** – статус заказа. Возможные значения:
  + CREATED – заказ создан, но push-уведомление ещё не отправлено или в процессе.
  + PUSH\_PENDING – заказ в режиме push-уведомления, ожидается отклик кондитеров.
  + ACTIVE – заказ доступен для выбора из пула (pull-метод).
  + ASSIGNED – заказ принят конкретным кондитером.
  + IN\_PROGRESS – заказ в процессе выполнения.
  + COMPLETED – заказ завершён.
  + CANCELED – заказ отменён.
* **customer** – ссылка на пользователя-заказчика (Many-to-One к User).
* **assignedConfectioner** – ссылка на выбранного/принявшего заказ кондитера (Many-to-One к User; может быть null, если заказ ещё не назначен).
* **createdAt** – время создания заказа.
* **updatedAt** – время последнего обновления.
* **location** – геоданные заказа (адрес или координаты, если требуется для поиска ближайших кондитеров).

*Опционально:*

* **pushNotificationDeadline** – временная метка, до которой заказ находится в режиме push, после чего автоматически переходит в пул активных заказов.

**3. OrderOffer (Заявка на выполнение заказа)**

Эта сущность нужна для реализации pull-метода, когда кондитеры сами выбирают заказы и отправляют свои предложения. Клиент сможет просмотреть список заявок и выбрать наиболее подходящую.

**Поля:**

* **id** (UUID или Long) – уникальный идентификатор заявки.
* **order** – ссылка на заказ (Many-to-One к Order).
* **confectioner** – ссылка на пользователя-кондитера (Many-to-One к User).
* **proposedPrice** – предлагаемая цена за выполнение заказа.
* **message** – сообщение от кондитера с пояснениями или комментариями.
* **offerStatus** – статус заявки, например:
  + PENDING – заявка подана и ожидает рассмотрения.
  + ACCEPTED – клиент принял заявку.
  + REJECTED – заявка отклонена.
* **createdAt** – время подачи заявки.

**4. Product (Товар магазина)**

Для режима покупки готовых изделий из магазина можно выделить отдельную сущность товаров.

**Поля:**

* **id** (UUID или Long) – уникальный идентификатор товара.
* **name** – название изделия.
* **description** – описание товара.
* **price** – цена.
* **imageUrl** – ссылка на изображение товара.
* **store** – ссылка на магазин или кондитера, который предлагает товар (Many-to-One к User или отдельной сущности Store, если планируется её выделение).
* **createdAt/updatedAt** – метки времени создания и обновления.

**5. (Опционально) Store (Магазин)**

Если хотите явно разделять физические лица и магазины, можно создать отдельную сущность для магазина. Альтернативно, информацию о магазине можно хранить как часть профиля пользователя с ролью кондитера.

**Поля:**

* **id** – уникальный идентификатор.
* **name** – название магазина.
* **description** – описание.
* **address** – физический адрес.
* **rating** – рейтинг магазина.
* **owner** – ссылка на пользователя-владельца.
* **createdAt/updatedAt**.

**6. Возможные связи и бизнес-логика**

* **Пользователь (заказчик)** создаёт заказ (Order).
* При создании индивидуального заказа:
  + Система инициирует push-уведомление кондитерам, которые находятся в статусе «доступен».
  + Если кондитер принимает заказ через push, поле assignedConfectioner заполняется, а статус заказа обновляется до ASSIGNED.
  + Если push-уведомление не приводит к быстрому принятию, заказ автоматически переходит в статус ACTIVE, и кондитеры могут подавать заявки (создавая записи в OrderOffer).
  + Клиент видит пул заявок (OrderOffer) и выбирает наиболее подходящую, после чего заказ получает статус ASSIGNED, и выбранный кондитер становится исполнителем.
* **Пользователь (кондитер)** может быть как физическим лицом, так и представлять магазин (зависит от поля role или отдельной связи со Store).

**Заключение**

Эта модель позволит:

* Обрабатывать заказы как из каталога готовых изделий, так и индивидуальные заказы.
* Использовать комбинированный подход с push-уведомлением и пулом заявок (pull-метод) для гибкого назначения заказов.
* Расширять функционал (например, добавлять отзывы, рейтинги, историю заказов) без значительного пересмотра базовой схемы.

Если нужно углубиться в детали, например, про отношения между сущностями или особенности реализации бизнес-логики в сервисном слое, можем разобрать конкретные моменты подробнее.

**Архитектура и описание компонентов**

1. **API Gateway**  
   Единая точка входа, которая маршрутизует запросы к соответствующим микросервисам. Это может быть, например, Spring Cloud Gateway или другой подобный инструмент.
2. **Authentication Service (Keycloak)**  
   Сервис аутентификации и авторизации. Keycloak управляет пользователями, ролями и токенами, а ваше приложение использует его для защиты REST API.
3. **User Service**  
   Отвечает за управление дополнительными данными пользователей (например, профили, информация о кондитерах, рейтинги, портфолио).
4. **Order Service**  
   Управляет созданием и обновлением заказов. Здесь реализуется логика перехода заказа между статусами: от создания, через push-уведомления, до активного состояния и назначения кондитеру.
5. **Order Offer Service**  
   Если заказ переходит в пул активных заказов, здесь кондитеры могут подавать заявки (оферы) на выполнение заказа. Клиент затем выбирает подходящую заявку.
6. **Product Service**  
   Обрабатывает товары из каталога магазинов (готовые изделия). Здесь хранится информация о продуктах, их описание, цена и изображение.
7. **Notification Service**  
   Отвечает за уведомления в режиме реального времени (через WebSocket, RabbitMQ или GraphQL Subscriptions) – информирует кондитеров о новом заказе или изменении статуса.
8. **Инфраструктурные компоненты:**
   * **Database:** Основная база данных (например, PostgreSQL) для хранения всех сущностей.
   * **Redis:** Используется для кэширования данных и хранения сессий, что ускоряет отклики сервисов.
   * **RabbitMQ:** Организация асинхронного обмена сообщениями между сервисами, например, для уведомлений или обработки заказов.

**Как работает система**

* **Клиентская сторона:**  
  Мобильное и веб-приложение обращаются к API Gateway для доступа ко всем функциональностям.
* **Аутентификация:**  
  API Gateway перенаправляет запросы по авторизации к Keycloak. После успешного логина клиент получает JWT-токен для дальнейших запросов.
* **Заказы и уведомления:**
  + Клиент создаёт заказ через Order Service.
  + Order Service устанавливает статус (например, CREATED и PUSH\_PENDING) и инициирует уведомление через Notification Service.
  + Notification Service (с помощью RabbitMQ и/или WebSocket) отправляет уведомления доступным кондитерам.
  + Если кондитер принимает заказ, Order Service обновляет заказ и назначает исполнителя.
  + Если push-уведомление не приводит к быстрому отклику, заказ переходит в статус ACTIVE, и кондитеры подают заявки через Order Offer Service.
* **Товары:**  
  Готовые изделия обрабатываются через Product Service, который предоставляет каталог для просмотра и покупки.
* **Кэширование и асинхронное взаимодействие:**  
  Redis ускоряет доступ к часто запрашиваемым данным, а RabbitMQ обеспечивает надежную асинхронную коммуникацию между сервисами (например, для обработки уведомлений и заказов).