Лабораторная работа №2

Проектирование рекомендательной ленты для социальной сети

Постановка задачи

К вам обратился Product Owner популярной социальной сети, с целью обеспечить доступ пользователя к контенту даже на парковке. Ваша задача – спроектировать архитектуру сервиса ленты постов для социальной сети. Пользователь подписывается на авторов, видит их новые посты в своей персональной ленте, может открывать комментарии и получать уведомления о новых событиях почти в реальном времени.

Функциональные требования:

1. Регистрация/логин, управление профилем.
2. Подписка/отписка на авторов.
3. Публикация постов (текст + опциональные медиа-ссылки).
4. Просмотр персональной ленты с **бесконечной прокруткой** (пагинация по курсору, а не по offset).
5. Добавление и чтение комментариев к постам.
6. Уведомления в реальном времени о новых постах подписок и новых комментариях к моим постам.

Нефункциональные требования:

1. Низкая задержка чтения ленты: p95 < 150 мс для партии из 20 элементов.
2. Высокая пропускная способность записи комментариев (бурсты до 10k RPS без 5xx).
3. Горизонтальное масштабирование ключевых компонентов.
4. Устойчивость к сбоям брокера/воркеров: события не теряются
5. Идемпотентность обработчиков событий и безопасная повторная доставка.

Задания:

1. Оценочные расчёты

- Оцените суточные и пиковые RPS:
  - публикация постов;
  - доставки постов в ленты;
  - чтение ленты (партии по 20);
  - запись/чтение комментариев.
- Оцените объём хранения на 1 и 3 года: посты (метаданные), индексы ленты, комментарии, кэш.
- Заложите бюджет на рост в 3 раза без архитектурных изменений.

2. Модель данных (логическая)

- Реляционный контур (users, posts, subscriptions, outbox_events): ключи, уникальности, транзакционные границы.
- Хранилище ленты
- Комментарии: схема под запрос «N последних по посту, сортировка по времени».
- Опишите, что и где кэшируется для гидратации постов по ID.

3. Технические решения (сравнение и выбор)

- Стратегия формирования ленты
- Пагинация для infinite scroll
- Outbox/Transactional messaging
- Идемпотентность и повторная доставка
- Гидратация: порядок источников (кэш → сервис постов), размеры батчей, таймауты и деградация.
- Бэкап/восстановление и репликация для разных хранилищ.

4. Нарисуйте UML-диаграммы

- **Component / Container** (уровень микросервисов, шина событий, хранилища, кэш).
- **Sequence** (три сценария):
    1. публикация поста с Outbox и доставкой события;
    2. чтение ленты с курсором и гидратацией;
    3. комментарий → событие → WebSocket-уведомление.
- **Deployment** (узлы/контейнеры, масштабирование, внешние зависимости).
- **Class/Data** (логическая модель для реляционной части + стереотипы/примечания для нереляционной).

Решение

Для того, чтобы выполнить оценочные расчеты, необходимо понимать какое у социальной сети количество активных пользователей каждый день. Предположим, что таких пользователей 3 000 000. Также нужно знать среднее значение каждой активности пользователя в день. Поскольку по требованиям известно только, что пользователь за один запрос на ленту получает 20 постов, то эти данные мы учтем, а по поводу остальных – предположим, что мы получили эти данные от отдела аналитики:

Событие	**Среднее значение в день на одного активного пользователя**
Публикация поста	0.4 поста
Открытие ленты	7 раз
Пролистывание ленты	5 страниц по 20 элементов

- Будем считать, что в среднем 0.6 комментария на пост
- Будем считать, что в среднем 30 подписок на пользователя
- Будем считать, что 1 000 000 пользователей активно пользуются лентой в день

- Будем считать, что каждый активный пользователь 4 раза в день открывает комментарии под постами

Также в требованиях не было предоставлено данных по поводу пиковой нагрузки для каждой активности. Поэтому там также будем производить расчеты исходя из собственных предположений.

1. Оценочные расчеты

1.1. Суточные и пиковые RPS

1.1.1. Публикации постов

– Суточный RPS:

$0.4 * 3\,000\,000 = 1\,200\,000$ - столько постов всего создают в сутки в нашей социальной сети

В сутках 86 400 секунд

$\rightarrow \frac{1\,200\,000}{86\,400} \approx$ **14 RPS** - запросов в секунду на создание поста

– Пиковый RPS:

Предположим, что мы знаем коэффициент пиковости. Пусть, к примеру, он будет равен 10

$\rightarrow 14 * 10 =$ **140 RPS** – пиковый показатель

1.1.2. Доставки постов в ленты

Мы должны учесть, что каждый пост должен быть доставлен в ленты всех подписчиков автора.

– Суточный RPS:

В сутки, создается 1 200 000 постов

→ $1\,200\,000 * 30 = 36\,000\,000$ – доставок постов в ленту в сутки

→ $\frac{36\,000\,000}{86\,400} \approx \mathbf{417\ RPS}$ – доставок постов в ленту в секунду

– Пиковый RPS:

$417 * 10 = \mathbf{4170\ RPS}$

1.1.3. Чтение ленты (партии по 20)

Мы имеем 1 000 000 пользователей в день, которые активно пользуются лентой, 7 открытый ленты в день + 5 страниц по 20 постов.

– Суточный RPS:

→ $1\,000\,000 * 7 * 5 = 35\,000\,000$ - запросов на чтение ленты в сутки

→ $\frac{35\,000\,000}{86\,400} \approx \mathbf{406\ RPS}$ - запросов на чтение ленты в секунду

– Пиковый RPS:

$406 * 10 = \mathbf{4060\ RPS}$

1.1.4. Запись комментариев

1 200 000 - столько постов всего создают в сутки в нашей социальной сети. В среднем 0.6 комментария на пост.

– Суточный RPS записи комментариев:

→ $1\,200\,000 * 0.6 = 720\,000$ - всего комментариев в сутки

→ $\frac{720\,000}{86\,400} \approx \mathbf{9\ RPS}$

– Пиковый RPS записи комментариев:

$9 * 10 = \mathbf{90\ RPS}$

1.1.5. Чтения комментариев

– Суточный RPS чтения комментариев:

1 000 000 * 4 = 4 000 000 (каждый активный пользователь 2 раза в день открывает комментарии под постами) - запросов на чтение комментариев в сутки.

$$\rightarrow \frac{4\ 000\ 000}{86\ 400} \approx \mathbf{47\ RPS}$$

– Пиковый RPS чтения комментариев:

$$47 * 10 = \mathbf{470\ RPS}$$

1.2. Объем хранения: посты, лента, комментарии, кэш

1.2.1. Посты

Пусть одна запись поста с индексами занимает где-то 800 байт (с учетом capacity, а также индексов, служебных структур и т.д.). Мы имеем:

- 1 200 000 – столько постов всего создают в сутки в нашей социальной сети
- 365 дней – в году

$\rightarrow 1\ 200\ 000 * 365 = 438\ 000\ 000$ – столько постов будет создано за год

$\rightarrow 438\ 000\ 000 * 800$ байт $\approx \mathbf{327}$ Гб – потребуется для хранения всех постов за год

$\rightarrow 327 * 3$ года $= \mathbf{981}$ Гб – потребуется для хранения всех постов за 3 года

1.2.2. Индексы ленты

Мы имеем:

- 36 000 000 - доставок в ленту в сутки

→ $36\,000\,000 * 365 = 13\,140\,000\,000$ – записей в хранилище ленты за год

→ При оценке 70 байт на запись, получаем

$13\,140\,000\,000 * 70 = \mathbf{857}$ Гб - потребуется для хранения индексов ленты за год

→ $857 * 3 = \mathbf{2571}$ Гб – потребуется для хранения индексов ленты за 3 года

1.2.3. Комментарии

Пусть одна запись поста с индексами занимает где-то 700 байт (с учетом capacity, а также индексов, служебных структур и т.д.). Мы имеем:

- 720 000 – всего создается комментариев в сутки

→ $720\,000 * 365 = 262\,800\,000$ – создается комментариев за год

→ $262\,800\,000 * 700$ байт $= \mathbf{172}$ Гб – потребуется для комментариев за год

→ $172 * 3 = \mathbf{516}$ Гб – потребуется для комментариев за 3 года

1.2.4. Кэш

Стоит отметить, что данные в кэше регулярно удаляются. Объем кэша нужно оценивать исходя из того, сколько актуальных, часто запрашиваемых данных мы хотим в нем хранить.

Мы будем кэшировать три типа объектов для гидрации ленты:

- Посты – будем кэшировать посты, например, за последние 7 дней.

- Данные о пользователях – кэшируем данные 3 млн активных пользователей.
- Статистика постов (лайки, комментарии) – будет кэшировать статистику для постов за последние, например, 7 дней.

Не забываем, что в сутки у нас 1 200 000 постов.
При оценке:

- Посты: 800 байт одна запись
- Пользователи: 800 байт одна запись
- Статистика: 500 байт одна запись

$\rightarrow 1\,200\,000 * 7 * 800 \approx \mathbf{6}$ Гб – необходимо для постов
$\rightarrow 3\,000\,000 * 800 \approx \mathbf{3}$ Гб
$\rightarrow 1\,200\,000 * 7 * 500 \approx \mathbf{4}$ Гб – необходимо для постов

Итого: $6 + 3 + 4 = \mathbf{13}$ Гб
Берем с capacity: $\mathbf{80}$ Гб – потребуется всего

1.3. Бюджет на рост в 3 раза без архитектурных изменений

При росте нагрузки в 3 раза мы получим:

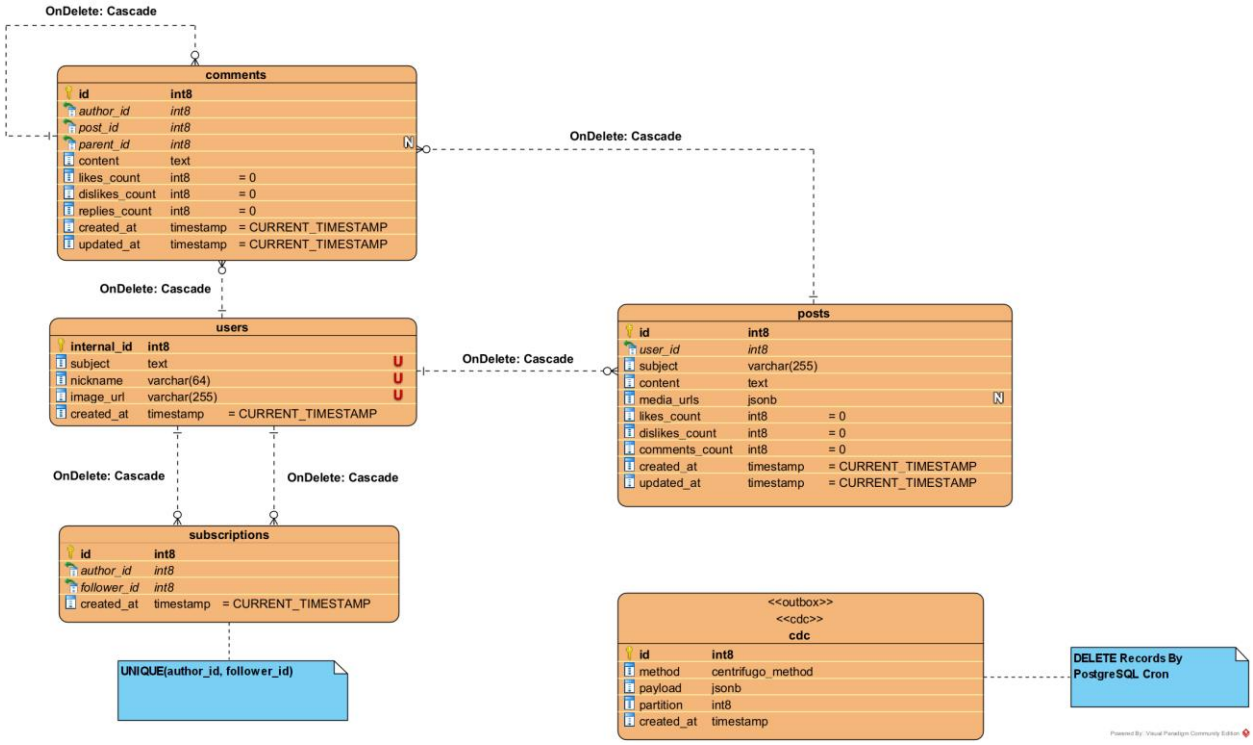
- 1 200 000 * 3 = 3 600 000 – постов будут создаваться в сутки
- 36 000 000 * 3 = 108 000 000 – будет доставок ленты в сутки
- 35 000 000 * 3 = 105 000 000 – будет запросов на чтение ленты в сутки
- И т.д.

Объёмы хранилищ увеличиваются до **8 Тб** для ленты и примерно до **2.87 Тб** для постов за 3 года. Наша архитектура (горизонтальное масштабирование сервисов, кластерное хранилище ленты,

партиционированная PostgreSQL с репликами, Redis Cluster) позволяет выдержать такой рост без принципиальных архитектурных изменений.

2. Модель данных (логическая)

2.1. Реляционный контур



Важно отметить, что тип centrifugo_method в таблице cdc – это enum, содержащий методы API Centrifugo.

Также важно отметить, что хоть мы и храним в PostgreSQL для постов и комментариев статистику: количество лайков, дизлайков, комментариев для постов, ответов для комментариев, но эти данные будут обновляться в PostgreSQL редко. Мы также эту статистику будем хранить в кэше – Redis. И именно в Redis мы будем регулярно обновлять эти значения и брать их оттуда. PostgreSQL будет иногда выполнять синхронизацию с Redis для такой статистики.

2.2. Транзакционные границы

Каждое событие (их всего два – создание поста и создание комментариев), требующее realtime-доставки, фиксируется в одной транзакции вместе с изменением бизнес-данных. То есть транзакционные границы проходят по созданию постов и комментариев. Операции CREATE POST и CREATE COMMENT выполняются атомарно вместе с записью события в таблицу cdc. Сначала сервис открывает транзакцию, затем:

- вставляет запись в posts или comments
- формирует событие (POST_CREATED или COMMENT_CREATED)
- сохраняет событие в таблицу cdc в той же транзакции

Для доставки realtime-уведомлений используется Centrifugo, работающий по PUB/SUB-модели. События, которые должны быть доставлены пользователям (создание постов и комментариев), фиксируются в таблице cdc в рамках одной транзакции вместе с бизнес-операцией, как было описано ранее.

После COMMIT запись в posts/comments и запись в cdc гарантированно попадают в PostgreSQL WAL (журнал транзакций). После этого Debezium сразу же считывает изменения таблицы cdc из WAL и публикует их в Kafka как CDC-сообщения (Debezium это делает практически мгновенно).

Сразу после этого (практически мгновенно) Centrifugo извлекает CDC-сообщения из Kafka (Centrifugo подключился до этого к Kafka как асинхронный consumer), интерпретирует поля method и payload как команды серверного API Centrifugo (broadcast / publish) и доставляет их в соответствующие WebSocket-каналы. А клиенты, которые были подписаны ранее на эти каналы, получат эти события в реальном времени.

Так будет обеспечиваться идемпотентность, гарантия доставки и отсутствие потерь при сбоях брокера.

2.3. Хранилище ленты

В качестве хранилища ленты будем использовать нереляционное СУБД. Лучше всего под заданные требования подходит Casandra/ScyllaDB:

```
CREATE TABLE feed (
    user_id  BIGINT,
    score    TIMESTAMP,
    post_id  BIGINT,
    PRIMARY KEY (user_id, score, post_id)
) WITH CLUSTERING ORDER BY (score DESC, post_id DESC);
```

Здесь:

- user_id – ключ партиции (partition key). Все строки ленты конкретного пользователя хранятся в одной партиции, что обеспечивает быстрое последовательное чтение без сканирования других пользователей. То есть это конкретный владелец ленты

- post_id – id поста, который показывается в ленте пользователю. Также это дополнительный кластеризующий ключ, гарантирующий уникальность строк, даже если два поста имеют одинаковое время публикации

- score – время публикации поста (но можно использовать рейтинг вместо времени). То есть кластеризующий ключ, определяющий порядок элементов внутри партиции.

Также:

- partition = user_id. То есть user_id — определяет владельца ленты. Все его посты хранятся в одной партиции на одной ноде.

- сортировка по score DESC, post_id DESC. То есть это сортирует строки внутри партиции, т.е. внутри ленты, а также дает возможность пагинации по курсору. Новые посты всегда находятся в начале партиции, а чтение по LIMIT получает самые свежие записи без сортировки и фильтрации

- чтение N элементов = один диапазон по кластеризованному ключу. То есть когда приходит запрос:

```
9   SELECT post_id FROM feed
10  WHERE user_id = 123
11  LIMIT 20;
12
```

Cassandra идёт в партицию пользователя 123. Берёт первые 20 элементов (они уже отсортированы DESC по времени и post_id). Возвращает результат. Это одна операция чтения в пределах одной партиции, без сканирования, сортировки или offset, то есть минимальная задержка и p95 < 150 мс легко достижимы.

То есть когда автор создает пост, в основной БД PostgreSQL создается запись, а потом создается записи в Cassandra в таблицу feed для каждого из подписчиков этого автора:

- user_id - id подписчика автора
- post_id - id поста автора, который выложил пост

И лента будет читаться мгновенно, потому что данные рассортированы заранее.

2.4. Комментарии - схема под запрос «N последних по посту, сортировка по времени»

У нас есть таблица comments. Запрос, который нужен: «Получить последние N комментариев к посту, сортировка по времени». За счет таблицы comments, мы можем просто выполнить запрос:

```
SELECT *
FROM comments
WHERE post_id = :postIdValue
ORDER BY created_at DESC
LIMIT N;
```

Данный запрос полностью покрывает данное требование. Также мы создадим следующий индекс: INDEX (post_id, created_at DESC) для оптимизации поиска под такой запрос.

2.5. Кэширование для гидратации постов по ID

Когда пользователь читает ленту, специальный сервис (например, можно назвать его FeedService) будет получать из таблицы feed в Cassandra/ScyllaDB - список post_id. Однако это просто список post_id, где самих постов нет. То есть остальные данные живут не в Cassandra/ScyllaDB, а в другой БД. Поэтому сервис, когда получит данные из Cassandra/ScyllaDB должен гидрировать их в объекты:

- Сам пост (его текст, заголовок и т.д.)
- Данные автора этого поста
- Статистику (лайки и комментарии)

Для этого будем использовать Redis в качестве кэша:

- Кэш постов: ключ post_id, а значение - PostDTO (текст, медиа, метаданные)

- Кэш пользователей: ключ user_id, значение - UserDTO (nickname, image_url и т.д.).

- Кэш статистики: ключ post_id, значение — счётчики лайков и комментариев

То есть работать это будет так:

- Пользователь читает ленту

- FeedService читает N элементов ленты, обращая к Cassandra/ScyllaDB. Получает оттуда список post_id длиной N

- Далее FeedService обращается в Redis. То есть делаем MGET по ключам постов. Если посты есть в кэше – берем их оттуда. Если постов нет, то обращаемся к основной БД PostgreSQL и берем данные оттуда. А после кладем эти данные в Redis

- Параллельно запрашиваем всех авторов этих постов. Тут также – обращаемся в Redis по MGET. Если есть данные – берем их из Redis. Если данных нет, то обращаемся к основной БД PostgreSQL и берем данные оттуда. А после кладем эти данные в Redis

- Параллельно запрашиваем статистику постов через MGET по такой же схеме

- Собираем PostDTO, UserDTO и PostStatsDTO в итоговый объект на основе полученных данных и отправляем на Frontend

При обновлениях постов, профилей пользователей и статистики применяется подход write-through: данные сначала записываются в PostgreSQL, затем соответствующие ключи в Redis перезаписываются или

инвалидируются (данные ключи удаляются, а новые будут добавлены туда со временем – когда сервис поймет, что в кэше нет этих данных и нужно заново туда их записать через основную БД), обеспечивая согласованность кэша и основного хранилища.

3. Технические решения

3.1. Стратегия формирования ленты

Было решено использовать стратегию fan-out on write. Fan-out on Write (рассылка при записи) - это стратегия, при которой при создании нового поста (записи) он сразу же асинхронно доставляется в персональное хранилище ленты (партицию) каждого подписчика, что обеспечивает мгновенное чтение ленты, поскольку все данные уже отсортированы и готовы к выдаче.

Таким образом:

- чтение ленты становится максимально быстрым — мы просто читаем готовый список post_id
- легко реализуется пагинация по курсору на уровне хранилища ленты
- нагрузка переносится на момент публикации, что проще контролировать через асинхронных воркеров

Для авторов с очень большим количеством подписчиков в будущем можно использовать гибридный подход (комбинация fan-out и fan-in), но базовый дизайн ориентирован на fan-out on write.

Fan-in (Fan-out on Read) - это стратегия формирования ленты, при которой, в отличие от Fan-out on Write, ничего не происходит при публикации поста. Лента формируется только в момент, когда пользователь запрашивает ее чтение.

3.2. Пагинация для infinite_scroll

Пагинация для ленты будет реализована по курсору (Cursor-based), а не по смещению (Offset). Такой подход устойчив к проблемам производительности и точности данных, возникающим при прокрутке глубоко в ленте (например, при запросе OFFSET 100000 LIMIT 20).

Курсор — это уникальная точка, которая однозначно указывает, где закончена предыдущая страница. В нашем случае курсор будет содержать пару значений (score, post_id) последнего поста в предыдущей партии, что соответствует кластеризованному ключу в таблице ленты.

Следующие страницы ленты запрашиваются вида GET /feed?cursor=<cursor>&limit=20 и считываются из Cassandra/ScyllaDB как диапазон по кластеризованному ключу (score, post_id). Это гарантирует низкую задержку чтения и исключает пропуск или дублирование постов при одновременной публикации новых.

3.3. Outbox/Transactional messaging

Для доставки в реальном времени клиенту событий создания новых постов подписок и создания нового комментария к собственным постам – используется Centrifugo – сервер обмена сообщениями в режиме реального времени. Centrifugo работает по модели PUB / SUB, используя механизм каналов по WebSocket для доставки событий в реальном времени. Также мы используем встроенный асинхронный потребитель Centrifugo из тем Kafka и Debezium для доставки событий в Kafka из основной базы данных PostgreSQL. По сути, мы используем Outbox-паттерн и CDC-паттерн.

Общий алгоритм работы следующий:

1. Клиент подписывается на канал Centrifugo

2. Когда происходит новое событие, например создание нового комментария к собственному посту, наш сервис в одной транзакции для основной базы данных PostgreSQL создаст новую запись в

таблице comments (или posts если новое событие – это создание нового поста), а также на основе этой новой созданной записи – создаст новую запись в таблице cdc, которая будет содержать следующие данные:

- id – идентификатор этой новой записи в cdc
- method – здесь будет указан один из API методов Centrifugo (publish или broadcast), который предназначен для публикации событий на каналы Centrifugo
- payload – здесь будет указан JSON, который будет являться телом запроса для выбранного API метода Centrifugo (publish или broadcast). Тут будут указаны каналы Centrifugo, на которые нужно опубликовать это новое событие, ключ идемпотентности данного события, данные нового события (текст комментария, его автор и т.д.) и т.д.
- partition – здесь будет указан номер потока Centrifugo, который должен будет получить данное событие и опубликовать его на канал (можно вычислять хэш по id нового события)
- created_at – дата создания новой записи в cdc

3. Если транзакция из предыдущего пункта успешно выполнит коммит, то она будет записана в журнал транзакций PostgreSQL WAL

4. Далее фоновый процесс Debezium мгновенно считает из этого журнала данные и отправит их в соответствующую тему Kafka

5. Далее фоновый процесс Centrifugo считает эти данные из темы Kafka и опубликует это новое событие на соответствующий канал, выполнив метод API, указанный в полученных данных

6. Все клиенты, которые подписаны на данный канал из пункта 5 – получат опубликованные данные в реальном времени по WebSocket

Таким образом, события будут публиковаться в реальном времени, обеспечивая гарантированную доставку всех событий, даже в случае какого-то сбоя (за счет использования встроенного асинхронного потребителя Centrifugo и сохранения событий в Kafka и PostgreSQL).

3.4. Идемпотентность и повторная доставка

Чтобы опубликовать в реальном времени событие на канал Centrifugo, нужно вызвать один из POST-методов API Centrifugo – publish или broadcast. Для обоих этих методов в теле запроса можно передать ключ идемпотентности. В качестве значения ключа идемпотентности – мы можем передавать id события, которое является уникальным, то есть использовать первичный ключ из базы данных. Такие ключи идемпотентности позволяют предотвратить повторную доставку на уровне системы. То есть Centrifugo будет брать событие из тем Kafka, затем Centrifugo увидит по полученным данным, что это API-запрос publish/broadcast на определенный канал, увидит, что в теле запроса для данного API есть ключ идемпотентности, а затем Centrifugo использует этот ключ, чтобы определить, публиковалось ли обрабатываемое событие ранее на указанный канал. Если событие с таким ключом уже публиковалось, оно не будет повторно отправлено. Информация о ключах хранится в кэше Centrifugo. Также не стоит забывать, что данные из Kafka будут автоматически очищаться по истечению определенного времени, чтобы Centrifugo каждый раз не читал уже обработанные события.

Таким образом, мы значительно снизим вероятность повторной доставки событий. То есть публикации события, которое ранее уже было опубликовано.

Стоит отметить, что хоть мы и используем ключи идемпотентности, а также своевременную очистку данных из Kafka, все-равно существует низкая вероятность повторной публикации событий, так как Centrifugo запоминает ключи идемпотентности событий только на определенное время. Хоть вероятность и низкая, но она существует.

Если такая низкая вероятность повторной доставки событий не приемлема, то можно использовать версионирование для постов. Каждый раз, когда пост будет создаваться, поле version в базе данных будет равно 0. Когда будет создаваться новый комментарий к данному посту – его version будет транзакционно увеличиваться. Также можно увеличивать version, если автор изменяет созданный пост (если важно в реальном времени показывать подписчикам изменение поста). Это поле version будет доставляться с остальными данными события в Centrifugo. Оттуда Centrifugo опубликует на канал данное событие. Затем клиент, подписанный на этот канал – получит данные этого события. Там также будет это поле version. И можно будет на стороне клиента – запоминать текущую версию поста и сравнивать с новой полученной. Если мы получили версию, которая меньше или равна текущей версии, то значит мы получили событие, которое уже опубликовали. Значит его можно не отображать на клиенте. Если же мы получили версию, которая больше текущей, то это новое событие, его нужно отображать, перезаписав текущую версию.

Если же произошел какой-то сбой в Debezium/Kafka/Centrifugo, то все события все-равно будут сохранены, как в Kafka, так и в таблице cdc основной базы данных PostgreSQL. После того, как Debezium/Kafka/Centrifugo восстановятся – все события, которые не было опубликованы – опубликуются. Это происходит за счет того, что мы используем встроенный асинхронный потребитель Centrifugo.

Таким образом, все неопубликованные события будут гарантированно доставлены.

3.5. Гидратация: порядок источников (кэш → сервис постов), размеры батчей, таймауты и деградация

Сначала происходит чтение из кэша (Redis) батчем по post_id и author_id. Если данные есть в кэше, они сразу используются. Если каких-то данных нет в Redis, недостающие элементы добираются из соответствующих сервисов (PostService и UserService), которые читают из основной базы данных. Все данные, полученные из основной БД, после этого записываются обратно в кэш.

Батч - это группировка нескольких элементов в один запрос вместо выполнения множества отдельных запросов. Например, MGET к Redis. К примеру, есть 20 post_id и 20 author_id, тогда вместо 20 отдельных запросов GET post:123, я буду делать один запрос MGET post:1 post:2 ... post:20.

Батч-обработка (например, 20 постов за один запрос ленты) минимизирует накладные расходы на сетевые вызовы и уменьшает число обращений в кэш и в основные сервисы. Для всех внешних запросов задаются явные таймауты (например, 50–100 мс), чтобы предотвратить зависание потоков.

При проблемах с отдельными источниками сервис применяет грациозную деградацию: например, возвращает ленту без части метаданных или без статистики, но не завершает запрос ошибкой 500.

3.6. Бэкап/восстановление и репликация для разных хранилищ

Основная база данных PostgreSQL

Используем потоковую репликацию (одна или несколько реплик), регулярные бэкапы и архивирование WAL с поддержкой восстановления до точки во времени (PITR). Это позволяет восстановить состояние БД даже в случае полной потери основного узла.

Хранилище ленты Cassandra/ScyllaDB

Разворачивается в виде распределённого кластера с репликацией (RF = 3) и snapshot-бэкапами. В случае потери части данных сегменты ленты могут быть частично реконструированы из исторических событий POST_PUBLISHED, полученных из Kafka.

Кэш Redis

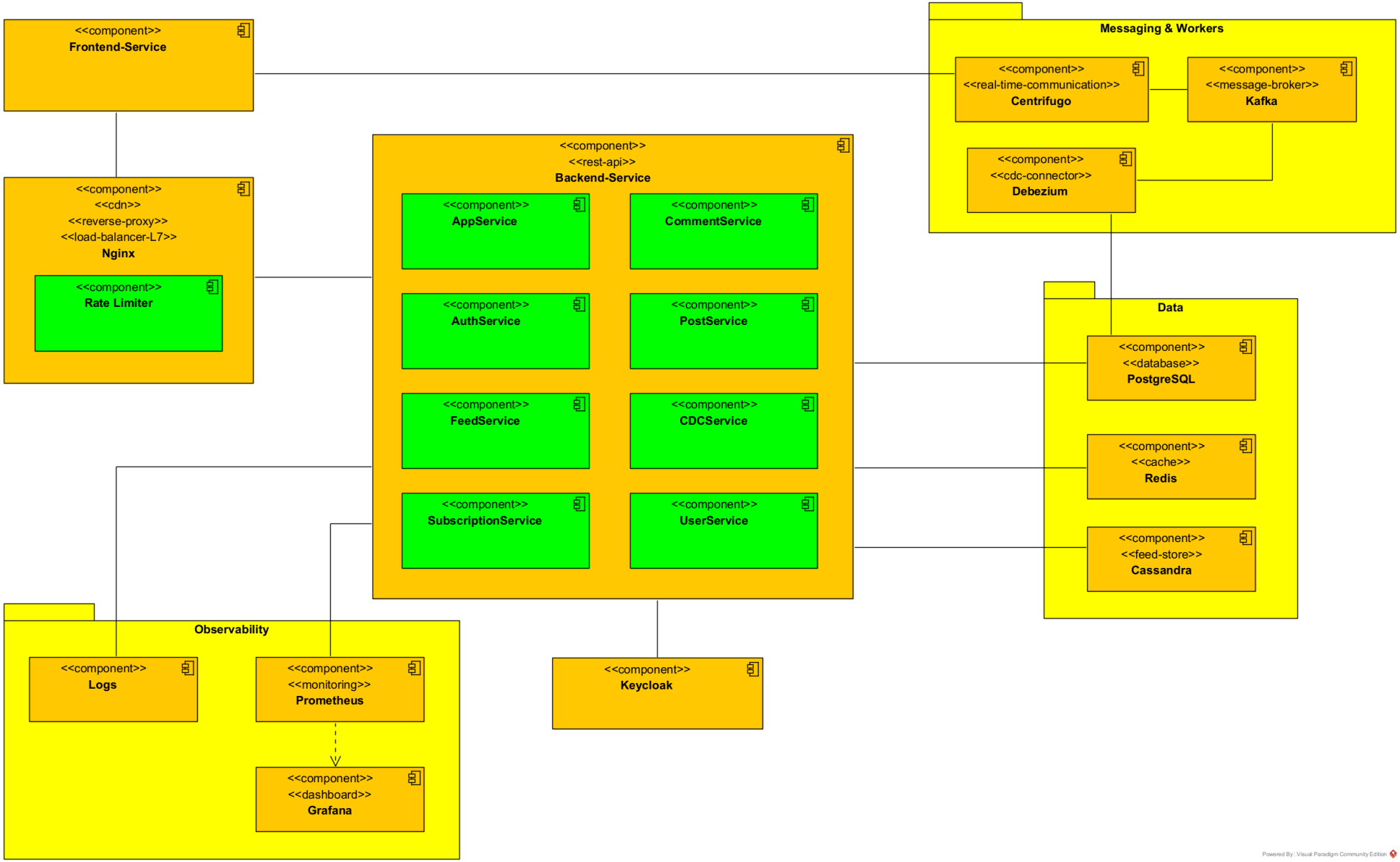
Redis хранит оперативную статистику постов (лайки, дизлайки, количество комментариев), которая периодически синхронизируется с PostgreSQL. В случае сбоя Redis может быть потеряна последняя дельта статистики, но данные не считаются безвозвратными: статистика может быть восстановлена путём периодической агрегации данных из PostgreSQL, либо путём повторного применения событий из Kafka (например, LIKE_CREATED, COMMENT_CREATED). После восстановления Redis автоматически прогревается по мере поступления запросов.

Брокер сообщений Kafka

Kafka настраивается с репликацией (обычно RF = 3) и достаточным retention, чтобы воркеры могли безопасно перечитывать события после сбоя или при перераспределении нагрузки. Репликация обеспечивает отказоустойчивость, а retention — возможность восстановления состояния обработчиков.

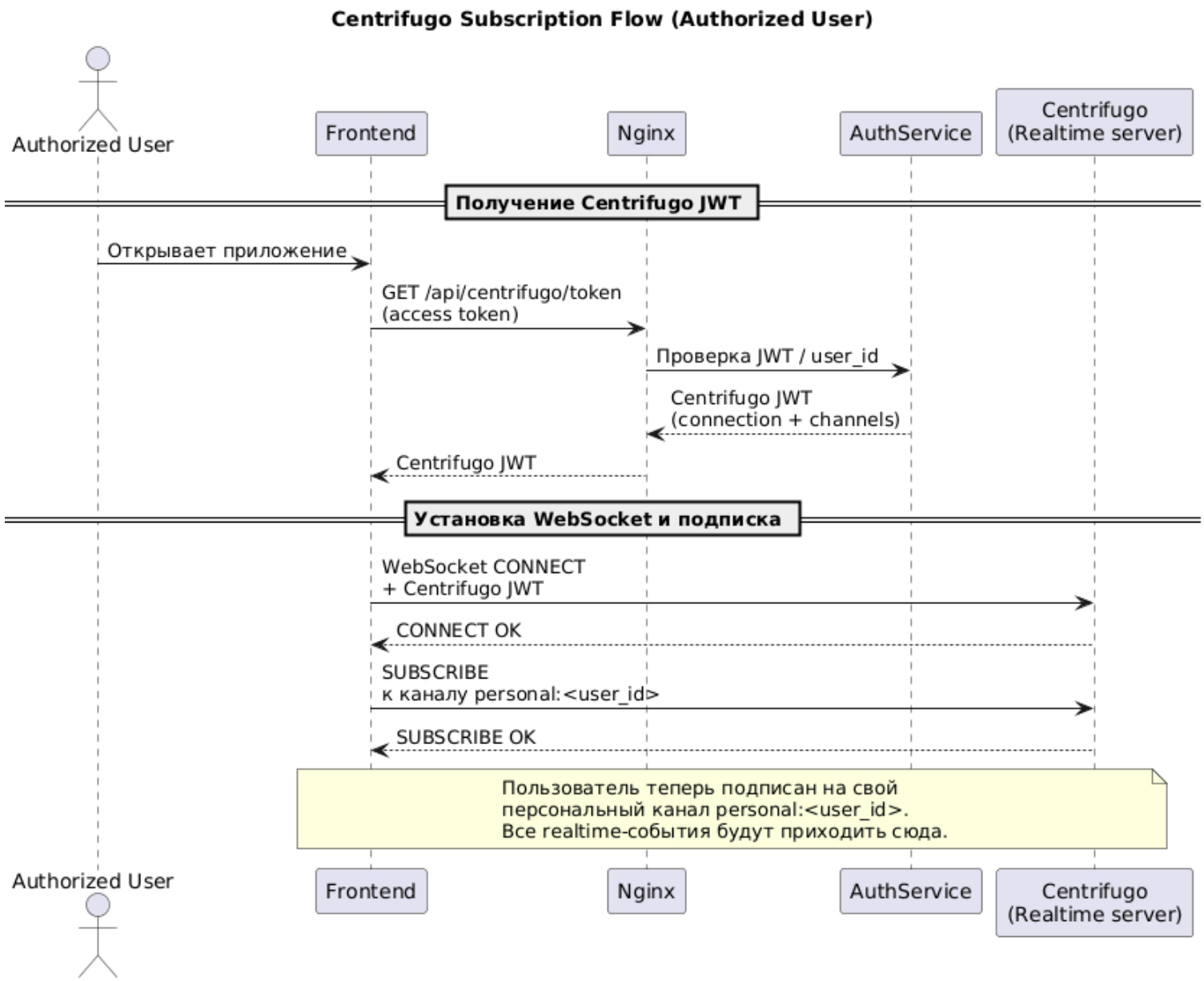
4. UML-диаграммы

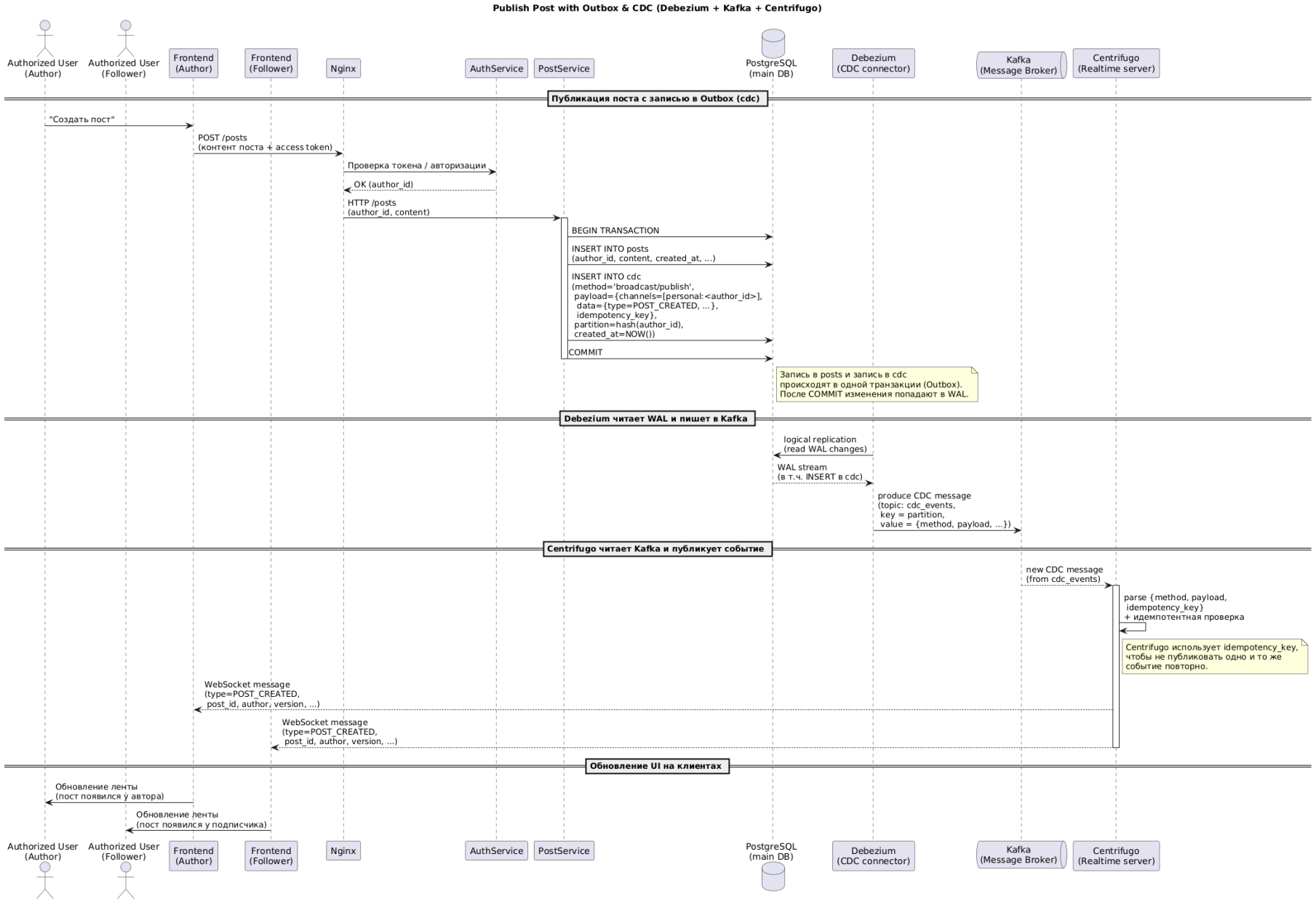
4.1. Component


<<component>>
Frontend-Service
Messaging & Workers
<<component>>
<<real-time-communication>>
Centrifugo
<<component>>
<<message-broker>>
Kafka
<<component>>
<<cdc-connector>>
Debezium
<<component>>
<<cdn>>
<<reverse-proxy>>
<<load-balancer-L7>>
Nginx
<<component>>
Rate Limiter
<<component>>
<<rest-api>>
Backend-Service
<<component>>
AppService
<<component>>
CommentService
<<component>>
AuthService
<<component>>
PostService
<<component>>
FeedService
<<component>>
CDCService
<<component>>
SubscriptionService
<<component>>
UserService
Data
<<component>>
<<database>>
PostgreSQL
<<component>>
<<cache>>
Redis
<<component>>
<<feed-store>>
Cassandra
Observability
<<component>>
Logs
<<component>>
<<monitoring>>
Prometheus
<<component>>
<<dashboard>>
Grafana
<<component>>
Keycloak
Powered By: Visual Paradigm Community Edition

4.2. **Sequence**

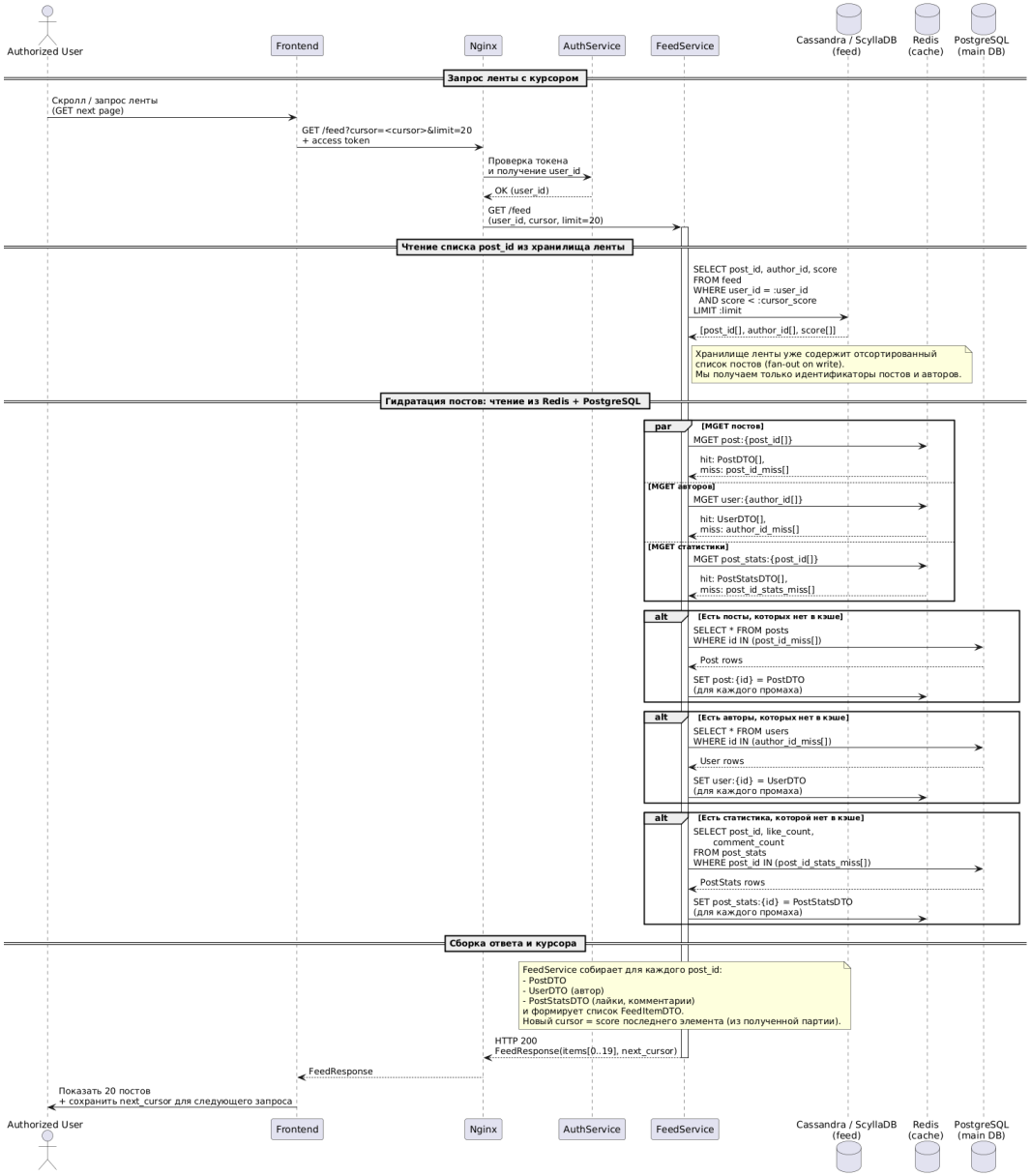
4.2.1. **Публикация поста с Outbox и доставкой события**


Centrifugo Subscription Flow (Authorized User)
Authorized User
Frontend
Nginx
AuthService
Centrifugo
(Realtime server)
Получение Centrifugo JWT
Открывает приложение
GET /api/centrifugo/token
(access token)
Проверка JWT / user_id
Centrifugo JWT
(connection + channels)
Centrifugo JWT
Установка WebSocket и подписка
WebSocket CONNECT
+ Centrifugo JWT
CONNECT OK
SUBSCRIBE
к каналу personal:<user_id>
SUBSCRIBE OK
Пользователь теперь подписан на свой
персональный канал personal:<user_id>.
Все realtime-события будут приходить сюда.
Authorized User
Frontend
Nginx
AuthService
Centrifugo
(Realtime server)

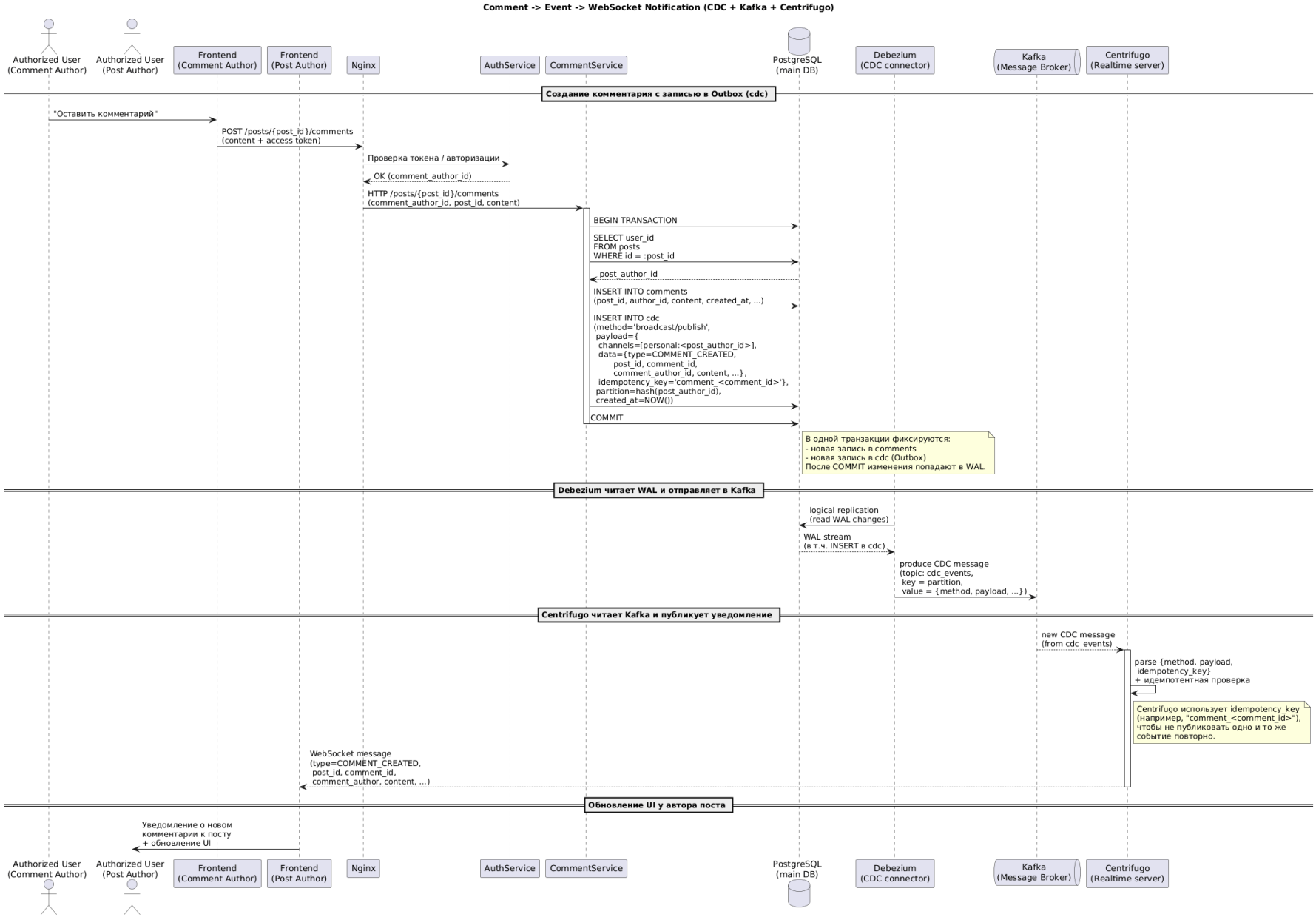

Publish Post with Outbox & CDC (Debezium + Kafka + Centrifugo)
Authorized User (Author)
Authorized User (Follower)
Frontend (Author)
Frontend (Follower)
Nginx
AuthService
PostService
PostgreSQL (main DB)
Debezium (CDC connector)
Kafka (Message Broker)
Centrifugo (Realtime server)
Публикация поста с записью в Outbox (cdc)
"Создать пост"
POST /posts (контент поста + access token)
Проверка токена / авторизации
OK (author_id)
HTTP /posts (author_id, content)
BEGIN TRANSACTION
INSERT INTO posts (author_id, content, created_at, ...)
INSERT INTO cdc (method='broadcast/publish', payload={channels=[personal:<author_id>], data={type=POST_CREATED, ...}, idempotency_key}, partition=hash(author_id), created_at=NOW())
COMMIT
Запись в posts и запись в cdc происходят в одной транзакции (Outbox). После COMMIT изменения попадают в WAL.
Debezium читает WAL и пишет в Kafka
logical replication (read WAL changes)
WAL stream (в т.ч. INSERT в cdc)
produce CDC message (topic: cdc_events, key = partition, value = {method, payload, ...})
Centrifugo читает Kafka и публикует событие
new CDC message (from cdc_events)
parse {method, payload, idempotency_key} + идемпотентная проверка
Centrifugo использует idempotency_key, чтобы не публиковать одно и то же событие повторно.
WebSocket message (type=POST_CREATED, post_id, author, version, ...)
WebSocket message (type=POST_CREATED, post_id, author, version, ...)
Обновление UI на клиентах
Обновление ленты (пост появился у автора)
Обновление ленты (пост появился у подписчика)

4.2.2. **Чтение ленты с курсором и гидратацией**

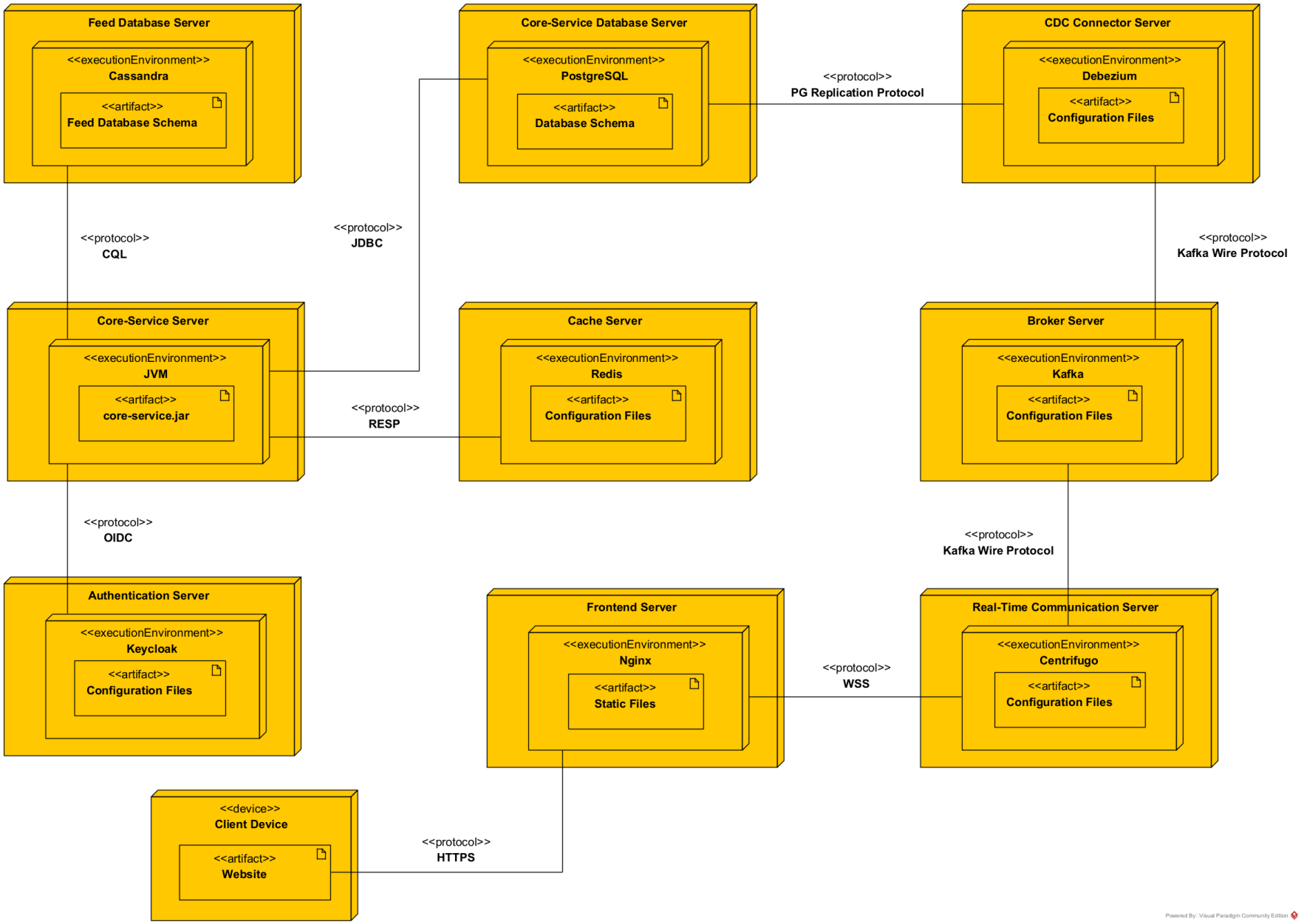
4.2.2. Чтение ленты с курсором и гидратацией



4.2.3. **Комментарий → событие → WebSocket-уведомление**


Comment -> Event -> WebSocket Notification (CDC + Kafka + Centrifugo)
Authorized User (Comment Author)
Authorized User (Post Author)
Frontend (Comment Author)
Frontend (Post Author)
Nginx
AuthService
CommentService
PostgreSQL (main DB)
Debezium (CDC connector)
Kafka (Message Broker)
Centrifugo (Realtime server)
Создание комментария с записью в Outbox (cdc)
"Оставить комментарий"
POST /posts/{post_id}/comments (content + access token)
Проверка токена / авторизации
OK (comment_author_id)
HTTP /posts/{post_id}/comments (comment_author_id, post_id, content)
BEGIN TRANSACTION
SELECT user_id FROM posts WHERE id = :post_id
post_author_id
INSERT INTO comments (post_id, author_id, content, created_at, ...)
INSERT INTO cdc (method='broadcast/publish', payload={ channels=[personal:<post_author_id>], data={type=COMMENT_CREATED, post_id, comment_id, comment_author_id, content, ...}, idempotency_key='comment_<comment_id>'}, partition=hash(post_author_id), created_at=NOW())
COMMIT
В одной транзакции фиксируются:
- новая запись в comments
- новая запись в cdc (Outbox)
После COMMIT изменения попадают в WAL.
Debezium читает WAL и отправляет в Kafka
logical replication (read WAL changes)
WAL stream (в т.ч. INSERT в cdc)
produce CDC message (topic: cdc_events, key = partition, value = {method, payload, ...})
Centrifugo читает Kafka и публикует уведомление
new CDC message (from cdc_events)
parse {method, payload, idempotency_key} + идемпотентная проверка
Centrifugo использует idempotency_key (например, "comment_<comment_id>"), чтобы не публиковать одно и то же событие повторно.
WebSocket message (type=COMMENT_CREATED, post_id, comment_id, comment_author, content, ...)
Обновление UI у автора поста
Уведомление о новом комментарии к посту + обновление UI

4.3. **Deployment**


Feed Database Server
<<executionEnvironment>>
Cassandra
<<artifact>>
Feed Database Schema
Core-Service Database Server
<<executionEnvironment>>
PostgreSQL
<<artifact>>
Database Schema
CDC Connector Server
<<executionEnvironment>>
Debezium
<<artifact>>
Configuration Files
<<protocol>>
PG Replication Protocol
<<protocol>>
CQL
<<protocol>>
JDBC
<<protocol>>
Kafka Wire Protocol
Core-Service Server
<<executionEnvironment>>
JVM
<<artifact>>
core-service.jar
Cache Server
<<executionEnvironment>>
Redis
<<artifact>>
Configuration Files
Broker Server
<<executionEnvironment>>
Kafka
<<artifact>>
Configuration Files
<<protocol>>
RESP
<<protocol>>
OIDC
<<protocol>>
Kafka Wire Protocol
Authentication Server
<<executionEnvironment>>
Keycloak
<<artifact>>
Configuration Files
Frontend Server
<<executionEnvironment>>
Nginx
<<artifact>>
Static Files
Real-Time Communication Server
<<executionEnvironment>>
Centrifugo
<<artifact>>
Configuration Files
<<protocol>>
WSS
<<device>>
Client Device
<<artifact>>
Website
<<protocol>>
HTTPS
Powered By: Visual Paradigm Community Edition

4.4. Class/Data