13. Vẽ và giải thích sơ đồ cho một cơ chế tái tạo lại trạng thái hệ thống từ các sự kiện đã lưu trong thiết kế Event Sourcing của bài lab 03. Vẽ sơ đồ luồng dữ liệu từ sự kiện ban đầu đến trạng thái cuối cùng. Liệt kê công cụ có thể sử dụng và các bước cần thực hiện để tái tạo trạng thái hệ thống.

**Trả lời**

A screenshot of a computer program

AI-generated content may be incorrect.

**1. Sơ đồ cơ chế tái tạo lại trạng thái (State Reconstruction)**

Sơ đồ này minh họa cách hệ thống "tua lại" (replay) các sự kiện để xây dựng lại trạng thái hiện tại của một đối tượng nghiệp vụ (ví dụ: một đơn hàng cụ thể).

**Giải thích sơ đồ:**

1. **Yêu cầu ban đầu**: Một tác nhân (Client hoặc một tiến trình khác) gửi yêu cầu đến **Application Layer**. Yêu cầu này cần biết trạng thái hiện tại của một Aggregate (ví dụ: để cập nhật hoặc chỉ để xem thông tin đơn hàng).
2. **Gọi Event Store**: Application Layer không truy vấn trực tiếp vào một bảng trạng thái. Thay vào đó, nó yêu cầu **Event Store** cung cấp toàn bộ lịch sử (danh sách sự kiện) của Aggregate đó bằng cách truyền vào aggregateId.
3. **Truy vấn sự kiện**: Event Store thực hiện một câu lệnh SELECT vào bảng events trong **PostgreSQL**. Điều quan trọng nhất ở đây là ORDER BY version ASC để đảm bảo các sự kiện được lấy ra theo đúng thứ tự thời gian chúng đã xảy ra.
4. **Nhận danh sách sự kiện**: Application Layer nhận về một mảng các đối tượng sự kiện đã được sắp xếp.
5. **Khởi tạo Aggregate rỗng**: Một đối tượng Order mới được tạo ra, nhưng nó ở trạng thái "trống" hoặc khởi tạo ban đầu.
6. **Tái tạo (Replay/Rehydration)**: Application Layer lặp qua từng sự kiện trong danh sách:
   * Với mỗi sự kiện, nó gọi một phương thức apply(event) trên đối tượng Order.
   * Bên trong Aggregate, phương thức apply sẽ đọc dữ liệu của sự kiện và cập nhật trạng thái nội tại của chính nó. Ví dụ, khi gặp sự kiện OrderStatusUpdated, nó sẽ cập nhật this.status.
7. **Trạng thái cuối cùng**: Sau khi áp dụng sự kiện cuối cùng, đối tượng Order trong bộ nhớ giờ đây đã có được trạng thái chính xác và đầy đủ nhất.
8. **Hoàn thành**: Application Layer bây giờ có thể sử dụng đối tượng Order này để thực hiện logic nghiệp vụ tiếp theo hoặc trả dữ liệu về cho Client.

**2. Sơ đồ luồng dữ liệu (Data Flow Diagram)**

**A diagram of a software system

AI-generated content may be incorrect.**

**Các thành phần chính**

1. **Client**
   * Là phía người dùng hoặc hệ thống gọi API.
   * Gửi yêu cầu Create Order và nhận phản hồi Response New Order.
2. **API Controller**
   * Nhận yêu cầu từ Client.
   * Gửi dữ liệu sang **Application Layer** để xử lý.
   * Sau khi Application Layer truy vấn và xử lý xong, trả kết quả (danh sách sự kiện hoặc Aggregate hiện tại) về cho Client.
3. **Application Layer**
   * Chịu trách nhiệm xử lý logic nghiệp vụ.
   * Gửi aggregateId để truy vấn Event Store.
   * Nhận Event List từ Event Store.
   * Gửi danh sách sự kiện tới **Replay Events** để dựng lại trạng thái hiện tại của Aggregate (ví dụ: trạng thái của một Order).
   * Sau khi tái dựng xong, trả lại kết quả cho API Controller.
4. **Event Store (D)**
   * Kho lưu trữ toàn bộ các sự kiện phát sinh (append-only, không sửa/xóa).
   * Application Layer sẽ thực hiện truy vấn kiểu:
   * SELECT \* FROM events WHERE aggregateId = ? ORDER BY version ASC
   * Trả về danh sách sự kiện liên quan đến aggregateId cần thiết.
5. **Replay Events**
   * Nhận Event List từ Application Layer.
   * Thực hiện việc **replay** (chạy lại toàn bộ sự kiện theo thứ tự version ASC) để dựng lên **Current Aggregate Order**.
   * Kết quả cuối cùng được gửi trả về cho API Controller → Client.

**Luồng dữ liệu**

1. Client gửi yêu cầu Create Order → API Controller.
2. API Controller truyền aggregateId sang Application Layer.
3. Application Layer truy vấn Event Store (SELECT ... ORDER BY version ASC) để lấy Event List.
4. Event Store trả về Event List cho Application Layer.
5. Application Layer gửi Event List cho Replay Events.
6. Replay Events dựng lại trạng thái Current Aggregate Order bằng cách chạy lại các event.
7. Kết quả trạng thái hiện tại hoặc phản hồi tạo mới (new order) được trả về API Controller.
8. API Controller phản hồi kết quả về Client.

**Ý nghĩa**

Sơ đồ này minh họa **nguyên lý Event Sourcing**:

* Thay vì lưu trữ trực tiếp trạng thái cuối cùng (order state), hệ thống chỉ lưu **chuỗi sự kiện** trong **Event Store**.
* Khi cần trạng thái hiện tại, hệ thống sẽ **replay toàn bộ sự kiện** để tái dựng aggregate (ở đây là Order).
* Điều này đảm bảo: tính toàn vẹn lịch sử, khả năng audit, dễ rollback, và dễ dàng mở rộng tính năng.

**3. Công cụ có thể sử dụng**

Việc tái tạo trạng thái chủ yếu dựa vào mã nguồn ứng dụng, không phụ thuộc quá nhiều vào công cụ bên ngoài.

1. **Ngôn ngữ lập trình**: **TypeScript/Node.js** (như trong project).
2. **Cơ sở dữ liệu**: **PostgreSQL** (với bảng events đã được thiết kế).
3. **Thư viện kết nối DB**: pg cho Node.js.
4. **(Tùy chọn) Caching Layer**: **Redis** hoặc **Memcached**. Nếu một aggregate có quá nhiều sự kiện và được truy cập thường xuyên, việc tái tạo trạng thái liên tục có thể tốn kém. Chúng ta có thể lưu "snapshot" (ảnh chụp nhanh trạng thái) của aggregate vào cache sau lần tái tạo đầu tiên để các lần truy cập sau nhanh hơn.

**4. Các bước cần thực hiện để tái tạo trạng thái**

Đây là các bước cụ thể trong mã nguồn để hiện thực hóa cơ chế này.

**Bước 1: Viết phương thức getEvents trong Event Store**

**Mục tiêu**: Lấy danh sách sự kiện đã được sắp xếp từ DB.

**File**: [postgres-event-store.ts](vscode-file://vscode-app/c:/Users/Admin/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html)

// ... trong class PostgresEventStore

async getEvents(aggregateId: string): Promise<OrderEvent[]> {

  const client = await this.pool.connect();

  try {

    const result = await client.query(

      `SELECT id, aggregate\_id, event\_type, event\_data, version, timestamp

       FROM events

       WHERE aggregate\_id = $1

       ORDER BY version ASC`, // Rất quan trọng!

      [aggregateId]

    );

    if (result.rows.length === 0) {

      return [];

    }

    // Chuyển đổi row data thành các đối tượng Event tường minh

    return result.rows.map(row => ({

      id: row.id,

      aggregateId: row.aggregate\_id,

      type: row.event\_type,

      data: row.event\_data, // JSONB được tự động parse

      version: row.version,

      timestamp: row.timestamp,

    }));

  } finally {

    client.release();

  }

}

**Bước 2: Tạo phương thức apply trong Domain Aggregate**

**Mục tiêu**: Cung cấp logic để Aggregate tự cập nhật trạng thái từ một sự kiện.

**File**: [Order.ts](vscode-file://vscode-app/c:/Users/Admin/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html)

// ... trong class Order

public apply(event: OrderEvent): void {

  switch (event.type) {

    case 'OrderCreated':

      // Constructor đã xử lý việc này, nhưng id cần được gán

      this.id = event.aggregateId;

      this.customerId = event.data.customerId;

      this.items = event.data.items;

      this.status = event.data.status;

      this.totalAmount = event.data.totalAmount;

      break;

    case 'OrderItemAdded':

      this.items.push(event.data.item);

      this.totalAmount = this.calculateTotal(); // Tính lại tổng tiền

      break;

    case 'OrderStatusUpdated':

      this.status = event.data.newStatus;

      break;

    case 'OrderCancelled':

      this.status = 'CANCELLED';

      break;

    // Thêm các case cho các sự kiện khác...

  }

}

// Thêm một constructor rỗng hoặc một static factory để tạo đối tượng ban đầu

static createEmpty(): Order {

    // Trả về một đối tượng Order với các giá trị mặc định, chưa hợp lệ

    return new Order('', [], 'INITIAL');

}

**Bước 3: Viết hàm tái tạo trạng thái trong Application Layer**

**Mục tiêu**: Điều phối toàn bộ quá trình: lấy events, tạo aggregate, và lặp để áp dụng.

**File**: Order-management/src/application/command-handlers.ts (hoặc một file helper riêng)

// ... trong class OrderCommandHandlers hoặc một service riêng

private async rebuildOrderState(orderId: string): Promise<Order> {

  // 1. Lấy toàn bộ lịch sử sự kiện

  const events = await this.eventStore.getEvents(orderId);

  if (events.length === 0) {

    throw new Error(`Order with ID ${orderId} not found.`);

  }

  // 2. Tạo một thực thể Order rỗng

  const order = Order.createEmpty();

  // 3. "Tua lại" các sự kiện

  events.forEach(event => order.apply(event));

  // 4. Trả về aggregate với trạng thái đầy đủ

  return order;

}

// Cách sử dụng:

public async handleUpdateOrderStatus(command: { orderId: string; newStatus: string }) {

  // Lấy trạng thái hiện tại trước khi thực hiện logic

  const order = await this.rebuildOrderState(command.orderId);

  // Bây giờ có thể thực hiện các quy tắc nghiệp vụ trên trạng thái hiện tại

  order.updateStatus(command.newStatus); // Giả sử phương thức này có logic kiểm tra

  // ... Tạo và lưu sự kiện OrderStatusUpdated

}