Trả lời câu hỏi lý thuyết

Câu 1: Mô tả chi tiết quy trình OAuth 2.0 trong Bitrix24, bao gồm các bước lấy access token, refresh token, và cách xử lý khi token hết hạn. Đề xuất một chiến lược để quản lý token trong ứng dụng của bạn (ví dụ: lưu trữ, làm mới, bảo mật).

Quy trình OAuth 2.0 trong Bitrix24

1. Hiển thị form đăng nhập và thu thập domain Bitrix24

- **Frontend (Laravel):** Tệp resources/views/auth/login.blade.php cung cấp giao diện đăng nhập, cho phép người dùng nhập domain Bitrix24 (ví dụ: yourcompany.bitrix24.vn). Form sử dụng regex (^[a-zA-Z0-9.-]+\.bitrix24\.(vn|com)$) để validate domain, đảm bảo chỉ chấp nhận các domain hợp lệ. Khi người dùng gửi form, yêu cầu được gửi tới route /connect, được xử lý bởi App\Http\Controllers\AuthController::redirectToBitrix

Tạo và chuyển hướng tới URL xác thực Bitrix24

- **Frontend (Laravel):** Trong AuthController::redirectToBitrix, domain được lưu vào session Laravel (Session::put('domain', $domain)). Sau đó, yêu cầu GET được gửi tới endpoint /api/auth/redirect của backend (NestJS) với tham số domain
- **Backend (NestJS):** Trong AuthController::redirectToBitrix (src/auth/auth.controller.ts), hàm gọi AuthService::generateAuthUrl để tạo URL xác thực:

  ```
  https://<domain>/oauth/authorize?client_id=<BITRIX24_CLIENT_ID>&state=<state>
  ```

  - client_id được lấy từ biến môi trường BITRIX24_CLIENT_ID (qua ConfigService)
  - state là một UUID được tạo bởi uuidv4() để chống tấn công CSRF. State này được lưu vào Redis với key state:<domain> và TTL 600 giây (AuthStateService::save)
- URL xác thực được trả về và người dùng được chuyển hướng tới trang đăng nhập Bitrix24

Xử lý callback từ Bitrix24

- **Backend (NestJS):** Sau khi người dùng xác thực trên Bitrix24, Bitrix24 chuyển hướng về endpoint /api/auth/callback với các query parameters: code, state, và domain
  - Xác minh state: AuthService::validateState kiểm tra state nhận được so với state đã lưu trong Redis (AuthStateService::validate). Nếu không khớp, ném BadRequestException với thông báo "Invalid state token”
  - Trao đổi code lấy token: AuthService::exchangeCodeForToken gửi yêu cầu POST tới:

    ```
    https://oauth.bitrix.info/oauth/token/?grant_type=authorization_code&client_id=<clientId
    ```

    Phản hồi từ Bitrix24 bao gồm access_token, refresh_token, expires_in, và member_id
  - Lưu token: AuthTokenService::saveToken mã hóa dữ liệu token (gồm access_token, refresh_token, expires_in, domain, created_at) bằng CryptoService và lưu vào Redis với key token:<memberId> và TTL bằng expires_in
  - Tạo session: AuthSessionService::create tạo một sessionToken (UUID) và lưu vào Redis với key session:<sessionToken> (giá trị là memberId, TTL 600 giây). sessionToken cũng được thêm vào Redis Set sessionSet:<memberId> để quản lý nhiều phiên
  - Chuyển hướng về client: sessionToken được lưu vào cookie session_token (HTTP-only, secure, SameSite=Strict, maxAge 10 phút) và chuyển hướng tới CLIENT_REDIRECT_URL/auth/callback?session=<sessionToken>
- **Frontend (Laravel):** Trong AuthController::handleCallback, sessionToken được lấy từ query parameter. Yêu cầu GET được gửi tới /api/auth/member để lấy memberId và /api/auth/domain để lấy domain. Các giá trị session_token, member_id, và domain được lưu vào session Laravel. Người dùng được chuyển hướng tới /leads

Xử lý token hết hạn và làm mới token

- **Backend (NestJS):** AuthService::ensureValidAccessToken được gọi bởi OAuthGuard để kiểm tra tính hợp lệ của access_token
  - AuthTokenService::getAccessToken lấy token từ Redis và kiểm tra thời gian hết hạn (created_at + expires_in). Nếu token còn hiệu lực, trả về access_token
  - Nếu token hết hạn, AuthService::refreshToken gửi yêu cầu POST tới:

    ```
    https://oauth.bitrix.info/oauth/token/?grant_type=refresh_token&client_id=<clientId>&cl
    ```

    Token mới (access_token, refresh_token, expires_in) được lưu lại vào Redis (AuthTokenService::saveToken)
  - Nếu không thể làm mới (do thiếu refresh_token hoặc lỗi), ném UnauthorizedException, yêu cầu người dùng đăng nhập lại

- OAuthGuard đảm bảo mọi request tới các endpoint bảo vệ đều có access_token hợp lệ, gán vào request.bitrixAccessToken

Đăng xuất

- **Frontend (Laravel):** AuthController::logout gửi yêu cầu POST tới /api/auth/logout, xóa toàn bộ session Laravel (Session::flush) và chuyển hướng về /login với thông báo "Bạn đã đăng xuất thành công”
- **Backend (NestJS):** AuthController::logout xóa sessionToken (AuthSessionService::delete), xóa token trong Redis (AuthTokenService::delete), và xóa cookie session_token

Chiến lược quản lý token

1. Lưu trữ token

- **Access và refresh token**: Được mã hóa bằng CryptoService và lưu trong Redis với key token:<memberId>. TTL được đặt bằng expires_in từ phản hồi của Bitrix24
- **Session token**: Được lưu trong Redis với key session:<sessionToken> (giá trị là memberId, TTL 600 giây). Redis Set sessionSet:<memberId> được sử dụng để theo dõi tất cả sessionToken của một người dùng
- **Cookie**: session_token được lưu trong cookie với thuộc tính HTTP-only, secure, và SameSite=Strict, đảm bảo an toàn trước các cuộc tấn công CSRF và XSS
- **Laravel session**: Các giá trị session_token, member_id, và domain được lưu trong session Laravel để sử dụng trong các request tiếp theo

Làm mới token

- AuthService::ensureValidAccessToken tự động kiểm tra và làm mới access_token nếu hết hạn, sử dụng refresh_token lưu trong Redis
- Yêu cầu làm mới được gửi với client_id, client_secret, và refresh_token. Token mới được lưu lại vào Redis, đảm bảo tính liên tục của phiên người dùng
- Bottleneck được sử dụng để giới hạn 2 request đồng thời, khoảng cách tối thiểu 333ms, với tối đa 3 lần thử lại cho lỗi 429 (rate limit) hoặc 5xx

Bảo mật

- **Chống CSRF**: Sử dụng state (UUID) được lưu trong Redis và xác minh trong callback (AuthStateService::validate) để ngăn chặn tấn công CSRF
- **Mã hóa token**: Tất cả token được mã hóa trước khi lưu vào Redis (CryptoService), đảm bảo dữ liệu không bị truy cập trái phép nếu Redis bị xâm phạm
- **Cookie bảo mật**: Cookie session_token được cấu hình với HTTP-only, secure, và SameSite=Strict để giảm nguy cơ XSS và CSRF

- **Middleware xác thực**: AuthenticateBitrix middleware kiểm tra sự tồn tại của domain, session_token, và member_id trong session Laravel, chuyển hướng về /login nếu thiếu
- **Rate limiting**: Bottleneck được tích hợp trong AuthService để giới hạn số lượng request tới Bitrix24 API, tránh vi phạm giới hạn rate limit
- **Xử lý lỗi**: Các lỗi như token hết hạn (401), state không hợp lệ, hoặc lỗi server (5xx) được xử lý bằng cách chuyển hướng về trang đăng nhập với thông báo lỗi rõ ràng

---

Câu 2. Giải thích lợi ích của batch request trong Bitrix24 REST API so với các request riêng lẻ. Cung cấp một ví dụ cụ thể về cách sử dụng batch để lấy danh sách lead, task, và deal liên quan trong một lần gọi. Xử lý lỗi trong batch request như thế nào?

Lợi ích của batch request trong Bitrix24 REST API

- **Giảm số lượng request:** Batch request cho phép gửi nhiều lệnh API (như crm.lead.list, crm.lead.fields, crm.status.list) trong một lần gọi HTTP duy nhất, giảm tải cho server Bitrix24 và ứng dụng
- **Tăng hiệu suất:** Bằng cách gộp các lệnh vào một request, thời gian phản hồi tổng thể giảm đáng kể do loại bỏ các chi phí kết nối HTTP lặp lại (handshake, latency)
- **Tôn trọng giới hạn rate limit:** Bitrix24 áp dụng giới hạn về số lượng request mỗi giây. Batch request được tính là một request duy nhất, giúp giảm nguy cơ vượt quá giới hạn rate limit (429 Too Many Requests)
- **Đồng bộ dữ liệu:** Batch request đảm bảo dữ liệu từ các lệnh (leads, fields, statuses, sources) được lấy cùng một thời điểm, tránh tình trạng dữ liệu không nhất quán do độ trễ giữa các request riêng lẻ

Ví dụ cụ thể về sử dụng batch request

Trong LeadsService::getLeads (leads.service.ts), đã triển khai batch request để lấy danh sách lead cùng các thông tin liên quan trong một lần gọi. Dưới đây là ví dụ cụ thể dựa trên mã nguồn:

- **Mã nguồn tham chiếu:**

```
const url = `https://${domain}/rest/batch`;
const cmd = this.buildLeadBatchQuery({ ...dto, start });
const response = await firstValueFrom(
  this.httpService.post(
    url,
    { halt: 0, cmd },
    { headers: { Authorization: `Bearer ${token}` } },
  ),
);
```

- **Chi tiết batch query (từ buildLeadBatchQuery):**

```
private buildLeadBatchQuery(dto: QueryLeadDto & { start?: number }) {
  const filter: Record<string, any> = {};
  const order: Record<string, string> = {};
  if (dto.find) {
    filter['LOGIC'] = 'OR';
    filter['%TITLE'] = dto.find;
    filter['%NAME'] = dto.find;
    filter['%EMAIL'] = dto.find;
  }
  if (dto.status) filter['STATUS_ID'] = dto.status;
  if (dto.source) filter['SOURCE_ID'] = dto.source;
  if (dto.date) {
    const parsedDate = new Date(dto.date).toISOString().split('T')[0] + ' 00:00:00';
    filter['>=DATE_CREATE'] = parsedDate;
  }
  const sortField = dto.sort || 'DATE_CREATE';
  order[sortField] = 'DESC';
  const query = qs.stringify(
    { filter, order, start: dto.start || 0, select: ['*', 'EMAIL', 'PHONE'] },
    { encode: true, arrayFormat: 'brackets', allowDots: true },
  );
  return {
    leads: `crm.lead.list?${query}`,
    fields: 'crm.lead.fields',
    statuses: 'crm.status.list?filter[ENTITY_ID]=STATUS',
    sources: 'crm.status.list?filter[ENTITY_ID]=SOURCE',
  };
}
```

- **Giải thích:**
  - **Endpoint:** https://${domain}/rest/batch
  - **Payload:** Gửi một object cmd chứa bốn lệnh:
    - leads: Gọi crm.lead.list với các bộ lọc (filter) cho TITLE, NAME, EMAIL (tìm kiếm từ khóa), STATUS_ID, SOURCE_ID, DATE_CREATE, và sắp xếp theo DATE_CREATE (hoặc trường khác được chỉ định)
    - fields: Gọi crm.lead.fields để lấy thông tin custom fields
    - statuses: Gọi crm.status.list?filter[ENTITY_ID]=STATUS để lấy danh sách trạng thái lead
    - sources: Gọi crm.status.list?filter[ENTITY_ID]=SOURCE để lấy danh sách nguồn lead
  - **Kết quả:** Phản hồi từ Bitrix24 chứa một object result.result với dữ liệu tương ứng (data.leads, data.fields, data.statuses, data.sources). Kết quả được lưu vào Redis với TTL 600 giây để caching

Xử lý lỗi trong batch request

Trong LeadsService::getLeads, đã xử lý lỗi batch request như sau:

- **Kiểm tra lỗi trong phản hồi:**

```
const error = response.data?.result?.error;
if (error) {
  this.logger.error(`Batch API error`, { memberId, error });
  throw new HttpException(
    error?.leads?.error_description || 'Failed to fetch leads',
    error?.leads?.error || HttpStatus.INTERNAL_SERVER_ERROR,
  );
}
```

  Nếu phản hồi chứa error (ví dụ: lỗi từ lệnh leads), ném HttpException với mô tả lỗi cụ thể (error.leads.error_description) hoặc mã lỗi mặc định (500)

- **Rate limiting và retry:**

  Sử dụng Bottleneck trong LeadsService với cấu hình:

```
private readonly limiter = new Bottleneck({
  maxConcurrent: 2,
  minTime: 333,
  retryOptions: {
    maxRetries: 3,
```

```
    delay: (retryCount) => retryCount * 1000,
    retryOn: (error: AxiosError) => {
      const status = error?.response?.status;
      const shouldRetry = status === 429 || (status && status >= 500);
      return shouldRetry;
    },
  },
});
```

Nếu gặp lỗi 429 (Too Many Requests) hoặc 5xx (server error), Bottleneck tự động thử lại tối đa 3 lần với độ trễ tăng dần (1s, 2s, 3s)

- **Xử lý lỗi token:**

  Trong executeWithRetry, nếu request thất bại với mã lỗi 401 (Unauthorized), bạn gọi authService.refreshToken để làm mới token và thử lại request:

```
if (error.response?.status === 401) {
  this.logger.warn(`Access token expired for memberId: ${memberId}, attempting refresh`);
  const newToken = await this.authService.refreshToken(memberId);
  return await fn(newToken);
}
```

- **Logging:**

  Mọi lỗi được ghi lại qua nest-winston (logger.error), bao gồm thông tin chi tiết về memberId, error, và stack trace để hỗ trợ debug

---

Câu 3. Tại sao cần xử lý rate limiting khi làm việc với Bitrix24 REST API? Đề xuất một chiến lược tối ưu để tránh vi phạm giới hạn này, bao gồm cả việc sử dụng caching và hàng đợi bất đồng bộ.

Tại sao cần xử lý rate limiting khi làm việc với Bitrix24 REST API

- **Giới hạn của Bitrix24 API:** Bitrix24 áp dụng giới hạn về số lượng request mỗi giây (theo tài liệu https://apidocs.bitrix24.com). Vượt quá giới hạn này dẫn đến lỗi 429 (Too Many Requests), làm gián đoạn hoạt động của ứng dụng, như bạn đã xử lý trong LeadsService và AuthService
- **Đảm bảo hiệu suất và ổn định:**

    - Các request liên tục và không kiểm soát có thể gây quá tải cho server Bitrix24, dẫn đến độ trễ cao hoặc lỗi server (5xx)
    - Trong LeadsService, bạn sử dụng Bottleneck để giới hạn số lượng request, đảm bảo ứng dụng không gửi quá nhiều request đồng thời, từ đó duy trì hiệu suất ổn định
- **Trải nghiệm người dùng:** Nếu không xử lý rate limiting, lỗi 429 có thể gây gián đoạn trải nghiệm, như khi người dùng cố gắng lấy danh sách lead hoặc xử lý webhook. Việc retry và queuing giúp giảm thiểu vấn đề này
- **Hỗ trợ nhiều người dùng:** Ứng dụng hỗ trợ nhiều memberId, mỗi người có thể gửi nhiều request. Rate limiting giúp đảm bảo công bằng và tránh vượt quá giới hạn API cho toàn bộ hệ thống

Chiến lược tối ưu để tránh vi phạm giới hạn rate limiting

- **Sử dụng Bottleneck cho rate limiting:**
    - **Triển khai:**

      ```
      private readonly limiter = new Bottleneck({
        maxConcurrent: 2, // Tối đa 2 request đồng thời
        minTime: 333, // Khoảng cách tối thiểu 333ms giữa các request
        retryOptions: {
          maxRetries: 3,
          delay: (retryCount) => retryCount * 1000,
          retryOn: (error: AxiosError) => {
            const status = error?.response?.status;
            return status === 429 || (status && status >= 500);
          },
        },
      });
      ```

      Mọi request tới Bitrix24 API (như crm.lead.list, crm.lead.get, tasks.task.add) được xếp hàng qua limiter.schedule, đảm bảo không vượt quá 2 request đồng thời và có khoảng cách 333ms
    - **Lợi ích:** Ngăn chặn lỗi 429 bằng cách kiểm soát tần suất request, đồng thời tự động thử lại nếu gặp lỗi 429 hoặc 5xx
- **Caching với Redis:**
    - **Triển khai:**

      ```
      const cacheKey = buildCacheKey(memberId, dto);
      const cached = await this.redisService.get(cacheKey);
      ```

```
if (cached) {
  try {
    const parsed = JSON.parse(cached);
    this.logger.debug(`Cache hit for leads: ${cacheKey}`, { memberId });
    return parsed;
  } catch (error) {
    await this.redisService.del(cacheKey);
  }
}

await this.redisService.set(cacheKey, JSON.stringify(result), 600);
```

        - Cache được xóa khi lead được thêm, cập nhật, hoặc xóa (redisService.delByPrefix('leads:${memberId}')) để đảm bảo dữ liệu mới nhất
        - Webhook (WebhookController::handleWebhook) cũng xóa cache liên quan khi nhận sự kiện ONCRMLEADADD hoặc ONCRMLEADUPDATE
    - **Lợi ích:**
        - Giảm số lượng request tới Bitrix24 API bằng cách trả về dữ liệu từ cache nếu còn hợp lệ
        - Cải thiện hiệu suất, đặc biệt khi người dùng truy vấn danh sách lead nhiều lần với cùng bộ lọc
- **Hàng đợi bất đồng bộ với Bull:**
    - **Triển khai:**
        - Trong WebhookModule, sử dụng Bull để xử lý webhook bất đồng bộ

```
@Module({
  imports: [
    BullModule.registerQueue({ name: 'webhook' }),
  ],
})
```

        - Trong WebhookService::processWebhook, webhook được đẩy vào hàng đợi webhook

```
await this.webhookQueue.add('handle-lead-webhook', {
  event,
  data,
  memberId: data.memberId,
});
```

        - WebhookProcessor::handleLeadWebhook xử lý các sự kiện như ONCRMLEADADD (tạo task, gửi thông báo) và ONCRMLEADUPDATE (tạo deal) trong hàng đợi.
    - **Lợi ích:**
        - Xử lý webhook bất đồng bộ giúp giảm tải tức thời lên Bitrix24 API, đặc biệt khi có nhiều sự kiện đồng thời
        - Đảm bảo các tác vụ nặng (như tạo task, gửi thông báo) không chặn luồng chính, tránh lỗi 429 hoặc timeout
- **Xử lý lỗi và retry:**
    - Trong WebhookProcessor::fetchWithRetry, retry request nếu gặp lỗi 401 (token hết hạn) bằng cách làm mới token

      ```
      if (error.response?.status === 401) {
        this.logger.warn(`Retrying ${action} due to 401 for lead ${leadId}`);
        token = await this.authService.refreshToken(memberId);
        const retryResponse = await firstValueFrom(/* retry request */);
        return retryResponse.data.result;
      }
      ```

      Bottleneck tự động retry cho lỗi 429 hoặc 5xx, giảm nguy cơ request thất bại do giới hạn API

Câu 4. Trong bối cảnh ứng dụng này, làm thế nào để đảm bảo tính bảo mật khi xử lý webhook từ Bitrix24? Mô tả ít nhất ba kỹ thuật bảo mật (ví dụ: xác minh chữ ký, giới hạn IP, mã hóa dữ liệu).

Xác minh webhook qua auth token

- Trong WebhookController::handleWebhook, kiểm tra sự tồn tại của webhookSecret (lấy từ body.auth.client_endpoint hoặc body.auth.access_token)

  ```
  const webhookSecret = body.auth?.client_endpoint || body.auth?.access_token;
  if (!webhookSecret) {
    this.logger.error(`Missing webhook auth`);
    throw new HttpException(
      'Invalid webhook request: Missing auth',
  ```

```
    HttpStatus.UNAUTHORIZED,
  );
}
```

- Kiểm tra thêm sự hợp lệ của payload (event, data, auth.member_id)

```
if (!event || !data || !auth?.member_id) {
  this.logger.error(`Invalid webhook payload`, { payload: body });
  throw new HttpException(
    'Invalid webhook payload',
    HttpStatus.BAD_REQUEST,
  );
}
```

- **Lợi ích:** Đảm bảo webhook đến từ Bitrix24 hợp lệ bằng cách yêu cầu thông tin xác thực (client_endpoint hoặc access_token) và kiểm tra các trường bắt buộc, ngăn chặn các request giả mạo hoặc không hợp lệ

Lưu trữ và ghi log webhook vào database

- Trong WebhookProcessor::handleLeadWebhook, lưu thông tin webhook vào database sử dụng TypeORM (WebhookLog entity)

```
const log = this.webhookLogRepository.create({
  event,
  payload: JSON.stringify(data),
  memberId,
  createdAt: new Date(),
});
await this.webhookLogRepository.save(log);
```

- Endpoint GET /api/webhook/logs cho phép truy vấn log với phân trang và lọc theo event, leadId, memberId

```
const [logs, total] = await this.webhookLogRepository.findAndCount({
  where,
  order: { createdAt: 'DESC' },
  skip,
  take: limit,
});
```

- **Lợi ích:**

- Lưu trữ log webhook giúp debug và theo dõi các sự kiện, đặc biệt khi cần kiểm tra tính hợp lệ hoặc phát hiện các request bất thường
- Bảo vệ endpoint log bằng OAuthGuard, đảm bảo chỉ người dùng đã xác thực (memberId) mới truy cập được

Xử lý bất đồng bộ qua hàng đợi Bull

- Webhook được đẩy vào hàng đợi webhook thông qua WebhookService::processWebhook

```
await this.webhookQueue.add('handle-lead-webhook', {
  event,
  data,
  memberId: data.memberId,
});
```

- WebhookProcessor::handleLeadWebhook xử lý các tác vụ nặng (tạo task, gửi thông báo, tạo deal) bất đồng bộ, sử dụng fetchWithRetry để đảm bảo xác thực token

```
const fetchWithRetry = async <T>(url: string, options: any, action: string): Promise<T> => {
  let token = await this.authService.getAccessToken(memberId);
  const domain = await this.authService.getDomain(memberId);
  try {
    const response = await firstValueFrom(
      this.httpService.request({
        ...options,
        url: `https://${domain}${url}`,
        headers: { Authorization: `Bearer ${token}`, ...options.headers },
      }),
    );
    return response.data.result;
  } catch (error) {
    if (error.response?.status === 401) {
      token = await this.authService.refreshToken(memberId);
      const retryResponse = await firstValueFrom(/* retry request */);
      return retryResponse.data.result;
    }
    throw error;
  }
};
```

- **Lợi ích:**

- Xử lý bất đồng bộ giúp giảm nguy cơ request giả mạo gây quá tải hệ thống, vì webhook được xếp hàng và xử lý tuần tự
- Retry token khi gặp lỗi 401 đảm bảo tính liên tục, đồng thời giảm nguy cơ xử lý request không hợp lệ do token hết hạn