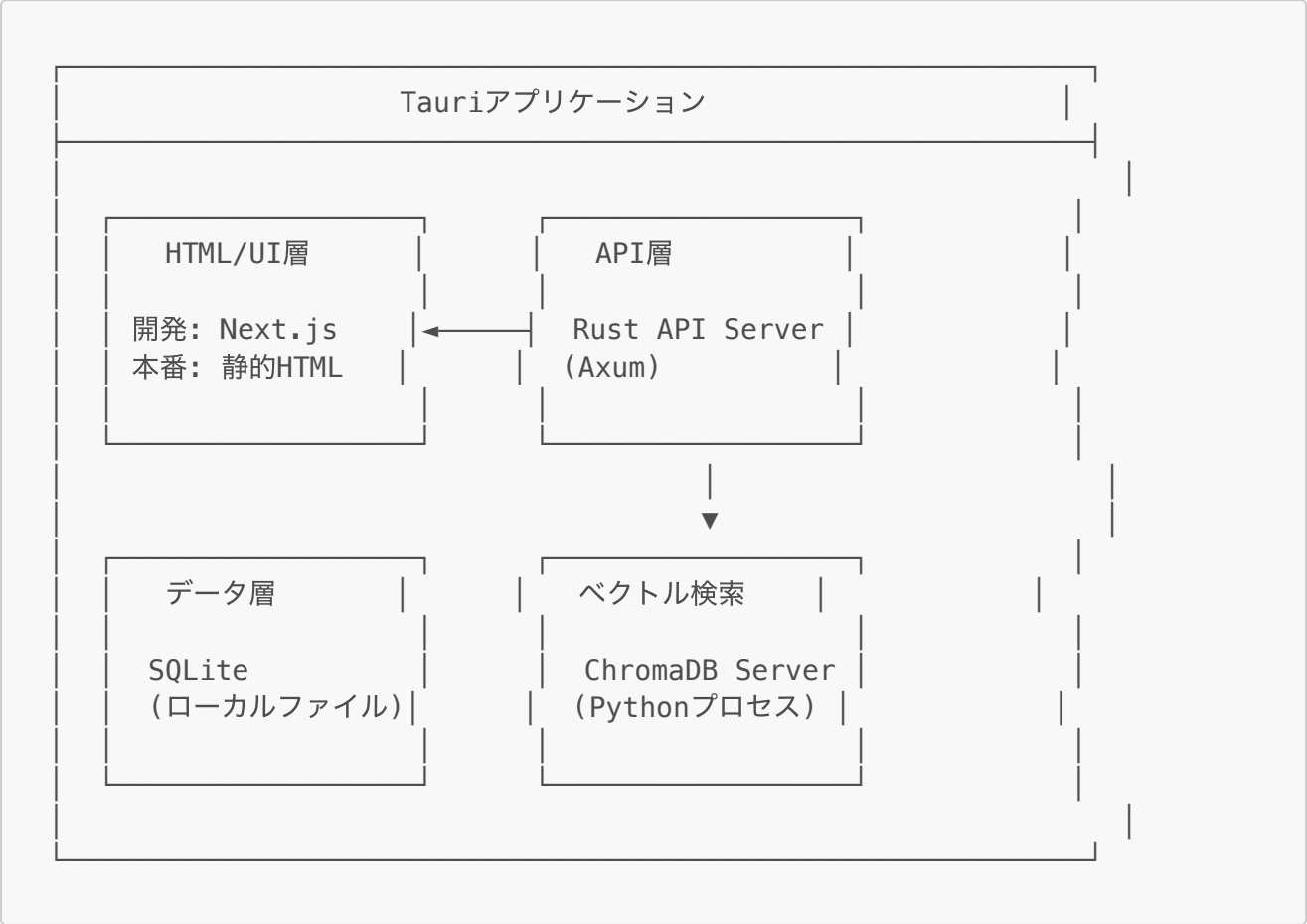
ポート設計とサーバー構成の設計書

概要

このドキュメントでは、開発環境と配布環境（本番環境）の両方を考慮した、ポート設計とサーバー構成を定義します。

アーキテクチャの全体像



ポート設計

開発環境（Development）

サービス	ポート	プロトコル	説明
Next.js開発サーバー	**3010**	HTTP	フロントエンドの開発サーバー（HMR対応）
Rust APIサーバー	**3011**	HTTP	バックエンドAPIサーバー
ChromaDB Server	**8000**	HTTP	ベクトル検索サーバー（Pythonプロセス）

特徴:

- Next.jsとRust APIサーバーは**異なるポート**を使用（競合回避）
- フロントエンドは`http://localhost:3010`からNext.js開発サーバーに接続

- フロントエンドは`http://localhost:3011`からRust APIサーバーに接続
- ChromaDBは`http://localhost:8000`で起動

配布環境（Production / Release）

サービス	ポート	プロトコル	説明
HTML配信	**カスタムプロトコル**	`tauri://localhost`	Tauriカスタムプロトコル（静的ファイル）
Rust APIサーバー	**3011**	HTTP	バックエンドAPIサーバー
ChromaDB Server	**8000**	HTTP	ベクトル検索サーバー（Pythonプロセス）

特徴:

- Next.jsは不要（静的エクスポート済み）
- HTMLはTauriのカスタムプロトコル（`tauri://localhost`）で配信
- フロントエンドは`http://localhost:3011`からRust APIサーバーに接続
- ChromaDBは`http://localhost:8000`で起動

HTMLサーバーの実装方式

開発環境

Next.js開発サーバー（ポート3010）

- ホットリロード（HMR）対応
- 開発者ツール（React DevToolsなど）が使用可能
- ソースマップが有効
- Tauriウィンドウは`http://localhost:3010`を表示

設定:

```
// tauri.conf.dev.json
{
  "build": {
    "beforeDevCommand": "npm run dev",
    "devUrl": "http://localhost:3010"
  },
  "app": {
    "windows": [{
      "url": "http://localhost:3010"
    }]
  }
}
```

動作:

- `npm run tauri:dev`実行時、`beforeDevCommand`で`npm run dev`が実行される
- Next.js開発サーバーがポート3010で起動
- Tauriウィンドウが`http://localhost:3010`を表示

配布環境

Tauriカスタムプロトコル（`tauri://localhost`）

- 静的HTMLファイルを直接配信
- Node.js不要
- アプリバンドルに含まれる
- オフライン動作可能

設定:

```
// tauri.conf.json
{
  "build": {
    "beforeBuildCommand": "npm run build",
    "frontendDist": "../out"
  },
  "app": {
    "windows": [{
      "url": "tauri://localhost"  // または "index.html"
    }]
  },
  "bundle": {
    "resources": ["../out", "template-data.json"]
  }
}
```

実装要件:

1. **Cargo.tomlの設定:**

   ```
   [features]
   custom-protocol = ["tauri/custom-protocol"]
   default = ["custom-protocol"]  # デフォルトで有効化
   ```

2. **ビルド時のフラグ:**

   - `tauri build`コマンドで`--features custom-protocol`を指定（またはCargo.tomlでデフォルト有効）
   - または、`Cargo.toml`の`[features]`で`default = ["custom-protocol"]`を設定

3. **動作確認:**

- 本番ビルド時に`tauri://localhost`でアクセスできるか
- 静的ファイルが正しく配信されるか
- 相対パス（`../out/index.html`など）が解決されるか

URL解決の優先順位:

1. `tauri.conf.dev.json`の`url`（開発環境のみ）
2. `tauri.conf.json`の`url`（本番環境）
3. `frontendDist`で指定されたディレクトリの`index.html`

⚠ Tauriカスタムプロトコルでの注意事項

相対パスとアセット参照の問題:

`tauri://localhost`配信では、Next.jsの静的出力時のパス解決に注意が必要です。

問題点:

- 相対パス参照（画像、CSS、JS）がNext.jsの構成に依存
- `basePath`が必要になるケースがある
- Next.jsの`assetPrefix`との関係が不明瞭になりがち
- 「本番だけ画像が出ない」問題が発生しやすい

推奨設定:

next.config.js:

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  output: 'export',
  distDir: 'out',
  images: {
    unoptimized: true,  // 必須: 画像最適化を無効化
  },
  assetPrefix: '',  // 必須: 空文字列に固定（相対パスを使用）
  trailingSlash: true,
  // basePath は設定しない（デフォルトのまま）
}
```

画像コンポーネントの使用:

```
// Next.js Imageコンポーネントは unoptimized:true を推奨
import Image from 'next/image';

<Image
  src="/images/logo.png"  // public/ からの相対パス
  alt="Logo"
  width={200}
  height={200}
```

```
  unoptimized={true}  // 必須
/>
```

public/ 内のパス:

- `public/`内のファイルは絶対パス（`/images/logo.png`）で参照
- 相対パス（`./images/logo.png`）は使用しない

CSS/JSの参照:

- Next.jsのビルド出力は自動的に正しいパスで解決される
- `_next/static/`配下のファイルは自動的に正しいパスで参照される

確認事項:

- ☐ `assetPrefix`が空文字列に設定されているか
- ☐ `images.unoptimized`が`true`に設定されているか
- ☐ `public/`内のファイルが絶対パスで参照されているか
- ☐ 本番ビルドで画像・CSS・JSが正しく読み込まれるか

フロントエンドのAPI接続設定

環境変数の伝播

命名規則:

- Rust側: `API_SERVER_PORT`（サーバー側の設定）
- フロントエンド側: `NEXT_PUBLIC_API_SERVER_PORT`（Next.jsのビルド時埋め込み）

理由:

- Next.jsでは`NEXT_PUBLIC_`プレフィックスがないとクライアント側で使用できない
- Rust側では通常の環境変数名を使用
- 両方を設定することで、開発環境と本番環境で一貫した動作を保証

設定ファイル:

```
# .env または local.env（開発環境）
API_SERVER_PORT=3011
NEXT_PUBLIC_API_SERVER_PORT=3011
CHROMADB_PORT=8000
```

本番環境:

- 環境変数ファイルは使用しない（ビルド時に埋め込まれる）
- または、アプリケーション起動時に環境変数から読み込む

コード実装

フロントエンド側（lib/apiClient.ts）：

```
/**
 * Rust APIサーバー用のクライアント
 * ポート番号は環境変数から読み込み、デフォルトは3011
 */

const API_SERVER_PORT = process.env.NEXT_PUBLIC_API_SERVER_PORT
  ? parseInt(process.env.NEXT_PUBLIC_API_SERVER_PORT, 10)
  : 3011; // デフォルト値

const API_BASE_URL = `http://127.0.0.1:${API_SERVER_PORT}`;

// APIリクエスト関数
async function apiRequest<T>(endpoint: string, options?: RequestInit):
Promise<T> {
  const url = `${API_BASE_URL}${endpoint}`;
  // ... 実装
}
```

バックエンド側（src-tauri/src/main.rs）：

```
// Rust APIサーバーを起動（ポート番号は環境変数から読み込み、デフォルトは3011）
let api_port = std::env::var("API_SERVER_PORT")
    .ok()
    .and_then(|s| s.parse::<u16>().ok())
    .unwrap_or(if cfg!(debug_assertions) { 3011 } else { 3011 });

let api_addr = SocketAddr::from(([127, 0, 0, 1], api_port));
```

注意事項:

- 開発環境と本番環境で同じポート（3011）を使用
- 環境変数が設定されていない場合のフォールバック動作を明確化
- フロントエンドとバックエンドでポート番号を一致させる

起動順序とタイミング制御

起動順序

```
1. Tauriアプリ起動
   ↓
2. Next.js開発サーバー起動（開発環境のみ、非同期）
   ↓
3. Rust APIサーバー起動（非同期、ヘルスチェック待機）
   ↓
4. ChromaDB Server起動（非同期、ヘルスチェック待機）
```

```
    ↓
5. Tauriウィンドウ表示
```

タイミング制御

Next.js開発サーバー:

- `beforeDevCommand`で自動起動
- Tauriは起動完了まで待機（`devUrl`が利用可能になるまで）

Rust APIサーバー:

- 非同期起動（`tauri::async_runtime::spawn`）
- フロントエンドはリトライロジックで待機
- **ポート番号をフロントエンドへ通知**（Tauriイベント）

ChromaDB Server:

- 非同期起動
- **APIサーバーはChromaDBの起動成功を待ってからエンドポイント登録**
- 起動失敗時はdegradeモードで起動（検索機能は最低限）

🔷 起動順序保証の実装

問題:

- フロント側のリトライに責務が寄っている
- ChromaDBが起動していない状態でAPIエンドポイントが登録される可能性

推奨実装:

Rust APIサーバー側:

```
// ChromaDBの起動を待ってからAPIエンドポイントを登録
async fn initialize_api_with_chromadb() {
    // 1. ChromaDB Serverを起動
    match chromadb::init_chromadb_server(chromadb_data_dir, port).await
{
        Ok(_) => {
            eprintln!("✅ ChromaDB Serverが起動しました");
            // 2. 成功したらAPIエンドポイントを登録
            register_chromadb_endpoints();
        },
        Err(e) => {
            eprintln!("⚠️  ChromaDB Serverの起動に失敗しました: {}", e);
            // 3. 失敗したらdegradeモードで起動
            register_degraded_endpoints();
            eprintln!("   SQLiteフォールバックモードで起動します");
        }
    }
```

```
    // 4. APIサーバーを起動
    start_api_server(api_addr).await;
}
```

フロントエンド側:

- ChromaDBの状態を確認するエンドポイント: `/api/chromadb/status`
- 状態に応じてUIを調整（検索機能の無効化など）

ヘルスチェック

Rust APIサーバー:

- エンドポイント: `/health`
- レスポンス: `{ "status": "ok" }`

ChromaDB Server:

- エンドポイント: `/api/v1/heartbeat`
- レスポンス: `{ "nanosecond heartbeat": <timestamp> }`

フロントエンドのリトライロジック

```
/**
 * APIサーバーの起動を待つ
 * @param maxRetries 最大リトライ回数（デフォルト: 30）
 * @param interval リトライ間隔（ミリ秒、デフォルト: 1000）
 */
async function waitForApiServer(
  maxRetries: number = 30,
  interval: number = 1000
): Promise<boolean> {
  const API_BASE_URL =
`http://127.0.0.1:${process.env.NEXT_PUBLIC_API_SERVER_PORT || 3011}`;

  for (let i = 0; i < maxRetries; i++) {
    try {
      const response = await fetch(`${API_BASE_URL}/health`);
      if (response.ok) {
        console.log(`✅ APIサーバーが起動しました（${i + 1}回目の試行）`);
        return true;
      }
    } catch (error) {
      // サーバーが起動していない
      if (i % 5 === 0) {
        console.log(`⏳ APIサーバーの起動を待機中... (${i +
1}/${maxRetries})`);
      }
    }
```

```
    await new Promise(resolve => setTimeout(resolve, interval));
  }

  throw new Error(`APIサーバーの起動を待てませんでした（${maxRetries}回試行）
`);
}

// 使用例
try {
  await waitForApiServer();
  // APIリクエストを実行
} catch (error) {
  console.error('❌ APIサーバーに接続できませんでした:', error);
  // エラー処理
}
```

ポート競合の回避策

🔥 ポート動的割り当ての推奨（最重要）

問題:

- 固定ポート（3011, 8000）がユーザー環境で使用できない可能性がある
- セキュリティソフトでポートがブロックされる
- 社内VPNや他のアプリケーションと衝突する
- 複数のインスタンスを同時起動できない

推奨解決策: ポート自動選択 → フロントエンドへ通知方式

アーキテクチャ:

```
起動フロー:
1. Rust APIサーバーが空いているポートを探索（3011から開始）
2. ポートを確保
3. Tauriウィンドウへ "このポートを使え" とメッセージ送信
4. フロントエンドはそのポートでAPIへ接続
```

実装:

Rust側（ポート自動割り当て）:

```
use std::net::{TcpListener, SocketAddr};

fn find_available_port(start_port: u16, max_attempts: u16) ->
Option<u16> {
    for port in start_port..start_port + max_attempts {
        let addr = SocketAddr::from(([127, 0, 0, 1], port));
        if TcpListener::bind(addr).is_ok() {
```

```
                return Some(port);
            }
        }
        None
    }

    // APIサーバーのポート自動割り当て
    let api_port = std::env::var("API_SERVER_PORT")
        .ok()
        .and_then(|s| s.parse::<u16>().ok())
        .unwrap_or_else(|| {
            find_available_port(3011, 10).unwrap_or(3011)
        });

    // フロントエンドへポート番号を通知
    app.emit("api-port-assigned", api_port).unwrap();
```

フロントエンド側（動的ポート受信）:

```
// Tauriイベントでポート番号を受信
import { listen } from '@tauri-apps/api/event';

let apiPort = 3011; // デフォルト値

listen<number>('api-port-assigned', (event) => {
  apiPort = event.payload;
  console.log(`✅ APIサーバーポートを受信: ${apiPort}`);
});

const API_BASE_URL = `http://127.0.0.1:${apiPort}`;
```

利点:

- ユーザー環境に依存しない
- 複数インスタンスの同時起動が可能
- セキュリティソフトとの競合を回避
- 環境変数での手動設定も可能（優先度: 環境変数 > 自動割り当て）

ChromaDB Serverのポート設定

問題:

- ポート8000が固定値で設定されている
- Pythonプロセスのクラッシュが検知しづらい
- Windowsでサイレントにプロセスが終了する可能性
- Python環境依存（DLL, venv）の破損リスク

解決策:

1. **ポート自動検出（推奨）**

```
let chromadb_port = std::env::var("CHROMADB_PORT")
    .ok()
    .and_then(|s| s.parse::<u16>().ok())
    .unwrap_or_else(|| {
        find_available_port(8000, 10).unwrap_or(8000)
    });
```

2. **環境変数による設定（フォールバック）**

```
let chromadb_port = std::env::var("CHROMADB_PORT")
    .ok()
    .and_then(|s| s.parse::<u16>().ok())
    .unwrap_or(8000);
```

3. **プロセス監視と自動再起動**

```
// ChromaDBプロセスの監視
tokio::spawn(async move {
    loop {
        tokio::time::sleep(Duration::from_secs(30)).await;
        if !is_chromadb_running(port).await {
            eprintln!("⚠  ChromaDB Serverが停止しています。再起動を試みま
す...");
            // 再起動ロジック
        }
    }
});
```

APIサーバーのポート設定

推奨実装（ポート自動割り当て）：

```
// 環境変数 > 自動検出 > デフォルト値の優先順位
let api_port = std::env::var("API_SERVER_PORT")
    .ok()
    .and_then(|s| s.parse::<u16>().ok())
    .unwrap_or_else(|| {
        find_available_port(3011, 10).unwrap_or(3011)
    });

// フロントエンドへポート番号を通知
app.emit("api-port-assigned", api_port).unwrap();
```

開発環境と本番環境の統一:

- 開発環境でもAPIサーバーは**3011**を使用（Next.jsは3010）
- 本番環境も**3011**を使用（統一）
- ポート競合時は自動的に次の利用可能なポートを選択

注意事項:

- `NEXT_PUBLIC_API_SERVER_PORT`はビルド時固定ではなく、起動時にRustがフロントへ通知する方式を推奨
- これにより「3011が使えない」問題が完全に解決される

環境変数の設計

開発環境用（`.env` または `local.env`）

```
# APIサーバーのポート
API_SERVER_PORT=3011
NEXT_PUBLIC_API_SERVER_PORT=3011

# ChromaDB Serverのポート
CHROMADB_PORT=8000

# Next.js開発サーバーのポート（package.jsonで設定）
# NEXT_PORT=3010  # コメントアウト（package.jsonで設定）
```

読み込み順序:

1. `local.env`（優先）
2. `.env`（フォールバック）
3. システム環境変数

実装（src-tauri/src/main.rs）:

```
#[cfg(debug_assertions)]
{
    // 環境変数ファイルの読み込み（local.envを優先、なければ.env）
    if let Err(_e) = dotenv::from_filename("local.env") {
        if dotenv::from_filename(".env").is_err() {
            eprintln!("⚠️  環境変数ファイル（local.env または .env）が見つかりません。環境変数から直接読み込みます。");
        }
    } else {
        eprintln!("✅ 環境変数ファイル（local.env）を読み込みました");
    }
}
```

本番環境用（環境変数または設定ファイル）

オプション1: 環境変数（推奨）

```
# アプリケーション起動時に環境変数から読み込む
API_SERVER_PORT=3011
CHROMADB_PORT=8000
```

オプション2: ビルド時埋め込み

- Next.jsのビルド時に`NEXT_PUBLIC_API_SERVER_PORT`が埋め込まれる
- Rust側は実行時に環境変数から読み込む

ポート範囲の推奨設定

推奨ポート範囲

用途	ポート範囲	説明
フロントエンド開発サーバー	3000-3099	Next.js、Viteなど
バックエンドAPIサーバー	3010-3019	Rust API、Node.js APIなど
ベクトル検索サーバー	8000-8099	ChromaDB、Qdrantなど
データベース	5432, 3306など	PostgreSQL、MySQLなど（本プロジェクトでは使用しない）

本プロジェクトのポート割り当て

サービス	開発環境	本番環境	備考
Next.js開発サーバー	3010	N/A	開発環境のみ
Rust APIサーバー	3011	3011	環境変数で変更可能
ChromaDB Server	8000	8000	環境変数で変更可能

接続フロー

開発環境

```
1. Tauriアプリ起動
   ↓
2. Next.js開発サーバー起動（ポート3010）
   ↓
3. Rust APIサーバー起動（ポート3011）
   ↓
```

```
4. ChromaDB Server起動（ポート8000）
   ↓
5. Tauriウィンドウが http://localhost:3010 を表示
   ↓
6. フロントエンドが http://localhost:3011 にAPIリクエスト
   ↓
7. Rust APIサーバーが http://localhost:8000 にChromaDBリクエスト
```

配布環境

```
1. Tauriアプリ起動
   ↓
2. Rust APIサーバー起動（ポート3011）
   ↓
3. ChromaDB Server起動（ポート8000）
   ↓
4. Tauriウィンドウが tauri://localhost を表示（静的HTML）
   ↓
5. フロントエンドが http://localhost:3011 にAPIリクエスト
   ↓
6. Rust APIサーバーが http://localhost:8000 にChromaDBリクエスト
```

ChromaDBフォールバック仕様

⚠ ChromaDBを「不安定プロセス」として扱う

問題点:

- Pythonプロセスのクラッシュ がRustから検知しづらい
- Windowsでサイレントにプロセスが終了する可能性
- Python環境依存（DLL, venv）の破損リスク
- 「RAGが使えないとアプリが完全に死ぬ」状態を避けたい

ChromaDBが起動しない場合の挙動

1. SQLiteの全文検索モードにフォールバック

```rust
// ChromaDB起動失敗時のフォールバック
match chromadb::init_chromadb_server(chromadb_data_dir, port).await {
    Ok(_) => {
        // ChromaDB使用モード
        use_chromadb_for_search = true;
    },
    Err(e) => {
        eprintln!("⚠  ChromaDB Serverの起動に失敗しました: {}", e);
        // SQLite全文検索モードにフォールバック
        use_chromadb_for_search = false;
```

```
        eprintln!("   SQLite全文検索モードで動作します");
    }
}
```

2. フロントエンドに通知

```
// TauriイベントでChromaDBの状態を受信
import { listen } from '@tauri-apps/api/event';

listen<{ available: boolean; reason?: string }>('chromadb-status',
(event) => {
  const { available, reason } = event.payload;
  if (!available) {
    console.warn('⚠  ChromaDBが利用できません:', reason);
    // UIを調整（検索機能の無効化など）
    disableVectorSearch();
  }
});
```

3. 次回起動時に自動再試行

```
// 起動時に前回の状態を確認
let last_chromadb_status = get_last_chromadb_status();
if !last_chromadb_status.success {
    eprintln!("   前回の起動でChromaDBが失敗しました。再試行します...");
    // 再試行ロジック
}
```

4. プロセス監視と自動再起動

```
// ChromaDBプロセスの監視タスク
tokio::spawn(async move {
    loop {
        tokio::time::sleep(Duration::from_secs(30)).await;

        if !is_chromadb_running(port).await {
            eprintln!("⚠  ChromaDB Serverが停止しています。再起動を試みま
す...");

            match
chromadb::init_chromadb_server(chromadb_data_dir.clone(), port).await {
                Ok(_) => {
                    eprintln!("✅ ChromaDB Serverを再起動しました");
                    app.emit("chromadb-status", json!({ "available":
true })).unwrap();
                },
```

```
                Err(e) => {
                    eprintln!("❌ ChromaDB Serverの再起動に失敗しました:
{}", e);
                    app.emit("chromadb-status", json!({
                        "available": false,
                        "reason": e
                    })).unwrap();
                }
            }
        }
    }
});
```

フォールバックモードの機能制限:

機能	ChromaDB使用時	SQLiteフォールバック時
ベクトル検索	✅ 高精度	❌ 利用不可
全文検索	✅ 利用可能	✅ 利用可能（FTS5）
セマンティック検索	✅ 利用可能	❌ 利用不可
エンティティ検索	✅ 高精度	⚠ 文字列マッチングのみ
関係検索	✅ 高精度	⚠ 文字列マッチングのみ

ユーザーへの通知:

```
// UIでの通知例
if (!chromadbAvailable) {
  showNotification({
    type: 'warning',
    title: 'ChromaDBが利用できません',
    message: 'ベクトル検索機能は制限されますが、アプリは正常に動作します。',
    actions: [
      { label: '再試行', onClick: () => retryChromaDB() },
      { label: '了解', onClick: () => {} }
    ]
  });
}
```

エラーハンドリングの詳細仕様

ポート競合エラー

Rust側のエラーハンドリング:

```
use std::net::TcpListener;
use std::io::{Error, ErrorKind};

match TcpListener::bind(addr) {
    Ok(listener) => {
        eprintln!("✅ ポート {} でAPIサーバーを起動しました", port);
        // 成功処理
    },
    Err(e) if e.kind() == ErrorKind::AddrInUse => {
        eprintln!("❌ ポート {} が既に使用されています", port);
        eprintln!("   対処法:");
        eprintln!("   1. 使用中のプロセスを確認: lsof -i :{}", port);
        eprintln!("   2. 環境変数でポートを変更: API_SERVER_PORT={}", port
+ 1);
        eprintln!("   3. 競合しているプロセスを終了");

        // ポート自動検出を試行
        if let Some(available_port) = find_available_port(port + 1, 10)
{
            eprintln!("   💡 代替ポート {} を使用します", available_port);
            // 代替ポートで再試行
        } else {
            return Err(format!("ポート競合: 利用可能なポートが見つかりませんでし
た"));
        }
    },
    Err(e) => {
        return Err(format!("ポートバインドエラー: {}", e));
    },
}
```

ChromaDB Server起動失敗:

```
match chromadb::init_chromadb_server(chromadb_data_dir, port).await {
    Ok(_) => {
        eprintln!("✅ ChromaDB Serverが正常に起動しました");
    },
    Err(e) => {
        eprintln!("⚠️  ChromaDB Serverの起動に失敗しました: {}", e);
        eprintln!("   対処法:");
        eprintln!("   1. Python環境を確認: python3 --version");
        eprintln!("   2. ChromaDBがインストールされているか確認: pip list |
grep chromadb");
        eprintln!("   3. ポート {} が使用中でないか確認: lsof -i :{}", port,
port);
        eprintln!("   4. 環境変数でポートを変更: CHROMADB_PORT={}", port +
1);
        eprintln!("   SQLiteフォールバックが使用されます");
        // フォールバック処理（SQLiteのみで動作）
```

```
    }
}
```

フロントエンドのエラーハンドリング:

```
async function apiRequest<T>(endpoint: string, options?: RequestInit):
Promise<T> {
  const url = `${API_BASE_URL}${endpoint}`;

  try {
    const response = await fetch(url, {
      ...options,
      headers: {
        'Content-Type': 'application/json',
        ...options?.headers,
      },
    });

    if (!response.ok) {
      const errorData: ApiError = await response.json().catch(() => ({
        error: `HTTP ${response.status}: ${response.statusText}`,
      }));
      throw new Error(errorData.error || `API request failed:
${response.statusText}`);
    }

    return response.json();
  } catch (error) {
    if (error instanceof TypeError && error.message.includes('fetch')) {
      // ネットワークエラー（サーバーが起動していない可能性）
      console.error('❌ APIサーバーに接続できませんでした');
      console.error('   対処法:');
      console.error('   1. アプリケーションを再起動してください');
      console.error('   2. ポート3011が使用中でないか確認してください');
      throw new Error('APIサーバーに接続できませんでした。アプリケーションを再起
動してください。');
    }
    throw error;
  }
}
```

セキュリティ考慮事項

権限モデルとセキュリティポリシー

MissionAIのセキュリティ特性:

MissionAIはすべてローカル内で完結し、外部ネットワークを必要としません。

重要なポイント:

- ローカルHTTPサーバーはアプリ間通信専用で、外部には公開されない
- すべての通信は`127.0.0.1`（localhost）に限定される
- 外部ネットワークへの接続は行わない（AI API呼び出しを除く）
- データはすべてローカルファイルシステムに保存される

企業環境での審査対応:

金融機関、医療機関、企業セキュリティポリシーでは、ローカルAPIソケットを立ち上げるアプリは監査に引っかかりやすい場合があります。

説明資料として使用可能な内容:

```
MissionAIのセキュリティ特性:

1. ローカル完結型アーキテクチャ
   - すべてのデータ処理はローカルマシン内で完結
   - 外部サーバーへのデータ送信は行わない

2. ローカルHTTPサーバーの用途
   - TauriアプリケーションとNext.jsフロントエンド間の通信専用
   - 127.0.0.1（localhost）のみでリッスン
   - 外部ネットワークからはアクセス不可能

3. データの保存場所
   - SQLite: ~/Library/Application Support/com.missionai.app/.../app.db
   - ChromaDB: ~/Library/Application
Support/com.missionai.app/.../chromadb/
   - すべてローカルファイルシステム内

4. ネットワーク通信
   - AI API呼び出し（OpenAI、Anthropic、Ollama）のみ外部通信
   - ユーザーが明示的に設定したAPIキーを使用
   - アプリケーション自体は外部サーバーに接続しない
```

CORS設定

開発環境:

```
// src-tauri/src/api/server.rs
let cors = CorsLayer::new()
    .allow_origin("http://localhost:3010".parse::<HeaderValue>
().unwrap())
    .allow_origin("http://127.0.0.1:3010".parse::<HeaderValue>
().unwrap())
    .allow_methods([Method::GET, Method::POST, Method::PUT,
Method::DELETE])
```

```
    .allow_headers(Any)
    .allow_credentials(true);
```

- Next.js開発サーバー（ポート3010）からのリクエストのみ許可
- 開発の利便性のため、特定のポートのみ許可

本番環境:

```
let cors = CorsLayer::new()
    .allow_origin("tauri://localhost".parse::<HeaderValue>().unwrap())
    .allow_methods([Method::GET, Method::POST, Method::PUT,
Method::DELETE])
    .allow_headers(Any)
    .allow_credentials(true);
```

- Tauriカスタムプロトコルからのリクエストのみ許可
- ローカルホストのみ許可

CSP（Content Security Policy）

開発環境:

```
"csp": "default-src 'self' http://localhost:3010; connect-src 'self'
http://localhost:3010 http://localhost:3011 http://127.0.0.1:3010
http://127.0.0.1:3011 ws://localhost:* ws://127.0.0.1:*
https://api.openai.com https://api.anthropic.com https://*.ollama.ai;
script-src 'self' 'unsafe-inline' 'unsafe-eval' http://localhost:3010;
style-src 'self' 'unsafe-inline' http://localhost:3010; img-src 'self'
data: https: file: http://localhost:3010; font-src 'self' data:
http://localhost:3010;"
```

本番環境:

```
"csp": "default-src 'self' tauri://localhost; connect-src 'self'
tauri://localhost http://localhost:3011 http://127.0.0.1:3011
ws://localhost:* ws://127.0.0.1:* https://api.openai.com
https://api.anthropic.com https://*.ollama.ai; script-src 'self'
'unsafe-inline' 'unsafe-eval'; style-src 'self' 'unsafe-inline'; img-src
'self' data: https: file: tauri://localhost; font-src 'self' data:
tauri://localhost;"
```

各ディレクティブの説明:

- `default-src`: デフォルトのソース
- `connect-src`: 接続可能なURL（API、WebSocketなど）

- `script-src`: スクリプトのソース
- `style-src`: スタイルシートのソース
- `img-src`: 画像のソース
- `font-src`: フォントのソース

ログ出力の仕様

ログレベル

開発環境:

- DEBUG: 詳細なデバッグ情報
- INFO: 通常の動作情報
- WARN: 警告（動作は継続）
- ERROR: エラー（動作に影響）

本番環境:

- INFO: 通常の動作情報
- WARN: 警告
- ERROR: エラー

ログ出力の形式

起動時:

```
🚀 Tauriアプリケーションを起動中...
✅ データベース初期化完了: /path/to/app.db
🔧 APIサーバーポート: 3011 (環境変数: 3011)
✅ APIサーバーが起動しました: http://127.0.0.1:3011
🔧 ChromaDB Serverの起動を開始します...
   データディレクトリ: /path/to/chromadb
   ポート: 8000
✅ ChromaDB Serverが正常に起動しました
```

エラー時:

```
❌ ポート 3011 が既に使用されています
   対処法:
   1. 使用中のプロセスを確認: lsof -i :3011
   2. 環境変数でポートを変更: API_SERVER_PORT=3012
   3. 競合しているプロセスを終了
```

ログ出力の実装

開発環境:

```
#[cfg(debug_assertions)]
{
    eprintln!("🔧 APIサーバーポート: {} (環境変数: {})", port, env_var);
    eprintln!("✅ APIサーバーが起動しました: http://{}", addr);
}
```

本番環境（構造化ログ推奨）：

```
use tracing_subscriber::fmt;

// 構造化ログの初期化
tracing_subscriber::fmt()
    .with_target(true)
    .json()  // JSON形式で出力
    .with_max_level(tracing::Level::INFO)
    .init();

// 構造化ログの使用例
tracing::info!(
    api_port = port,
    chromadb_port = chromadb_port,
    status = "started",
    "APIサーバーとChromaDB Serverが起動しました"
);

// エラー時の構造化ログ
tracing::error!(
    error = %e,
    port = port,
    action = "port_bind_failed",
    "ポートのバインドに失敗しました"
);
```

構造化ログの利点:

- エラー原因、時刻、ポート番号、起動フローのどこで失敗したかが機械的に解析可能
- サポート・デバッグのコストが激減
- ログ解析ツール（ELK、Lokiなど）との統合が容易

ログ出力例（JSON形式）：

```
{
  "timestamp": "2025-01-15T10:30:45.123Z",
  "level": "INFO",
  "fields": {
    "api_port": 3011,
    "chromadb_port": 8000,
```

```
    "status": "started"
  },
  "target": "mission_ai::api::server",
  "message": "APIサーバーとChromaDB Serverが起動しました"
}
```

設定ファイルの完全な仕様

tauri.conf.json（本番環境）

```
{
  "productName": "MissionAI",
  "version": "2.1.2",
  "identifier": "com.missionai.app",
  "build": {
    "beforeBuildCommand": "npm run build",
    "frontendDist": "../out"
  },
  "app": {
    "withGlobalTauri": true,
    "windows": [
      {
        "title": "MissionAI",
        "width": 1400,
        "height": 900,
        "resizable": true,
        "fullscreen": false,
        "devtools": false,
        "url": "tauri://localhost"
      }
    ],
    "security": {
      "csp": "default-src 'self' tauri://localhost; connect-src 'self'
tauri://localhost http://localhost:3011 http://127.0.0.1:3011
ws://localhost:* ws://127.0.0.1:* https://api.openai.com
https://api.anthropic.com https://*.ollama.ai; script-src 'self'
'unsafe-inline' 'unsafe-eval'; style-src 'self' 'unsafe-inline'; img-src
'self' data: https: file: tauri://localhost; font-src 'self' data:
tauri://localhost;"
    }
  },
  "bundle": {
    "active": true,
    "targets": "all",
    "resources": [
      "../out",
      "template-data.json"
    ]
  }
}
```

tauri.conf.dev.json（開発環境）

```
{
  "build": {
    "beforeDevCommand": "npm run dev",
    "devUrl": "http://localhost:3010"
  },
  "app": {
    "windows": [
      {
        "title": "MissionAI",
        "width": 1400,
        "height": 900,
        "resizable": true,
        "fullscreen": false,
        "devtools": true,
        "url": "http://localhost:3010"
      }
    ],
    "security": {
      "csp": "default-src 'self' http://localhost:3010; connect-src
'self' http://localhost:* http://127.0.0.1:* ws://localhost:*
ws://127.0.0.1:* https://api.openai.com https://api.anthropic.com
https://*.ollama.ai; script-src 'self' 'unsafe-inline' 'unsafe-eval'
http://localhost:3010; style-src 'self' 'unsafe-inline'
http://localhost:3010; img-src 'self' data: https: file:
http://localhost:3010; font-src 'self' data: http://localhost:3010;"
    }
  },
  "bundle": {
    "resources": []
  }
}
```

package.json

```
{
  "name": "mission-ai",
  "version": "2.1.2",
  "scripts": {
    "dev": "next dev -p 3010",
    "build": "next build",
    "start": "next start -p 3010",
    "lint": "next lint",
    "tauri": "tauri",
    "tauri:dev": "tauri dev",
    "tauri:build": "tauri build"
```

```
  }
}
```

.env / local.env

```
# APIサーバーのポート
API_SERVER_PORT=3011
NEXT_PUBLIC_API_SERVER_PORT=3011

# ChromaDB Serverのポート
CHROMADB_PORT=8000
```

Cargo.toml

```
[features]
custom-protocol = ["tauri/custom-protocol"]
default = ["custom-protocol"]  # デフォルトで有効化
```

実装チェックリスト

開発環境

- [ ] Next.js開発サーバーがポート3010で起動
- [ ] Rust APIサーバーがポート3011で起動（環境変数対応）
- [ ] ChromaDB Serverがポート8000で起動（環境変数対応）
- [ ] Tauriウィンドウが`http://localhost:3010`を表示
- [ ] フロントエンドが`http://localhost:3011`にAPIリクエスト
- [ ] ポート競合が発生しない
- [ ] 環境変数ファイル（`.env`または`local.env`）が読み込まれる
- [ ] フロントエンドのリトライロジックが動作する

配布環境

- [ ] 静的HTMLが`tauri://localhost`で配信される
- [ ] Rust APIサーバーがポート3011で起動
- [ ] ChromaDB Serverがポート8000で起動
- [ ] Node.jsが不要である
- [ ] オフライン動作が可能
- [ ] `custom-protocol`機能が有効になっている
- [ ] 静的ファイルがバンドルに含まれている

テスト方法

開発環境のテスト

1. ポート競合のテスト:

```
# ポート3011を占有
nc -l 3011 &

# アプリを起動（エラーメッセージを確認）
npm run tauri:dev

# ポートを解放
killall nc
```

2. APIサーバーの接続テスト:

```
# アプリ起動後
curl http://localhost:3011/health
# 期待されるレスポンス: {"status":"ok"}
```

3. ChromaDB Serverの接続テスト:

```
curl http://localhost:8000/api/v1/heartbeat
# 期待されるレスポンス: {"nanosecond heartbeat": <timestamp>}
```

4. フロントエンドのAPI接続テスト:

```
# ブラウザの開発者ツールで確認
# Networkタブで http://localhost:3011 へのリクエストを確認
```

本番環境のテスト

1. 静的HTMLの配信テスト:

```
npm run build
npm run tauri:build
# ビルドされたアプリを起動して、tauri://localhostが動作するか確認
```

2. ポート設定のテスト:

```
# 環境変数を変更してビルド
API_SERVER_PORT=3012 npm run tauri:build
# アプリ起動後、ポート3012でAPIサーバーが起動するか確認
curl http://localhost:3012/health
```

3. オフライン動作のテスト:

```
# インターネット接続を切断
# アプリが正常に動作するか確認（APIサーバーとChromaDBはローカルで動作）
```

トラブルシューティング

ポート競合エラー

症状:

- `Address already in use`エラー
- サーバーが起動しない

対処法:

1. 使用中のポートを確認: `lsof -i :PORT`
2. 環境変数でポートを変更
3. 競合しているプロセスを終了

ChromaDB Serverが起動しない

症状:

- ChromaDB Serverの初期化に失敗
- ポート8000が使用中

対処法:

1. ポート8000の使用状況を確認: `lsof -i :8000`
2. `CHROMADB_PORT`環境変数でポートを変更
3. Python環境を確認: `python3 --version`
4. ChromaDBがインストールされているか確認: `pip list | grep chromadb`

APIサーバーに接続できない

症状:

- フロントエンドからAPIサーバーに接続できない
- 404エラーまたは接続エラー

対処法:

1. APIサーバーが起動しているか確認: `curl http://localhost:3011/health`
2. ポート番号が正しいか確認（環境変数`NEXT_PUBLIC_API_SERVER_PORT`）
3. CORS設定を確認
4. アプリケーションを再起動

静的HTMLが表示されない（本番環境）

症状:

- 本番ビルドで`tauri://localhost`が表示されない
- 404エラーが表示される

対処法:

1. `custom-protocol`機能が有効になっているか確認（`Cargo.toml`）
2. `tauri.conf.json`の`url`設定を確認
3. `frontendDist`が正しく設定されているか確認
4. `bundle.resources`に`"../out"`が含まれているか確認

まとめ

推奨される最終設計

環境	Next.js	Rust API	ChromaDB	HTML配信
開発	3010	3011	8000	Next.js (3010)
本番	N/A	3011	8000	Tauri Protocol

重要なポイント:

1. ✅ **ポート動的割り当てを推奨**（起動時にRustがフロントへ通知）
2. ✅ **ChromaDBフォールバック仕様を明文化**（SQLite全文検索へのフォールバック）
3. ✅ **Tauriカスタムプロトコルでの注意事項**（assetPrefix、相対パス）
4. ✅ **起動順序保証の実装**（ChromaDB起動成功後にAPIエンドポイント登録）
5. ✅ **構造化ログの採用**（JSON形式、デバッグコスト削減）
6. ✅ **セキュリティ権限モデルの記述**（企業環境での審査対応）
7. 開発環境と本番環境でポートを統一（API: 3011、ChromaDB: 8000）
8. 環境変数でポートを変更可能にする（優先度: 環境変数 > 自動割り当て）
9. 本番環境ではNode.js不要（静的HTML + Rust API）
10. エラーハンドリングとログ出力を適切に実装する

改善の優先順位

🔥 最重要（即座に実装推奨）

1. **ポート動的割り当て → フロントエンド通知方式**
    - ユーザー環境依存のトラブルの8割を解決
2. **ChromaDBフォールバック仕様の実装**
    - 「RAGが使えないとアプリが完全に死ぬ」状態を回避
3. **TauriカスタムプロトコルでのNext.js制約の対応**

- 「本番だけ画像・CSSが死ぬ」事故を防止

⚡ 高優先度（次回リリースまでに実装）

4. **起動順序保証の実装**
    - ChromaDB起動成功後にAPIエンドポイント登録
5. **構造化ログの採用**
    - サポート・デバッグコストの削減
6. **セキュリティ権限モデルの文書化**
    - 企業環境での審査対応