React/Next.js設定ドキュメント

> 📋 **ステータス**: アクティブ（設定仕様書）
> 📅 **最終更新**: 2025-01-15
> 👤 **用途**: React/Next.jsフロントエンドの設定、ビルド、実行環境の詳細

概要

このプロジェクトでは、**Next.js 14**と**React 18**を使用してフロントエンドを構築しています。Tauriアプリケーションのフロントエンドとして、静的エクスポート方式で動作します。

技術スタック

コアライブラリ

- **Next.js**: `^14.0.0` - Reactフレームワーク
- **React**: `^18.2.0` - UIライブラリ
- **React DOM**: `^18.2.0` - DOMレンダリング
- **TypeScript**: `^5.0.0` - 型安全性

データフェッチング

- **@tanstack/react-query**: `^5.0.0` - サーバー状態管理とキャッシング

開発ツール

- **ESLint**: `^8.0.0` - コード品質チェック
- **eslint-config-next**: `^14.0.0` - Next.js用ESLint設定

Next.js設定

`next.config.js`

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  output: 'export',         // 静的エクスポート
  distDir: 'out',           // 出力ディレクトリ
  images: {
    unoptimized: true,      // 画像最適化を無効化（Tauri用）
  },
  trailingSlash: true,      // URL末尾にスラッシュを追加
  // クエリパラメータ方式のルーティングを使用するため、動的ルーティングは無効化
  // すべてのページは静的エクスポート可能
}

module.exports = nextConfig
```

設定の説明

`output: 'export'`

- **静的エクスポート**: すべてのページを静的HTMLとして出力
- **利点**: Node.jsサーバー不要、Tauriアプリに統合可能
- **制限**: サーバーサイド機能（API Routes、SSR）は使用不可

`distDir: 'out'`

- **出力ディレクトリ**: ビルド結果を`out/`ディレクトリに出力
- **Tauri統合**: `tauri.conf.json`の`frontendDist`で指定

`images.unoptimized: true`

- **画像最適化無効化**: Next.jsの画像最適化機能を無効化
- **理由**: Tauriのカスタムプロトコル（`tauri://localhost`）では画像最適化が動作しない
- **代替**: 通常の`<img>`タグまたは`next/image`の`unoptimized`プロップを使用

`trailingSlash: true`

- **末尾スラッシュ**: URLの末尾にスラッシュを追加
- **利点**: 静的エクスポート時のパス解決の一貫性

TypeScript設定

`tsconfig.json`

```
{
  "compilerOptions": {
    "target": "ES2020",
    "lib": ["dom", "dom.iterable", "esnext"],
    "allowJs": true,
    "skipLibCheck": true,
    "strict": true,
    "noEmit": true,
    "esModuleInterop": true,
    "module": "esnext",
    "moduleResolution": "bundler",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "jsx": "preserve",
    "incremental": true,
    "plugins": [
      {
        "name": "next"
      }
    ],
    "paths": {
```

```
      "@/*": ["./*"]
    }
  },
  "include": [
    "next-env.d.ts",
    "**/*.ts",
    "**/*.tsx",
    ".next/types/**/*.ts",
    "out/types/**/*.ts"
  ],
  "exclude": ["node_modules"]
}
```

重要な設定

`target: "ES2020"`

- **ターゲット**: ES2020の機能を使用可能
- **互換性**: モダンブラウザとTauri環境に対応

`strict: true`

- **厳格モード**: TypeScriptの厳格な型チェックを有効化
- **利点**: 型安全性の向上、バグの早期発見

`paths: { "@/*": ["./*"] }`

- **パスエイリアス**: `@/`でプロジェクトルートを参照
- **例**: `import Component from '@/components/Component'`

`jsx: "preserve"`

- **JSX処理**: Next.jsがJSXを処理するため、TypeScriptは変換しない
- **利点**: Next.jsの最適化を活用

ポート設定

開発環境

Next.js開発サーバー: ポート`3010`

- **設定**: `package.json`の`dev`スクリプト
- **コマンド**: `npm run dev` → `next dev -p 3010`
- **理由**: Rust APIサーバー（3011）とポートを分離

Rust APIサーバー: ポート`3011`

- **設定**: 環境変数`API_SERVER_PORT`またはデフォルト値
- **フロントエンド接続**: `lib/apiClient.ts`で設定

本番環境

静的エクスポート: `out/`ディレクトリ

- **配信**: Tauriカスタムプロトコル（`tauri://localhost`）
- **Node.js不要**: 静的ファイルのみ

環境変数

フロントエンド用環境変数

命名規則: `NEXT_PUBLIC_`プレフィックスが必要

```
# .env.local（開発環境）
NEXT_PUBLIC_API_SERVER_PORT=3011
```

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
- ビルド時に埋め込まれる

APIクライアント設定

`lib/apiClient.ts`:

```
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}`;
```

動作:

1. 環境変数`NEXT_PUBLIC_API_SERVER_PORT`を読み込み
2. 未設定の場合はデフォルト値`3011`を使用
3. Rust APIサーバーに接続

React Query設定

`components/QueryProvider.tsx`

```
'use client';

import { QueryClient, QueryClientProvider } from '@tanstack/react-
query';
import { useState } from 'react';
```

```
export default function QueryProvider({ children }: { children:
React.ReactNode }) {
  const [queryClient] = useState(
    () =>
      new QueryClient({
        defaultOptions: {
          queries: {
            staleTime: 60 * 1000,               // 1分間はキャッシュを有効
            gcTime: 5 * 60 * 1000,               // 5分間キャッシュを保持（旧
cacheTime）
            refetchOnWindowFocus: false,         // ウィンドウフォーカス時の
自動再取得を無効化
            refetchOnReconnect: false,           // 再接続時の自動再取得を無
効化
            retry: 1,                            // リトライ回数を1回に制限
          },
        },
      })
  );

  return <QueryClientProvider client={queryClient}>{children}
</QueryClientProvider>;
}
```

設定の説明

`staleTime: 60 * 1000` **(1分)**

- **キャッシュ有効期間**: 1分間はデータを新鮮とみなす
- **利点**: 不要な再取得を防止、パフォーマンス向上

`gcTime: 5 * 60 * 1000` **(5分)**

- **ガベージコレクション時間**: 5分間キャッシュを保持
- **旧名**: `cacheTime`（v5で`gcTime`に変更）

`refetchOnWindowFocus: false`

- **フォーカス時再取得無効化**: ウィンドウフォーカス時の自動再取得を無効化
- **理由**: デスクトップアプリでは不要な再取得を防止

`refetchOnReconnect: false`

- **再接続時再取得無効化**: ネットワーク再接続時の自動再取得を無効化
- **理由**: ローカルAPIサーバーでは不要

`retry: 1`

- **リトライ回数**: 失敗時に1回だけリトライ
- **理由**: ローカルAPIサーバーでは即座に失敗を検知

エラーハンドリング

Error Boundary

components/ErrorBoundary.tsx:

- **役割**: Reactコンポーネントツリーで発生したエラーをキャッチ
- **機能**:
  - エラー画面の表示
  - エラー情報の表示
  - リロード機能
  - エラーレセット機能

使用箇所: app/layout.tsxでアプリケーション全体をラップ

```
<ErrorBoundary>
  <QueryProvider>{children}</QueryProvider>
</ErrorBoundary>
```

フォント設定

Google Fonts

app/layout.tsx:

```
import { Inter, Noto_Sans_JP } from 'next/font/google';

const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
  variable: '--font-inter',
});

const notoSansJP = Noto_Sans_JP({
  weight: ['300', '400', '500', '600', '700'],
  subsets: ['latin'],  // 注意: 実装では'latin'を使用（日本語フォントだが、
Next.jsのフォント最適化により動作）
  display: 'swap',
  variable: '--font-noto',
  preload: true,
  adjustFontFallback: true,
});
```

フォントの説明

Inter

- **用途**: ラテン文字用のフォント
- **CSS変数**: `--font-inter`

Noto Sans JP

- **用途**: 日本語用のフォント
- **ウェイト**: 300, 400, 500, 600, 700
- **CSS変数**: `--font-noto`
- **preload**: `true` - フォントのプリロードを有効化
- **adjustFontFallback**: `true` - フォールバックフォントの調整

ルーティング

App Router（Next.js 14）

ディレクトリ構造:

```
app/
  ├── layout.tsx          # ルートレイアウト
  ├── page.tsx            # ホームページ
  ├── organization/       # 組織ページ
  │   ├── page.tsx
  │   └── detail/
  │       └── page.tsx
  ├── companies/          # 事業会社ページ
  ├── knowledge-graph/   # ナレッジグラフ
  ├── rag-search/         # RAG検索
  └── ...
```

クエリパラメータ方式

動的ルーティングの代わりにクエリパラメータを使用:

```
// 動的ルーティング: /companies/detail/[id]
// クエリパラメータ: /companies/detail?id=123

// 使用例
<Link href={`/companies/detail?id=${companyId}`}>
  {companyName}
</Link>
```

理由:

- 静的エクスポートが可能
- Node.jsサーバー不要

- Tauriアプリに統合しやすい

Tauri統合

開発環境

tauri.conf.dev.json:

```
{
  "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": "... http://localhost:3010 http://localhost:3011 ..."
    }
  },
  "bundle": {
    "resources": []
  }
}
```

動作:

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

本番環境

tauri.conf.json:

```
{
  "build": {
    "beforeBuildCommand": "npm run build",
    "frontendDist": "../out"
  },
  "app": {
    "withGlobalTauri": true,
```

```
    "windows": [{
      "url": "http://localhost:3010"  // 開発環境用（本番では使用されない）
    }],
    "security": {
      "csp": "... tauri://localhost ..."  // 本番環境ではtauri://localhost
プロトコルを使用
    }
  },
  "bundle": {
    "resources": ["../out"]
  }
}
```

動作:

1. `npm run tauri:build`実行時、`beforeBuildCommand`で`npm run build`が実行される
2. Next.jsが静的ファイルを`out/`ディレクトリに出力
3. Tauriが`out/`ディレクトリをバンドルに含める（`resources`に指定）
4. アプリ起動時、`tauri://localhost`プロトコルで静的ファイルを配信（CSPで指定）
5. **注意**: `url`フィールドは開発環境でのみ使用され、本番環境では`frontendDist`で指定されたディレクトリから静的ファイルが配信されます

ビルドと実行

開発環境

```
# Next.js開発サーバーのみ起動
npm run dev

# Tauri開発環境（Next.js + Tauri）
npm run tauri:dev
```

本番ビルド

```
# Next.jsの静的エクスポートのみ
npm run build

# Tauriアプリのビルド（Next.jsビルド + Tauriバンドル）
npm run tauri:build
```

出力ディレクトリ

- **開発**: `.next/`（Next.jsの内部ディレクトリ）
- **本番**: `out/`（静的エクスポート）

パフォーマンス最適化

静的エクスポート

- **利点**: ビルド時にすべてのページを事前生成
- **結果**: 初回読み込みが高速

画像最適化

- **制限**: `unoptimized: true`によりNext.jsの画像最適化は無効
- **推奨**: 画像は事前に最適化してから`public/`に配置

コード分割

- **自動**: Next.jsが自動的にコード分割を実行
- **結果**: 必要なJavaScriptのみ読み込まれる

フォント最適化

- **preload**: Google Fontsのプリロードを有効化
- **display: swap**: フォント読み込み中のフォールバック表示

トラブルシューティング

画像が表示されない

原因: `tauri://localhost`プロトコルでのパス解決の問題

解決方法:

1. `next.config.js`で`images.unoptimized: true`を確認
2. `public/`内のファイルは絶対パス（`/images/logo.png`）で参照
3. `next/image`コンポーネントを使用する場合は`unoptimized={true}`を設定

API接続エラー

原因: ポート番号の不一致

解決方法:

1. 環境変数`NEXT_PUBLIC_API_SERVER_PORT`を確認
2. Rust APIサーバーがポート3011で起動しているか確認
3. `lib/apiClient.ts`のデフォルト値を確認

ビルドエラー

原因: TypeScriptの型エラー、ESLintエラー

解決方法:

1. `npm run lint`でESLintエラーを確認
2. TypeScriptの型エラーを修正
3. `tsconfig.json`の設定を確認

静的エクスポート時のエラー

原因: サーバーサイド機能の使用

解決方法:

1. API Routesを使用していないか確認
2. `getServerSideProps`を使用していないか確認
3. すべてのページが静的エクスポート可能か確認

関連ドキュメント

- ポート設計とサーバー構成
- 開発ガイドライン
- データベース設計

最終更新: 2025-01-15