ChatGPT

skill-sheet-maker バックエンド移行計画（Rust+Tauri → Edge対応TypeScript）

概要: 現在 Rust+Tauri で構築されているデスクトップアプリケーションを、Vercel 上で動作する完全なTypeScript ベースのウェブアプリケーションに移行します。Next.js（App Router）を採用し、フロントエンドとバックエンドを統合してVercelにデプロイします。ORMには Prisma を使用し、Edge Runtime を活用できる部分はEdge Function化してグローバルに高速配信します。以下に、バックエンド構成変更の手順、具体的な作業一覧、技術選定の理由、最終的な理想構成について詳述します。

1. 新バックエンド構成と移行戦略

Next.js + TypeScriptへの全面移行: Vercel は Node.js/Edge 上で動作するNext.jsアプリを公式サポートしているため、Rust/Tauri製バックエンドを廃止し、Next.js 13+ (App Router) のフレームワーク上にバックエンド機能を実装します [1] [2]。Rust製のTauriバックエンドはVercelではそのまま動かせないため、まず**サーバレス/Edge環境で動作するTypeScriptコード**に置き換えることが最優先です。Next.js App RouterとEdge Runtimeを用いることで、フロントエンドとAPIバックエンドを同一プロジェクト内に統合し、デプロイを容易にします。

既存機能のサービス化: 現行アプリの各種コマンド（ユーザCRUDやスキル情報取得など）は、Next.jsのAPI RouteもしくはtRPCルーターとして実装し直します。現在Rust/Tauri側で定義されている**多数のコマンド**（ユーザ、スキル、資格、MBTI、経歴管理など）を確認し、それぞれTypeScriptで同等の処理を実装します [3] [4]。Next.js App Routerでは `app/api` 以下に `route.ts` ファイルを配置してAPIエンドポイントを作るか、tRPCを組み込んでエンドポイントレスなRPCを構築できます。どちらの場合も、**Prisma ORM**を使ってPostgreSQLにアクセスし、従来のRust(sqlx)クエリに相当する処理を行います。Vercel上でもPrismaはサーバレス・エッジ対応のPostgresソリューション（Prisma Data PlatformやNeon等）により動作可能です [5]。

Edge Runtimeの活用: Next.jsはデフォルトでNode.jsランタイムで動作しますが、Edge Functionsとして動かすことでグローバル展開・低遅延化が可能です。App Routerの `route.ts` ごとに `export const runtime = 'edge'` と指定すれば、Edge Runtime上で関数を実行できます。ただし**Edge Runtimeには制約**があり、Node.jsの標準モジュールが使えないため注意が必要です [6] [7]。例えば、データベースドライバの `pg` やNode版CryptoはEdgeでは動作せず、Next.js Edge RuntimeではNode版cryptoに依存するパッケージ（`pg` 等）は利用不可です [8]。そのため、**Prisma + Vercel Postgres**のようなエッジ対応の手段を取るか、一部の重い処理はNode.js Runtime（サーバレス関数）で動かす構成を検討します。PrismaはVercelと提携した「Prisma Postgres」を提供しており、**サーバレス・エッジ特化のPostgres**としてコネクションプーリングやグローバルキャッシュに対応しています [5]。これを使えばEdge Functionから直接高速にDBアクセスが可能で、Cold Startも発生しないため、次世代の構成に適しています。

フロントエンド統合: フロントエンドは既存のReact+TypeScriptコードをNext.jsプロジェクトに組み込みます。Next.jsのApp Routerではページコンポーネントを `app/` 以下に配置し、レイアウトやルーティングを定義します。既存のReactコンポーネントやVite設定を、Next.jsに合わせて調整します（例えば、Routingを React RouterからNext.jsのルーティングに置き換えるなど）。CSSフレームワーク（tailwindcssやdaisyUI）はNext.jsでもそのまま利用可能で、グローバルCSSや `tailwind.config.js` を設定します。

2. フェーズ別移行作業リスト

フェーズ0: 現行機能の棚卸し
- Rust+Tauri側で実装されている全APIコマンド・DB機能をリストアップする（ユーザ管理、スキル・資格マスタ、MBTI、キャリア経歴管理など）[3] [4]。
- フロントエンド（React側）でTauri経由で呼び出している機能（例：`invoke('create_user_cmd')`等[9]）を洗い出す。これらがどのデータをやり取りし、UIフローがどうなっているか整理する。

フェーズ1: Next.jsプロジェクトのセットアップ
- 新しいNext.js 13プロジェクトを作成する（App Router有効）。`create-next-app`でプロジェクト雛形を生成し、既存リポジトリに統合するか、現在のリポジトリに`next`ブランチを作成して構築を開始します。
- TypeScript設定（`tsconfig.json`）やlint設定、Prettier設定を既存プロジェクトから移行し、Next.js環境でも整合性が取れるよう調整します。
- Tailwind CSSとdaisyUIをNext.jsに組み込みます。Next.jsでは`globals.css`にTailwindのベースをインポートし、`postcss.config.js`と`tailwind.config.js`を用意して、既存のデザインを再現できるようにします。
- **Bulletproof-React流の構成**を参考に、ディレクトリを整理します（詳細は後述のディレクトリ構成参照）。`src/`ディレクトリを作成し、その配下に`app/`や`features/`を置いて、フロントとバックのコードを整理します[10] [11]。Next.jsはデフォルトで`/app`や`/pages`をルート直下に置きますが、Bulletproof-Reactではそれらを`src/`以下にまとめる手法が推奨されています[10]。これに従い、Next.jsの構成も`src`配下にまとめます。

フェーズ2: Prisma 導入とデータベース移行
- Prisma ORMをプロジェクトに追加します（`npm install prisma @prisma/client`）。既存のPostgreSQLスキーマをPrismaの`schema.prisma`で定義し直します。手動でユーザやスキル等のモデルを記述するか、既存DBから`prisma db pull`でスキーマをインポートします。
- `.env`にデータベース接続文字列`DATABASE_URL`を設定します（Vercel上では環境変数に設定予定）。VercelのPostgresアドオン（Prisma PostgresやNeon）を用いる場合、その提供する接続URLを利用します。
- Prismaのマイグレーションツールで既存DBに差分がないか確認します。現在Rust側で`sqlx migrate`管理されているマイグレーション履歴がありますが、Prismaでも同様のマイグレーションを適用し、DB構造を再現します。必要に応じてテスト環境用と本番環境用の2つのDB（README記載のusers_test, users_prod）もVercel環境で用意するか、接続先を分けて管理できるようにします[12] [13]。
- Prisma Clientを生成します（`npx prisma generate`）。Next.jsのAPIやtRPCルータからこのPrisma Clientをインポートしてdb操作を行います。Prismaはサーバレス/エッジ環境でも動作しますが、Vercel Edge上で直接Prismaを使う場合**Data Proxy**経由になる可能性があります[14]（Prisma自体はEdge Functions上でそのままは動かないため、PrismaのData Proxyサービスや Prisma Postgresを利用することでEdge対応します）。もしEdge対応が複雑であれば、**Next.jsのAPI RouteはNode.jsランタイムで動作させる**設定にして、通常のPrisma接続（直DB接続）を用いる手もあります。

フェーズ3: バックエンドAPI機能の実装（Node/Edge）
- フェーズ0で洗い出した各種機能を、Next.jsのAPIエンドポイントまたはtRPCルータとして一つ一つ実装します。**優先度順**に、ユーザCRUD→スキル/資格マスタ→MBTI→キャリア履歴、のように段階を追って移植します。
- **tRPCを採用**する場合: プロジェクトに`tRPC`関連パッケージ（例えば`@trpc/server`、`@trpc/next`, `@trpc/react`等）を追加し、サーバ側に`tRPC router`を定義します。`src/server/routers/user.ts`など機能ごとのルータを作り（例：`userRouter`にcreateUser, listUsers, updateUser, deleteUserなどのプロシージャを定義）、それらを合成して`appRouter`を構築します。Next.js App Router環境では**Route Handler**でtRPCエンドポイントを作成できます[15]。クライアント側ではReact Hooks（`useQuery`, `useMutation`）をtRPC用に生成し、コンポーネントから呼び出せるようにします[16]。
- **API Routeを利用**する場合: `src/app/api/`以下にRESTエンドポイントを作成します。例えばユーザ一覧取

得用に`src/app/api/users/route.ts`（GETメソッド対応）、ユーザ新規作成はPOSTメソッドで同じ`users/route.ts`内で処理、ユーザ更新・削除は`users/[id]/route.ts`でHTTPメソッド毎に処理を書く、といった構成にします。各エンドポイント内でPrismaを用いてDBを参照・変更し、JSONレスポンスを返すよう実装します。
- **既存Rustロジックの再現:** Rust側でのビジネスロジック（例えば「activeなユーザ数カウント」や「MBTIタイプ詳細取得」など特定のクエリ処理）はTS側でも忠実に再現します。Rust実装を読み解き、同等のSQLクエリや処理をPrismaクエリやTSロジックに置換します。すでにRust側でドメインロジックとクエリが分離されている場合（クリーンアーキテクチャ風にapplication/domain層が存在[1]）、その知見を活かしてTS側でも必要ならサービス層を設けます。ただし、まずは**動作優先で直接Prisma操作する実装**でも問題ありません。
- **ファイルアップロード対応:** ユーザのプロフィール画像などバイナリデータの扱いについて、Vercel上で完結させるために**Vercel Blobストレージ**を利用することを検討します。Vercel Blobは画像・動画等のファイルを最適化して保存するマネージドストレージであり、ユーザのアバター画像保存用途に適しています[17]。サーバ側でBlob SDKを使って画像をアップロード・取得するAPIを実装し、画像URL等をPrisma経由でユーザプロフィールに紐付けます。こうすることで、画像も含めアプリケーションデータのすべてをVercelプラットフォーム内で管理できます。
- **認証基盤の準備:** （本フェーズでは詳細実装は行いませんが）将来的にログイン機能を追加することを見据え、NextAuth.js（Auth.js）の導入を下準備しておきます。環境変数にGitHubやGoogle OAuthクレデンシャルを登録できるようにするとともに、ディレクトリ構成上`src/app/api/auth/[...nextauth]/route.ts`を置けるようにしておきます。Edge Runtime上でNextAuthを動かすにはミドルウェアと本体を分離する方法など工夫が必要ですが[18][19]、基本的にNextAuthはVercel上で問題なく動作することが確認されています。後で実装を追加しやすいよう、Protected RouteのためのMiddleware（`middleware.ts`）を配置するなどの検討も行っておきます。

フェーズ4: フロントエンドの改修と統合
- Reactフロントエンド側をNext.js上で動作するように改修します。ルーティングはReact RouterからNext.jsのApp Routerに置き換わるため、各ページコンポーネントを`src/app/(group)/page.tsx`または`(group)/[id]/page.tsx`等として実装します。たとえば`Home`ページは`src/app/page.tsx`に、ユーザー一覧ページは`src/app/users/page.tsx`に、といった形です。App Routerではレイアウトを`layout.tsx`で定義できるため、共通レイアウトやナビゲーションは適宜実装します。
- フロント側でこれまで`window.__TAURI__.invoke`（あるいは`@tauri-apps/api`の`invoke`関数）で呼んでいた箇所を全て、新しいAPI呼び出しに差し替えます。tRPCを導入した場合は`useQuery`や`useMutation`フックを使い、例として**ユーザー一覧取得**は`tRPC.user.list.useQuery()`といった形でデータを取得します。REST APIの場合は`fetch('/api/users')`でGETリクエストを送り、JSONをハンドリングします。
- フロントとバックが一体化したことにより、型定義を共有できます。たとえばユーザーデータのTypeScript型を共通`types`モジュールに定義し、サーバ（Prismaからの返却型を整形）でもクライアント（propsや状態管理）でも同じ型を使うことで整合性を保ちます。
- UIの動作確認: 各機能（ユーザのCRUD、スキル・資格の表示、MBTI結果表示、経歴編集など）が一通りNext.js上で動作することを確認します。必要に応じて一時的にダミーデータを使ったり、Prisma Studioや直接DB操作でテストデータを投入しながら、画面が正しくデータを表示・更新できるか検証します。

フェーズ5: 不要コードの整理と最終調整
- 移行が完了したら、旧バックエンド（Rust+Tauri）関連のファイルをリポジトリから削除します。具体的には`src-tauri/`以下のRustコードやTauri設定、`Cargo.toml`等Rustプロジェクトの構成ファイルは不要になるため削除します。Node.js依存でないバイナリや、プラットフォーム固有モジュールもすべて排除します。これによりプロジェクトは純粋なNext.jsアプリの構成になります。
- パッケージの整理: Rust/Tauri関連のnpmパッケージ（`@tauri-apps/api`など）もアンインストールし、package.jsonをクリーンアップします。代わりに追加したNext.js, Prisma, tRPC等の依存関係について、バージョンの整合や不要なものがないかチェックします。
- セキュリティ・パフォーマンス確認: APIルートが認証無しで公開される状態になっているため、将来的な認証導入までの暫定措置としてエッジミドルウェアでアクセス制限を検討します（社内ツールであればIP制限

やBasic認証を一時的にかけるなど）。またビルドしてVercelにデプロイし、各関数のレスポンス性能やCold Start時間を計測します。Edge化した関数のサイズが大きすぎないか（Vercel Edge関数はデプロイサイズ上限1MB程度[20]）も確認します。問題があれば一部関数はNode.js Runtimeへの切り替えを検討します[14][21]。

フェーズ6: Vercelデプロイと動作確認
- GitHub連携またはVercel CLIを用いて、アプリをVercelにデプロイします。デプロイ先で環境変数（DATABASE_URLや、後述するAuthのプロバイダキーなど）を正しく設定します。
- Vercel上でビルドエラーやランタイムエラーがないかログを監視します。特にPrisma周りで「エッジ関数上で動かない」といったエラーが出る場合、Prisma Data Proxyの利用設定を行うか、`export const runtime = 'nodejs'`に切り替えて再デプロイします。
- 最終的に、**ユーザプロフィール画像を含めた全機能がクラウド上で安定動作**することを確認します。Blobストレージに画像をアップロードしそれを表示する流れ、DBへの読み書き、SSR/動的レンダリングの挙動、Edge関数の地理的なレスポンスなどをテストします。問題がなければ移行作業は完了です。

3. 技術スタック／設計選定の理由

Rust+TauriからNext.jsへの移行理由: 最大の理由は**Vercelデプロイとの親和性**です。VercelはサーバレスFunctions（Node.js or Edge）でバックエンドを動かすプラットフォームであり、Web標準のNode/ブラウザ環境以外で書かれたバックエンドは直接ホスティングできません。Rustバイナリを直接Vercelに置くこともできませんし、Tauriはデスクトップ向けフレームワークのためクラウド上での提供に向きません。Next.jsはVercelが生み出したフレームワークであり、**フロントエンドとAPIバックエンドを統合してデプロイ**するユースケースに最適化されています。Next.js上でTypeScriptに統一することで、開発効率も上がり、型定義やコードをフルスタックで共有できるメリットがあります。

Next.js App Router + Edge Runtime: Next.js 13以降のApp Routerは、React 18のサーバコンポーネント機能を活かした最新の構成です。これによりページごとにレイアウトやデータフェッチを宣言的に行え、従来のページズルーターよりスケーラブルです。またEdge Runtimeを利用すれば、グローバルなCDNエッジ上で関数が実行されるため、ユーザからの低レイテンシ応答が期待できます。例えば、日本からのリクエストは東京近辺のEdgeで処理され、欧米からのリクエストはそれぞれ近いリージョンで処理されるため、**地理的な応答時間を短縮**できます。もっともEdge Runtimeには先述のように一部制約がありますが、VercelはPrismaとの統合などを通じてそれを克服しつつあります[5]。EdgeとNodeを使い分けるハイブリッド構成も可能で、パフォーマンスと互換性を両立できます。

Prisma + PostgreSQL（Neon/Prisma Data Platform）: ORMにPrismaを選定したのは、TypeScriptとの相性が良くスキーマ定義から型生成まで自動化してくれるためです。Rustのsqlx同様にマイグレーション管理もでき、開発者体験が向上します。またPrismaは**サーバレス環境に最適化**されており、接続プーリングやデータキャッシュの仕組みが備わった専用ソリューション（Prisma Data ProxyやNeon）を利用できます[22]。これにより、これまで懸念だった「サーバレスでのコネクション数問題」も解決しやすくなります。Vercel上でPrismaを使う例も増えており、安心して採用できます。PostgreSQLを継続利用する点も、既存データをそのまま活かせるためリスクが低く、Prisma対応も問題ありません。

tRPC の採用理由: tRPCはクライアントとサーバ間の通信を**完全に型安全**にしてくれるRPCフレームワークです。GraphQLのようにスキーマ定義やオーバーヘッドを書く必要なく、関数呼び出し感覚でバックエンドの処理をフロントから呼べます。今回、バックエンドとフロントが同じNext.jsプロジェクト内にあるため、tRPCを使うことで関数定義をそのまま共有し、ビルド時に型チェックができる点が非常に有用です。特にスキルシート管理アプリのように多数のCRUD APIがある場合、**エンドポイントごとの型ミスを防ぎ、生産性を向上**させます。またNext.jsとの統合事例も多く、App Router環境でもtRPCの公式サポート（Next.js用アダプタ）が用意されています[23]。一方でREST APIのほうが理解しやすいチームの場合は無理に採用せず、

Next.js組み込みのAPI Routeを使っても構いません。いずれにせよTypeScriptで一貫実装できるため、型の利点は享受できます。

ディレクトリ構成（Bulletproof-React 準拠）： フロントエンドのReact部分については、Bulletproof-Reactのディレクトリ構成を参考にします。Bulletproof-Reactでは `src/` 以下にすべてのソースコードを集約し、機能単位（feature単位）でフォルダを分けるアプローチを取ります [10] [24]。例えば `src/features/user/` の中にユーザ管理に関するコンポーネント、フック、APIクライアント（tRPC呼び出し）などをまとめることで、機能ごとに関心事を閉じ込めます [24] [25]。この構成を取る理由は、**可読性・メンテナンス性の向上**です。機能が増えてもフォルダ単位で増やせばよく、ある機能を削除する際もそのディレクトリを丸ごと消せば関連コードが一掃できます [25]。プロジェクトルート直下が煩雑になるのも避けられ、CI設定等でも `src` 以下を指定して一括管理しやすくなります [26]。Reactコンポーネントだけでなく、今回のようにバックエンドロジックも同居する場合、共通の `src/features` 構成にしておけば「この機能はバックもフロントもここに実装がある」と把握しやすいメリットがあります。

Vercelプラットフォーム活用:　Vercelは単にホスティングするだけでなく、**周辺サービスとの統合**が魅力です。今回言及したVercel Postgres（Neon提携）やBlob Storage、Edge Config、そして認証や画像最適化などの機能も提供されています。技術スタックをNext.js中心に据えることで、Vercelのエコシステムを余すところなく活用できます。例えば、画像アップロードはVercel Blobで行いCDNキャッシュさせる、環境変数や機密情報管理はVercelのDashboardで安全に実施、デプロイプレビューも自動化、といった具合です。さらに将来的にPDF生成機能を追加する場合でも、**サーバレス関数でヘッドレスブラウザを動かすワークアラウンド**が確立されています [27]。以上の理由から、Next.js + Vercel + Prismaという構成は本プロジェクトを安定稼働させる上で最適な選択といえます。

4. 理想的な最終ディレクトリ構成

移行作業完了後のプロジェクト構成を示します。大枠はBulletproof-Reactに沿った `src/` ディレクトリ構造で、Next.js (App Router)の要求するファイル構成と両立するよう調整しています。

```
skill-sheet-maker/
├── prisma/                  # Prisma のスキーマおよびマイグレーション
│   └── schema.prisma         # DBスキーマ定義（PostgreSQL）
├── public/                  # 公開アセット（プロファイル画像の一時保存先など）
├── src/
│   ├── app/                  # Next.js App Router のエントリポイント群
│   │   ├── layout.tsx        # アプリ全体のレイアウトコンポーネント
│   │   ├── page.tsx          # ホームページ（"/"ルート）コンポーネント
│   │   ├── users/            # 以下、各ページ毎のディレクトリ
│   │   │   ├── page.tsx      # ユーザー一覧ページ（"/users"）
│   │   │   └── [id]/page.tsx # ユーザー詳細ページ（"/users/[id]"）
│   │   ├── skills/
│   │   │   └── page.tsx      # スキル一覧ページ（例）
│   │   ├── api/              # APIルート（App Router版ルーティング）
│   │   │   ├── users/
│   │   │   │   ├── route.ts  # /api/users エンドポイントの実装（GET, POST等を定義）
│   │   │   │   └── [id]/route.ts  # /api/users/[id] エンドポイント（GET(id), PUT,
DELETE)
│   │   │   ├── auth/
│   │   │   │   └── [...nextauth]/route.ts  # NextAuth 認証用エンドポイント（今後使用）
│   │   │   └── ...          # 他のAPIエンドポイント（skillsやqualificationsなど必要に応
じて）
```

```
│   │   └── (他のページやエンドポイント)...
│   ├── features/            # フロントエンドの機能モジュール単位のコード
│   │   ├── users/           # "ユーザー管理" 機能に関するまとめ
│   │   │   ├── components/  # ユーザー関連のUIコンポーネント（フォームや一覧テーブル等）
│   │   │   ├── hooks/       # ユーザー関連のカスタムフック（例: useCreateUser, useListUsers）
│   │   │   ├── api/         # ユーザー関連のAPIクライアント（tRPCのフック or fetch呼び出し定義）
│   │   │   └── index.ts     # ユーザー機能モジュールの公開インターフェース
│   │   ├── skills/          # "スキル管理" 機能
│   │   │   └── ...(components, hooks, api, index.ts)
│   │   ├── auth/            # 認証機能（将来的なログイン用 UIやフック）
│   │   └── ... (その他機能)
│   ├── lib/                 # 共通ライブラリ・ユーティリティ
│   │   ├── prisma.ts        # Prisma Client 初期化とエクスポート
│   │   ├── auth.ts          # 認証関連ユーティリティ（例：認可ミドルウェア）
│   │   └── ... (Blobアップロード用関数など)
│   ├── components/          # 全体で共有される汎用UIコンポーネント
│   │   ├── Button.tsx       # 例: ボタンコンポーネント（複数機能で使われるもの）
│   │   └── ... (他の共通コンポーネント)
│   ├── providers/           # アプリケーションプロバイダー（グローバルな状態やコンテキスト）
│   │   └── app.tsx          # App全体に適用するProvider（テーマ, Auth, tRPC client 等）
│   └── styles/              # グローバルスタイルやTailwindのエントリ
│       └── globals.css      # Tailwindのbase, components, utilities をインポート
├── package.json
├── next.config.js
├── tailwind.config.js
├── tsconfig.json
└── ... その他設定ファイル（ESLint, Prettier など）
```

ディレクトリ構成のポイント:

- `src/app`: Next.jsのルーティング構造。App Routerを使い、ページやAPIエンドポイントをフォルダ構造で定義します。Edge/Node Runtimeの設定は各`route.ts`内で指定可能です。例えばDBアクセスを伴う`api/*`は`export const runtime = 'nodejs'`としてNode上で動かし、CDNキャッシュ可能なページは`export const runtime = 'edge'`とする、といった切り分けもできます。
- `src/features`: Bulletproof-Reactに倣った**機能別モジュール**です [24]。各機能ごとにReactコンポーネントや関連フック・API呼び出しを内包させています。これによりコードの見通しが良くなり、新機能追加時もテンプレートに沿って追加可能です [25]。
- `src/lib`: プロジェクト全体で使うサポートコード。特にバックエンドロジック（Prismaの初期化や外部サービス連携）などはここに置き、`api/route.ts`や`tRPC`ルータから呼び出します。AuthやBlobアップロードなど横断的関心事もこちらです。
- `prisma/`: Prisma ORMの設定ディレクトリ。既存のRust sqlxで管理していたマイグレーションは、Prismaのmigrationに移行するか、一度既存DBをそのままPrismaに認識させて運用開始します。今後はPrismaのスキーマファイルが真実の源泉となります。

このような構成にすることで、**フロントエンドとバックエンドが一体化しながらも整理されたコードベース**を実現できます。React側はBulletproofの原則で機能分割され、バックエンドはNext.jsに従いつつ適宜サー

ビス層に切り出されるため、開発者は関心のある機能のディレクトリを開くだけで関連コードをすべて確認できます。結果としてVercelへのデプロイもスムーズで、再現性・拡張性の高いプロジェクトとなります。

5. Vercelにおける追加機能の可能性検証（認証・PDF出力）

認証（ログイン機能）： Next.jsアプリでは、OAuthやメールログイン等の実装にNextAuth.js（現在はAuth.jsと名称変更）が広く使われています。NextAuthはVercel上で公式にサポートされており（開発元がVercel傘下）、GitHubやGoogle認証連携も容易です。Edge Runtimeを用いるMiddlewareでも、トークンによる**簡易な認証チェック**は可能です [18] し、本格的な認証はNode.jsランタイムのRoute Handlerで行えます [21] 。したがって**ログイン機能の追加は十分可能**です。セッション管理にはJWTを使えばサーバレスでもステートレスに実装できますし、Prismaとの相性も良い（Prisma AdapterでユーザDB管理）ため、今後要件に応じて組み込めるでしょう。

PDF生成とダウンロード: スキルシートのPDF出力についても、Vercel上で実現可能です。サーバレス関数でヘッドレスブラウザ（Chromium）を動かし、HTMLをPDF化する方法があります。ただしそのままの`puppeteer`は大容量のChromiumバイナリを含むため、Vercel（AWS Lambda 50MB制限）では動作しません [27] 。解決策として**軽量版Chromiumとpuppeteer-core**を使う手法が知られています [28] 。例えば`@sparticuz/chromium`というパッケージと`puppeteer-core`を組み合わせ、Lambda対応ビルドを利用します。このアプローチにより、Vercel Function内でHTMLをレンダリングしてPDFバッファを生成し、適切なHTTPヘッダを付けてダウンロードレスポンスとして返すことができます [29] [30] 。Next.js App Routerでは、`app/api/your-pdf/route.ts`内で上記処理を実装し、クライアントからそのエンドポイントを叩いてPDFを取得するといった構成になります。したがって、**PDF生成と手元ダウンロードもVercel上で実装可能**であり、若干の工夫（軽量ツールの採用）は必要ですが十分実現できます。

結論: 認証機能もPDF出力機能も、Next.js (TypeScript) + Vercel環境でサポート可能な範囲です。ログイン機能はNext.js標準のAuthライブラリで容易に追加でき、PDF生成もサーバレス関数の活用で解決策が存在します。これらを踏まえ、本計画に沿ってバックエンドをTypeScript化しVercelに最適化することで、skill-sheet-makerアプリはクラウド上で安定稼働し、将来的な機能拡張（ユーザ認証や帳票出力など）にも柔軟に対応できるようになるでしょう。

参考文献・出典:

- 現行Rust+Tauriバックエンドで定義されているコマンド群（ユーザ・スキル等のAPI機能） [3] [4]
- Bulletproof-React に基づくReactプロジェクトのディレクトリ構成ガイド [10] [24] [25]
- Next.js Edge Runtime の制約（Node.js API非対応の例） [6] [8]
- Vercel + Prisma(Postgres) によるサーバレス/エッジDB接続ソリューション [5]
- Vercel Blob ストレージの概要と画像ファイル保存用途への適合 [17]
- Vercel Serverless Function上でのPDF生成（puppeteer利用）の課題と解決策 [27] [28]

---

[1] [3] [4] main.rs
https://github.com/MasayukiYamagishi/skill-sheet-maker/blob/3978a7894c0c36170e179df85076c94eb2df6772/src-tauri/src/main.rs

[2] [12] [13] README.md
https://github.com/MasayukiYamagishi/skill-sheet-maker/blob/3978a7894c0c36170e179df85076c94eb2df6772/README.md

[5] [22] Prisma for Vercel
https://vercel.com/marketplace/prisma

6 7 API Reference: Edge Runtime | Next.js
https://nextjs.org/docs/app/api-reference/edge

8 14 18 19 21 typescript - i got this error "The edge runtime does not support Node.js 'crypto' module." when integrating my pg db database to my nextjs 14 project using auth.js - Stack Overflow
https://stackoverflow.com/questions/78469850/i-got-this-error-the-edge-runtime-does-not-support-node-js-crypto-module-wh

9 Home.tsx
https://github.com/MasayukiYamagishi/skill-sheet-maker/blob/3978a7894c0c36170e179df85076c94eb2df6772/src/app/routes/Home.tsx

10 11 24 25 26 "bulletproof-react" is a hidden treasure of React best practices! - DEV Community
https://dev.to/meijin/bulletproof-react-is-a-hidden-treasure-of-react-best-practices-3m19

15 solaldunckel/next-13-app-router-with-trpc - GitHub
https://github.com/solaldunckel/next-13-app-router-with-trpc

16 Let's Build a Full-Stack App with tRPC and Next.js App router
https://dev.to/itsrakesh/lets-build-a-full-stack-app-with-trpc-and-nextjs-14-29jl

17 Vercel Storage
https://vercel.com/docs/storage

20 Rendering: Edge and Node.js Runtimes - Next.js
https://nextjs.org/docs/14/app/building-your-application/rendering/edge-and-nodejs-runtimes

23 How to use tRPC with Next.js App Router - Brock Herion
https://brockherion.dev/blog/posts/how-to-use-trpc-with-nextjs-app-router/

27 28 29 30 Nextjs: Generate PDF's on Vercels Serverless Functions using Puppeteer | Nabil Benhamou - Frontend Engineer, Engineering Manager, Designer, Developer Experience Advocate
https://www.thebiltheory.com/blog/use-puppeteer-on-aws-lambdas