



木幅アルゴリズムの学習システムの構築 進捗状況

B4 小林紹子

November 6, 2025



目次

1 画像生成について

2 データベース使用技術

- Prisma
- SQLite

3 データベース構築の流れ

4 今後の課題

5 参考文献



react-graph-visについて

- 調べた結果、画像を自動で作成するのは難しそう。
- 画像を事前に保存してデータベース登録する方法になる。
- したがって、React Flow を使用する方向。
- うまく図形になるように調整の仕方を調査する。



データベースの使用技術

- Prisma ORM
- SQLite

Prisma

- Node.js と Typescript 向けのオープンソースの ORM[1].
- ORM (Object Relational Mapping) :
プログラミング言語のエンティティ (オブジェクト) とそれに対応するデータベース要素との関係を抽象化するプロセス [2].
—————>SQL 文を直接書く必要がなく、データベース操作を簡単にしてくれる.
- 特徴
 - 型安全性 : Typescript との相性がいいため、コンパイル時にエラーを検出しやすい.
 - 直感的な API : SQL を書く必要がなく、オブジェクトを操作する感覚でデータベース操作が可能.
 - データベース非依存 : MySQL, SQLite など様々な DBMS に対応.



Prisma ORM の機能

- **Prisma Client**：アプリから DB を操作するためのライブラリ.
- **Prisma Migrate**：データベースの変更記録と, schema.prisma に書いた内容を DB に反映する技術.
- **Prisma Schema**：テーブル構造やデータベースの種類を定義するファイル.
→これらを schema.prisma というファイルに書いて DB 操作していく.



SQLite

- オープンソースで軽量の RDBMS.
- サーバーとしてではなくアプリケーションに組み込むことで利用.
- ライブラリとして使用可能なため設定不要な**自己完結型システム**.
- マルチプラットフォームに対応（Linux, Windows, iOS など）.
- 1つのテーブルカラムの中に、複数のデータ型を格納可能.

Next.js + React + Prisma + SQLite によるデータベース構築の流れ |

① Prisma のセットアップ

- パッケージをインストール
- 初期化コマンドで `schema.prisma` を生成
- SQLite と接続設定を行う

② スキーマ定義 (DB 設計)

- `prisma/schema.prisma` にモデルを記述
- 例: Problem テーブル (`id`, `text`, `image`, `answer`)

③ マイグレーション実行

- `npx prisma migrate dev --name init`
- スキーマをもとに SQLite ファイル (`dev.db`) を生成

④ Prisma Client 生成

- TypeScript から DB 操作が可能になる
- `@prisma/client` を利用

Next.js + React + Prisma + SQLite によるデータベース構築の流れ II

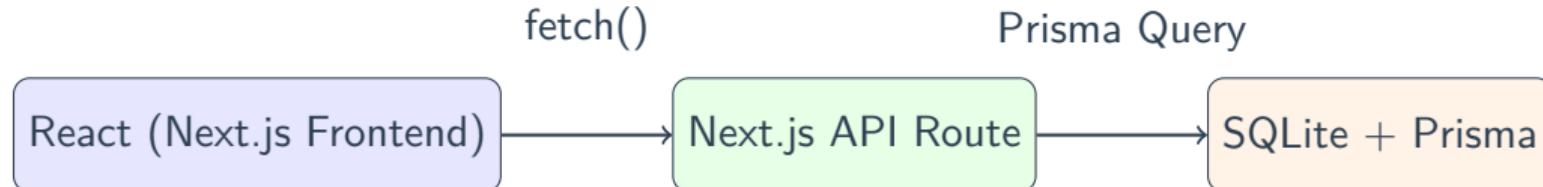
5 Next.js API Routes の作成

- app/api/problems/route.ts に API を定義
- GET : データ取得, POST : データ登録

6 フロントエンド (React) でデータ表示

- fetch("/api/problems") でサーバーから取得
- 問題文や画像を動的に表示

システム構成図（概要）



- フロント（React）：UI 表示・ユーザ入力
- API ルート（Next.js）：DB 操作用の中間層
- Prisma + SQLite: データ永続化

ファイル構造

```
app/
  └── api/
    ├── problems/
    │   ├── route.ts ← GET, POST (出題・解答 API)
    │   └── generate/
    │       ├── route.ts ← 問題を DB に追加する API
    └── components/
        └── ProblemCard.tsx ← 問題表示用コンポーネント
  └── lib/
      └── prisma.ts ← Prisma クライアント
  └── src/
      └── page.tsx ← 問題ページ (React Flow 埋め込み)
```

Step 1: Prisma の初期化

```
1 # Prisma の導入  
2 npm install prisma --save-dev  
3 npx prisma init  
4  
5 # SQLite を使用するように設定 (.env)  
6 DATABASE_URL="file:./dev.db"
```

Step 2: Prisma スキーマ定義 (prisma/schema.prisma)

```
1 datasource db {  
2     provider = "sqlite"  
3     url      = env("DATABASE_URL")  
4 }  
5  
6 generator client {  
7     provider = "prisma-client-js"  
8 }  
9  
10 model Problem {  
11     id      Int      @id @default(autoincrement())  
12     text    String  
13     image   String?  
14     answer  Boolean  
15 }
```



Step 3: データベース作成

```
1 # マイグレーションの作成と反映  
2 npx prisma migrate dev --name init  
3  
4 # データベースを確認  
5 npx prisma studio
```



Step 4: Prisma Client の使用 (src/lib/prisma.ts)

```
1 import { PrismaClient } from "@prisma/client";
2 const prisma = new PrismaClient();
3
4 export default prisma;
```

これにより、どの API ルートでも DB に接続可能となる。

Step 5: バックエンド側 (src/app/api/problems/route.ts) |

```
1 import { NextResponse } from "next/server";
2 import prisma from "@/lib/prisma";
3
4 // GET: 問題一覧を取得
5 export async function GET() {
6   const problems = await prisma.problem.findMany();
7   return NextResponse.json(problems);
8 }
9 // POST: 回答を送信し、正誤を判定
10 export async function POST(req: Request) {
11   const { id, answer } = await req.json();
12   const problem = await prisma.problem.findUnique({ where: { id } });
13
14   if (!problem) {
15     return NextResponse.json({ correct: false, message: "問題が見つかりません" });
16 }
```

Step 5: バックエンド側 (src/app/api/problems/route.ts) II

```
17
18     const correct = problem.answer === (answer === "true");
19     return NextResponse.json({
20         correct,
21         message: correct ? "正解！" : "不正解",
22     });
23 }
```

Step 5: フロントエンドで取得 (src/app/problems/page.tsx)

```
1 "use client";
2 import { useEffect, useState } from "react";
3 %型定義
4 type Problem = {
5     id: number;
6     text: string;
7     image?: string;
8 };
9 //DBからデータの取得
10 export default function ProblemsPage(){
11     const [problems, setProblems] = useState<Problem[]>([]);
12     const [message, setMessage] = useState("");
13
14     useEffect(() => {
15         fetch("/api/problems")
```

Step 5: フロントエンドで取得 (src/app/problems/page.tsx)

||

```
16     .then((res) => res.json())
17     .then((data) => setProblems(data));
18 }, []);
19 //回答の送信
20 async function handleAnswer(id: number, answer: boolean) {
21   const res = await fetch("/api/problems",
22     method: "POST",
23     headers: {"Content-Type": "application/json"},
24     body: JSON.stringify({id, answer}),
25   );
26   const data = await res.json();
27   setMessage(data.message);
28
29 }
30
```

Step 5: フロントエンドで取得 (src/app/problems/page.tsx)

III

```
31 | return(
32 |   <div style={{padding: 20}}>
33 |     <h1>◎× 問題</h1>
34 |
35 |     {problems.map((p) => (
36 |       <div key={p.id} style={{marginBottom: 30}}>
37 |         <p>{p.text}</p>
38 |         {p.image && <img src={p.image} alt="問題画像" width={200}></img>}
39 |         <div>
40 |           <button onClick={() => handleAnswer(p.id, true)}>◎</button>
41 |           <button onClick={() => handleAnswer(p.id, false)}>×</button>
42 |         </div>
43 |       </div>
44 |     )));
45 |     {message && <h3>{message}</h3>}
```

Step 5: フロントエンドで取得 (src/app/problems/page.tsx)

IV

```
46     </div>
47   );
48 }
```



今後の課題

- 正解・不正解をそれぞれ個別に出力・保存.
- 画像の出力.
- 今は○×問題だけだが、数値の入力問題でも正答判定が可能.
- 問題が一覧でなく、ページ分割可能.



参考文献 |

- [1] Prisma. <https://www.prisma.io/docs/>, (参照日 = 2025/10/27).
- [2] オブジェクトリレーションナルマッピング (orm) とは.
<https://aws.amazon.com/jp/what-is/object-relational-mapping/>, (参照日 = 2025/10/27).