

Prismaを試してみた








Press Space for next page →





自己紹介

-  飯野陽平 (wheatandcat)
-  フリーランスエンジニア (シェアフル株式会社CTO)
-  Blog: <https://www.wheatandcat.me/>
-  Booth: <https://wheatandcat.booth.pm/>
-  今までに作ったもの
 - memoir
 - ペペロミア
 - Atomic Design Check List

Prismaとは？

- Node.js製のORM
- RDB周りの処理を簡易に扱えるようにする
- Schemaファイルから型情報を自動生成
- 以下のDB対応をサポート
 - PostgreSQL、MySQL、SQL Server、SQLite、MongoDB

モチベーション

[Hasura](#)や[AWS Amplify](#)を使用することで、
工数を掛けずお手軽にRESTful APIやGraphQL APIを作成できるようになった。

[Why Prisma?](#)

モチベーション

[Hasura](#)や[AWS Amplify](#)を使用することで、
工数を掛けずお手軽にRESTful APIやGraphQL APIを作成できるようになった。

ただし上記は使用できるプラットフォームが固定されて、
大規模な開発では柔軟性が足りないこともある。

[Why Prisma?](#)

モチベーション

[Hasura](#)や[AWS Amplify](#)を使用することで、
工数を掛けずお手軽にRESTful APIやGraphQL APIを作成できるようになった。

ただし上記は使用できるプラットフォームが固定されて、
大規模な開発では柔軟性が足りないこともある。

そこでエンジニアの工数も削減しつつ、
システムの柔軟性を持たせることのできるPrismaを試してみる。

[Why Prisma?](#)

サンプルを作ってみる

以下のチュートリアルをベースに、どんな感じで実装するのか試してみる。

- [Prisma チュートリアル](#)

CLIツールで初期設定

以下のコマンドを実行することで初期設定を行なう。

```
$ npx prisma init
```


CLIツールで初期設定

以下のコマンドを実行することで初期設定を行なう。

```
$ npx prisma init
```

上記のコマンドで以下のファイルが生成される。

```
.
├─ .env
└─ prisma
    └─ schema.prisma
```

CLIツールで初期設定

以下のコマンドを実行することで初期設定を行なう。

```
$ npx prisma init
```

上記のコマンドで以下のファイルが生成される。

```
.
├── .env
└── prisma
    └── schema.prisma
```

■ prisma/schema.prisma

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}
```

CLIツールで初期設定

以下のコマンドを実行することで初期設定を行なう。

```
$ npx prisma init
```

上記のコマンドで以下のファイルが生成される。

```
.
├─ .env
└─ prisma
    └─ schema.prisma
```

■ .env

```
DATABASE_URL="postgresql://johndoe:randompassword@localhost:5432/mydb?schema=public"
```

DBのマイグレーション①

PrismaではSchemaファイルを修正することでマイグレーションファイルを生成して実行していく。

■ prisma/schema.prisma

```
model Post {
  id          Int          @default(autoincrement()) @id
  createdAt   DateTime      @default(now())
  updatedAt   DateTime      @updatedAt
  title       String        @db.VarChar(255)
  content     String?
  published   Boolean        @default(false)
  author      User          @relation(fields: [authorId], references: [id])
  authorId    Int
}

model User {
  id          Int          @default(autoincrement()) @id
  email       String        @unique
  name        String?
  posts       Post[]
}
```

DBのマイグレーション②

以下のコマンドを実行する。

```
$ npx prisma migrate dev --name init
```

DBのマイグレーション②

以下のコマンドを実行する。

```
$ npx prisma migrate dev --name init
```

Schemaファイルを元に以下のSQLファイルを生成する。

■ prisma/migrations/20220321025430_init/migration.sql

```
-- CreateTable
CREATE TABLE "Post" (
  "id" SERIAL NOT NULL,
  "createdAt" TIMESTAMP(3) NOT NULL DEFAULT CURRENT_TIMESTAMP,
  "updatedAt" TIMESTAMP(3) NOT NULL,
  "title" VARCHAR(255) NOT NULL,
  "content" TEXT,
  "published" BOOLEAN NOT NULL DEFAULT false,
  "authorId" INTEGER NOT NULL,

  CONSTRAINT "Post_pkey" PRIMARY KEY ("id")
);
```

DBのマイグレーション②

以下のコマンドを実行する。

```
$ npx prisma migrate dev --name init
```

Schemaファイルを元に以下のSQLファイルを生成する。

■ prisma/migrations/20220321025430_init/migration.sql

```
-- CreateTable
CREATE TABLE "User" (
  "id" SERIAL NOT NULL,
  "email" TEXT NOT NULL,
  "name" TEXT,

  CONSTRAINT "User_pkey" PRIMARY KEY ("id")
);
```

DBのマイグレーション②

以下のコマンドを実行する。

```
$ npx prisma migrate dev --name init
```

Schemaファイルを元に以下のSQLファイルを生成する。

■ prisma/migrations/20220321025430_init/migration.sql

```
-- CreateIndex
CREATE UNIQUE INDEX "User_email_key" ON "User"("email");

-- AddForeignKey
ALTER TABLE "Post" ADD CONSTRAINT "Post_authorId_fkey" FOREIGN KEY ("authorId") REFERENCES "User"("id") ON DELETE RESTRICT ON UPDATE CASCADE;
```


Prisma Clientからデータを取得①

以下のファイルを作成する。

■ index.ts

```
import {PrismaClient} from "@prisma/client";

const prisma = new PrismaClient()

async function main() {
  const allUsers = await prisma.user.findMany()
  console.log(allUsers)
}

main()
  .catch((e) => {
    throw e
  })
  .finally(async () => {
    await prisma.$disconnect()
  })
```

Prisma Clientからデータを取得①

以下のファイルを作成する。

■ index.ts

```
import {PrismaClient} from "@prisma/client";

const prisma = new PrismaClient()

async function main() {
  const allUsers = await prisma.user.findMany()
  console.log(allUsers)
}

main()
  .catch((e) => {
    throw e
  })
  .finally(async () => {
    await prisma.$disconnect()
  })
```

Prisma Clientからデータを取得②

以下のコマンドを実行する。

```
$ npx ts-node index.ts
```

Prisma Clientからデータを取得②

以下のコマンドを実行する。

```
$ npx ts-node index.ts
```

テストデータを挿入して実行すると以下のように値が返ってくる。

```
[
  {
    id: 1,
    email: 'test@prisma.io',
    name: 'test',
    posts: [
      {
        id: 1,
        createdAt: 2022-01-01T12:00:00.985Z,
        updatedAt: 2022-01-01T12:00:00.986Z,
        title: 'Hello World',
        content: "foo bar baz",
        published: false,
        authorId: 1
      }
    ]
  }
]
```

Schemaファイルからtypeを自動生成

Prisma Clientから生成されたコードはSchemaファイルからtypeも生成してくれるので、VSCodeでコーディングした際に、TypeScriptの補完も効く。

```
6
7  async function main() {
8    await prisma.user.create({
9      data: {
10        em
11      },
12    })
13
14    const allUsers = await prisma.user.findMany({
15      include: {
16        posts: true,
17      },
18    })
```



The image shows a VS Code editor with a TypeScript file. The code is a Prisma Client query. On line 10, the user is typing 'em' inside a 'data' object. A dropdown menu appears, showing two options: 'email' (with a cube icon) and 'vFormEmailRules' (with a square icon). The 'email' option is selected, and the text '(property) email: string' is shown to the right. The 'vFormEmailRules' option is also visible, with the text 'v-form email rule' shown to the right.

Prisma Clientの書き方①

条件に一致するデータを1件抽出

```
const user = await prisma.user.findUnique({  
  where: {  
    id:1,  
  }  
})
```

条件に一致するデータ抽出(id >= 20)

```
const users = await prisma.user.findMany({  
  where: {  
    AND :{  
      id: { gte: 20 }  
    }  
  }  
})  
  
console.log(users)
```

■ 参考1: Prisma チートシート

Prisma Clientの書き方②

リレーションのSQLの発行を含める/含めない

```
const users = await prisma.user.findMany({  
  include: {  
    posts: false,  
  },  
})
```

sort & select

```
const users = await prisma.user.findMany({  
  select: {  
    email: true,  
  },  
  orderBy: [  
    {  
      name: 'desc',  
    },  
  ],  
})
```

Prisma Clientの書き方③

Transaction & Rollback

```
async function transfer(from: string, to: string, amount: number) {
  return await prisma.$transaction(async (prisma) => {
    const sender = await prisma.account.update({
      data: { balance: { decrement: amount } },
      where: { email: from },
    })
    if (sender.balance < 0) {
      throw new Error(`${from} doesn't have enough to send ${amount}`)
    }
    const recipient = prisma.account.update({
      data: {
        balance: { increment: amount },
      },
      where: { email: to },
    })
    return recipient
  })
}
```

■ 参考: Prismaでのトランザクションとロールバック

SQLの発行のログを見たい

```
const prisma = new PrismaClient({  
  log: ["query"],  
})
```

SQLの発行のログを見たい

```
const prisma = new PrismaClient({  
  log: ["query"],  
})
```

以下のコードを実行した場合。

```
await prisma.user.create({  
  data: {  
    name: 'Alice',  
    email: 'alice@prisma.io',  
    posts: {  
      create: { title: 'Hello World' },  
    },  
  },  
})  
  
const allUsers = await prisma.user.findMany({  
  include: {  
    posts: true,  
  },  
})
```

SQLの発行のログを見たい

```
const prisma = new PrismaClient({
  log: ["query"],
})
```

以下のようにログが表示される。

```
prisma:query BEGIN
prisma:query INSERT INTO "User" ("email","name") VALUES ($1,$2) RETURNING "User"."id"
prisma:query INSERT INTO "Post" ("createdAt","updatedAt","title","published","authorId") VALUES ($1,$2,$3,$4,$5) RETURNING "Post"."id"
prisma:query SELECT "User"."id", "User"."email", "User"."name" FROM "User" WHERE "User"."id" = $1 LIMIT $2 OFFSET $3
prisma:query COMMIT
prisma:query SELECT "User"."id", "User"."email", "User"."name" FROM "User" WHERE 1=1 OFFSET $1
prisma:query SELECT "Post"."id", "Post"."createdAt", "Post"."updatedAt", "Post"."title", "Post"."content", "Post"."published", "Post"."authorId" FROM "Post" WHERE "Post"."authorId"
```

Prisma Studio

以下のコマンドを実行するとDBの状態をブラウザからGUIで確認/操作ができる。

```
$ npx prisma studio
```

各DBシステムでサポートしているので、お手軽にDBの中身进行操作したい際に使用する。

User X +				
↻ Filters None Fields All Showing 1 of 1 Add record				
<input type="checkbox"/>	id #	email A	name A?	posts []
<input type="checkbox"/>	1	alice@prisma.io	Alice	1 Post

GraphQLとの親和性①

もともと、Prisma v1はGraphQLを前提としたノーコードで扱える系のフレームでしたが、v2のタイミングで方向転換をして、ORMとして切り離しを行い、今の形式になっている。

Prisma 2 is Coming Soon

GraphQLとの親和性①

もともと、Prisma v1はGraphQLを前提としたノーコードで扱える系のフレームでしたが、v2のタイミングで方向転換をして、ORMとして切り離しを行い、今の形式になっている。

[Prisma 2 is Coming Soon](#)

ライブラリ的にはGraphQLと切り離されたが、公式的には、GraphQLと一緒に使うことを推奨している節もあるので、基本は合わせて使う方向性が良いされている。

[How Prisma and GraphQL fit together](#)

GraphQLとの親和性②

実際の親和性の話に関しては以下の参考記事を参照。

- [GraphQLと相性の良いORM Prisma。](#)
- [Apollo ServerとPrismaではじめるGraphQL API開発入門](#)

まとめ

Prismaもv3になり、プロダクトで使えるくらいの品質になったのかなと感じた。

Prisma + GraphQL + [GraphQL Code Generator](#) の組み合わせで、Schemaファイルから、すべてのエンティティの型情報を自動生成できるのは魅力的なので、どこかのプロダクトで運用を試してみたい。

個人的にはNode.js自体のコーディングし辛い問題が若干あるので、ここだけ悩み中。（**Deno**で書ければ解決しそうな予感もするので来月調査してみる）

ご清聴ありがとうございました