はじめに

本書はウェブフロントエンドとバックエンドを併用して実践的なプロトタイピングについて解説する本です。

ウェブフロントエンドのみのプロトタイピングに使える資料はそれなりにあります。Firebase のような特定の機能をバックエンドに持った本も多く見られます。

既存のそれらの本はバックエンドにはあまり踏み込みません。NoSQL やファイルストレージ、認証サービスなどを利用することがほとんどです。

この本の特徴は 2 点あります。

1. 実際のプロトタイピングを様子を解説すること（ストーリー仕立てではありませんが、プロトタイピングでよくありそうなシチュエーションを想定して、プロトタイピングを解説します）
2. Next.js および Hasura GraphQL Engine によるバックエンドを解説する

本書で紹介する Hasura GraphQL Engine は SQL を使った本格的な RDB をフロントエンドからお手軽に扱えるものです。また Next.js はサーバーサイド JavaScript と、フロントエンドのコードをある程度透過的に混ぜ合わせるユニバーサルフレームワークです。これらを使えば、バックエンドに踏み込んだ、より本物に近く、幅広いプロトタイピングが可能です。

プロトタイピングで大切なことは、日頃から慣れている技術を武器にして、手早くプロトタイプを作り上げるかです。

フロントエンドだけのプロトタイピングには限界があります。バックエンドも込みでプロトタイピングできれば強い武器になることでしょう。

本書では、そういったより実践に即したプロトタイピングを紹介します。

対象読者

TypeScript + React が分かること、もしくは別の本などで React や TypeScript について勉強できるひとを前提とします。本書ではページ数その他の都合により React や TypeScript の詳細には踏み込みません。

プロトタイピングのコツを知りたいひと、フロントエンドのプロトタイピング（React/Next.js）を覚えたいひと、バックエンドのプロトタイピング（Next.js/Hasura GraphQL Engine）を覚えたいひとを対象読者と想定して本書を執筆しています。

- React + TypeScript の開発経験があるか、それらを別の本なりで埋められるひと
- プロトタイピングのコツを知りたいひと
- フロントエンドプロトタイピングを覚えたいひと
- バックエンドプロトタイピングを覚えたいひと

本書を読むに当たって

本書には ○○.js みたいなライブラリやフレームワークの名称が度々登場しますが、初回の紹介以外では .js 部分を省略します。

例: Next.js → Next

ただし Node.js のみ、Node.js と表記します。**Node** という単語があまりに一般用語かつ、技術系文書に頻出しうるためです。

モチベーション

幾つかの機会があり、筆者が思っていた以上に、プロトタイピングのコツを知らない人が多いようなのでそれをまとめたくなったことと、Next.js + Hasura GraphQL Engine が意外に凄いヤツだったので、それを紹介したくなったことが、本書を執筆したモチベーションです。

プロトタイピングの章では、試行錯誤について生々しさを重視して書いています。プロトタイピングに慣れていない人にも、プロトタイピングのやり方が伝われば幸いです。

もっとも、本書で行っているプロトタイピングは筆者の経験に基づくものなので、もっとうまいやり方があるかもしれません。もしご存じの方は筆者に教えていただけると、筆

者がとても喜びます。

プロトタイピングとは

プロトタイピングとは、手早く、実際に動く物を作って触りながら、他の人からのフィードバックを得て、よりよい物を設計するという技法です[*1]。

さて、一般的な設計からの開発という流れの場合、アイデアや機能を机上で考えても細部が異なる、様々な制約から思っていたものとは違うものを作らざるをえないこともしばしばです。

「〇〇が欲しい」と口にしている人がいたとしても、実装されたら「実は思ってたのと違った」というようになってしまうケースが多いことを皆さんも体験しているのではないでしょうか？

> It’s really hard to design products by focus groups. A lot of times, people don’t know what they want until you show it to them.
>
> [BusinessWeek, May 25, 1998]

これは、スティーブジョブズの名言で、ものすごく意訳すると「人は形にして、見せてみないと、本当に欲しいものが何なのか分からない」というものです。さらに源流をたどると、ヘンリーフォードが言った「もし顧客に、彼らの望む物を聞いたら、彼らは『もっと速い馬が欲しい』と応えていただろう」という言葉もあります。

そこでプロトタイピングの出番です。

プロトタイピングは、開発の一工程として作られるケースもありますし、リーン開発のように MVP（概念検証ができる最小限のプロダクト）を作り、市場に問うというケースもあります。

ひとは言葉だけでやりとりをするとすれ違いがちです。言葉の曖昧さによって解釈の余地がそれぞれのひとで異なるためです。より具体的になればなるほど、議論は適切になります。

たとえば、言葉だけのやりとりよりは図を交えたやりとりの方がより正確でしょう。さらにいうと、図があっても動きがあるわけではないので、動きも含めたものであればより良いでしょう。

実際に動くことで、議論の曖昧さを排除するためにプロトタイピングがあります。

[*1] 開発工程としては設計に該当しますが、やっていることは設計と実装をまたぐものです。

本書の構成

本書は、Next.js + Hasura GraphQL Engine の詳細を必要以上に解説するための本ではありません。プロトタイピングのノウハウを伝えることと、バックエンドに踏み込んだプロトタイピングについて説明することの 2 点が主目的です。

- 第 1 章「プロトタイピング」 ではプロトタイピングのコツについて詳細
- 第 2 章「要素技術」 では Next.js, Hasura GraphQL Engine, GraphQL code generator, など要素技術の概要のみ
- 第 3 章「ローカルで開発しよう」でローカル環境のセットアップ手順
- 残りの章は、ブログサービスをプロトタイピングするという設定のもと、実践的なプロトタイピング

という構成にしてあります。

プロトタイピングの心得

プロトタイピングは手早くです。またプロトタイピングは容易に捨てられるようにしておくべきでしょう。また人に見せて、改良をする前提のため、容易に変更できるようにしておく必要もあります。

フロントエンドとバックエンド

ウェブ技術を使ったプロトタイピングでは、データを決め打ちにする、データの登録はできるが揮発性にする、ローカルストレージに入れるなどといった形にすることで、フロントエンドのみのプロトタイピングを作ることがあります。

とくにフロントエンド開発とバックエンド開発でチームが分かれているような場合は、そういった手法をとることが多いでしょう。

Firebase のような特定の機能を提供してくれるサービス（BaaS）を使う事例もあります。バックエンドでできることは限定されますが、ある程度実用的なプロトタイピングも可能です。

プロトタイピングにおいては日頃から慣れた手段が重要な武器になります。プロトタイピングは、手早く、実際に動く物を作る必要があります。たとえば Firebase を使うなら

ば、ある程度慣れておかないとプロトタイピングとして成立しません。

この本では Next.js と Hasura GraphQL Engine について解説しています。Firebase を使う場合に比べて、圧倒的な自由度でバックエンドにもがっつり踏み込んだプロトタイピングが可能になります。

武器を増やしておいて損はないでしょう。

Next.js

Next.js（以後 Next）は React を使うための軽量フレームワークです。Next にインスパイアを受けて誕生した Nuxt.js は手厚いフレームワークですが、元祖 Next はシンプルさ・軽量さが特徴です。

- ビルドツールである Webpack の面倒を見てくれて、一般的なものであれば特に設定なしでも動く
- ページルーティングをしてくれる
- サーバーサイドレンダリング（SSR）や静的サイト生成（SSG）や、モバイル向けの AMP 生成をしてくれる
- バックエンドとフロントエンドをユニバーサルに繋ぐことができる

という機能があり、JavaScript インフラとでもいうべきフレームワークです。

SSR/SSG/AMP はともかく他は、プロトタイピングにも向いた機能であり、とくに最後に挙げたユニバーサルというのは、プロトタイピングにおいて、とても興味深いものです。

Next では API サーバーなどバックエンドのコードと、フロントエンドのコードを混在して記述ができます。たとえば、バリデーションやビジネスロジックなど、フロントエンドとバックエンドで共有したいコードがあれば、簡単に共有ができるのです。

一般的に、ウェブフロントエンドは JavaScript（及び TypeScript や他 AltJS）以外の選択肢は、事実上ありません。

まだ WebAssembly を実践に投入するには問題がありますし、ある程度熟れてきたとしても、JavaScript/TypeScript の成熟したエコシステムを捨てることは難しいでしょう。

バックエンドには、Ruby, PHP, Java など様々な選択肢がありますが、こちらでも JavaScript を採用できれば、不要な技術スタック分散を抑えられます。

複数の言語を使うよりは、単一の言語で行う方が手っ取り早いでしょう。ユニバーサルなフレームワークである Next なら言語面以外でも、コードをある程度シームレスに扱え

るようになっています。

また、Next は、特にバージョン 9 以降で精力的に使い勝手の改善をしていて、利用シーンも増え続けています。もしまだ触ったことがなければ、試してみて損は無いはずです。

GraphQL

ウェブやモバイルアプリの API は REST 思想に基づいたものが一般的でしたが、最近は GraphQL も使われるようになりました。

GraphQL は元々 Facebook 社が開発したものですが、今は The GraphQL Foundation という団体が定義する仕様です。

Apollo Server[*2] を使い、既存の REST API を、クライアントが扱いやすいように抽象化・変換する BFF（Backends For Frontends）として使う事例が多いです。

Apollo Server を Next と同居して動かすという選択肢も良いでしょう。

Hasura GraphQL Engine

Hasura GraphQL Engine は RDB（PostgreSQL や MySQL）に特化した GraphQL サーバーです。

Hasura では、テーブル構造に合った GraphQL API が自動生成されるため、RDB のテーブル（とリレーションなど）さえあれば、ウェブアプリから簡単に RDB を操作できます。

Rails の ActiveRecord のような ORM は、Ruby のオブジェクトとして抽象化された RDB 上のデータ操作を行うためのものです。Hasura GraphQL Engine は RDB の操作を GraphQL API に抽象化するものだと思えばいいでしょう。

オブジェクトとして直接アクセスするのではなくて API としてアクセスするのは、ワンクッション必要に感じるかもしれませんが、GraphQL API はその API アクセスを、全自動でコード生成可能なため、開発者から見るとダイレクトに RDB アクセスしているのと変わりありません。

また、コード生成では、TypeScript による型も適切に生成されるため、RDB の中のデータは自動生成された型によって保証されます[*3]。

[*2] https://www.apollographql.com/docs/apollo-server/

[*3] JavaScript の HTTP クライアントで REST API を叩く場合、返ってくるデータに型はありません。エンドポイント、アクセス方法などによって、帰ってくるデータには無限の可能性があるため、型を定義でき

著者による個人的な見解

2020 年 9 月時点での著者の個人的な見解では、Hasura GraphQL Engine はとても便利で凄いプロダクトだと思うものの、まだ、使い方に気をつけなければいけないプロダクトというものです。

データの全てを Hasura に依存する、あるいは凝った認証をするなどといった使い方は、必要な操作手順、開発が分散してあれこれやらなければならなくなり、かなり苦痛です（あくまで筆者の感じ方です）。

サーバーには Next.js 及び Apollo Server を採用して BFF（Backends For Frontends）を実現して、フロントエンドから唯一見える存在としたうえで、BFF の背後に Hasura GraphQL Engine という構成が望ましいと考えています。

現状では Node.js には、標準的に採用すべき良い ORM がないため、RDB にアクセスするための手段として Hasura GraphQL Engine は素晴らしい選択肢です。

もちろん今後の Hasura GraphQL Engine の進化によっては、単体で BFF をやれるようになるかもしれませんが、あくまで現状としてはです。

開発元の Hasura 社は、うまく資金調達を行い、急成長をしているスタートアップであり、今後の Hasura GraphQL Engine はもっと良い形になっているはずです。ちょうど先日、MySQL への対応が発表されたところです。

Hasura GraphQL Engine の認証について触れましたが、本書ではプロトタイピングであるという前提のもと、認証を無視して、ダイレクトに Hasura GraphQL Engine にアクセスしています。

本書について

- プロトタイピングのサンプルコードについては https://github.com/erukiti/prototyping-sample にて公開しております
- 本書のサポートページは https://rabbit-house.tokyo/books/proto-book/ です。

ないためです。そのため GprahQL に限りませんが、API からのアクセスコード自動生成なら、型情報を API 定義から参照できるため、型安全に API アクセスできるという利点があります。

履歴

日付	更新情報
2020/09/12	初版 rev.1 @技術書典 9
2020/09/17	初版 rev.2 @技術書典 9

コードのスタイルについて

▼リスト 1　.prettierrc

```
1: {
2:   "tabWidth": 2,
3:   "singleQuote": true,
4:   "semi": false,
5:   "trailingComma": "all",
6:   "printWidth": 80
7: }
```

個人的な好みにより、この記法でコードをフォーマットしています。

ファイル名について

React 界隈ではファイル名にパスカルケースの採用が多いですが、個人的な好みによりファイル名は、大文字小文字入り交じるパスカルケースではなく、小文字をハイフンで区切るハイフンケース[*4]を採用しています[*5]。

npm / **yarn** について

Node.js 上で動くパッケージマネージャには npm と yarn があり、本書では基本的には yarn で書いています。一部では npm での説明も並記しています。

[*4] ケバブケースともいいます。というかケバブケースの方が一般的かもしれません。

[*5] 筆者の個人的な考えですが、Windows や Mac ではファイル名の大文字小文字を、特殊な設定をしなければ区別しないため、大文字小文字入り交じるファイル名は、望ましくないと考えています。

免責事項

- 本書の内容は、情報提供のみを目的としております。正確性には留意しておりますが、正確性を保証するものではありません。この本の記載内容に基づく一切の結果について、一切の責任を負いません。
- 会社名、商品名については、一般に各社の登録商標です。TM 表記等については記載しておりません。また、特定の会社、製品、案件について、不当に貶める意図はありません。
- サンプルコードを除き、本書の一部あるいは全部について、無断での複写・複製はお断りします。
- サンプルコードは特段の明記なき場合は AS-IS で利用を許諾します。
  - 一部のコードは MIT License です。

プロトタイピングサンプル（ブログ）のライセンス **(MIT License)**

```
 1: Copyright 2020 SASAKI Shunsuke
 2: https://github.com/erukiti/prototyping-sample
 3:
 4: Permission is hereby granted, free of charge, to any person obtaining
 5: a copy of this software and associated documentation files (the "Software"),
 6: to deal in the Software without restriction, including without limitation
 7: the rights to use, copy, modify, merge, publish, distribute, sublicense,
 8: and/or sell copies of the Software, and to permit persons to whom the
 9: Software is furnished to do so, subject to the following conditions:
10:
11: The above copyright notice and this permission notice shall be included
12: in all copies or substantial portions of the Software.
13:
14: THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
15: OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
16: FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
17: THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
18: LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
19: FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
20: DEALINGS IN THE SOFTWARE.
```

目次

第 1 章

プロトタイピング

序章でも述べた通り、プロトタイピングとは、手早く、実際に動く物を作って触りながらフィードバックを得て改良していくという手法です。

本章ではそれらプロトタイピングにまつわるコツやテクニックを紹介します。

1.1 原則

プロトタイピングに限りませんが、手早く開発するための原則をいくつか挙げておきます。

1.1.1 同時に複数のことをしない

同時に複数のことをしないようにしましょう。

- 新機能を開発する
- 新しいライブラリを試す
- 新しい書き方を試す
- 新しいツールを使う

こういったことを同時にすべきではありません。

少なくともプログラミングにおいて人間の脳はマルチタスクに適していません。擬似マルチタスクになってしまい、まず効率が落ちます[*1]。また、問題が生じたときに切り分け

[*1] そもそもプログラミングに限らないという話もあります。

しづらいということも効率を落とす要因です。

いかにして、同時にやることを一つに絞り込めるかが大切です。

1.1.2 設定

調査や設定は意外にエネルギーを消耗します。特にプロトタイプを作ることに集中したいモードに入ろうとしているときに、設定という全く別の方向性のものに脳のリソースを奪われるのは望ましくありません。

対処方法としては、前述の通り、設定とプロトタイピングという 2 つの事を同時にしない、分けるということです。業務時間内のプロトタイピングで、必要ならば設定のための時間を見積もって確保しておき、プロトタイピングとは別工数にすると良いでしょう。

余裕のあるときにあらかじめ、設定関連を片付けておくといいでしょう。これはプロトタイピングに使える武器を常日頃から準備しておくということです。

一番ありがちなミスは、設定なんてサクっと終わるだろうと思ってハマるケースです[*2]。

オススメのテクニックはタイマーの活用です。たとえば 10〜20 分くらいでタイマーをかけて作業を開始します。タイムアウトしたら、いったん諦める、方針を見直す、最悪なやり方でも採用する、自分の足りない知識を棚卸しする、気分転換するなど、といった見切りをつけるのです。

これは別に設定に限ったものではありません。何かしらのハマりそうな作業にはとても有効です。

ライブラリの調査

これも設定と同じですが、ライブラリ特有の解決策もあります。たとえば、CodeSandbox というウェブサービスがあります。これはテキストエディタと JavaScript が動くサンドボックスと呼ばれるもので、エディタで書いたコードがブラウザ上で動作するというものです。

このサンドボックスには npm もインストールでき、試したいライブラリと依存するものだけをインストールした最小限の環境を用意して、試したいコードを書いて動かすことで、ライブラリの調査ができます^[*3]。

調査で大切なポイントとして、なるべくミニマムに実験しましょう。

[*2] とてもよくあります……。

[*3] もちろん、ローカルで実験してもいいですが、CodeSandbox ならお手軽ですし、結果を簡単に共有できます。

よほどの理由が無い限りは、開発中プロダクトのコードベースと、ライブラリ調査用コードベースは分離すべきです[*4]。場合によっては社内の謎ライブラリやフレームワークに依存して、完全な分離はできないかもしれませんが、ライブラリの調査であれば大抵はサンドボックスを活用できるはずです。

1.1.3 ツールの整備をしておく

たとえば筆者は VSCode を愛用しています。2020 年時点でウェブ開発としては最善のエディタ（IDE）だと考えていますが、もちろん WebStorm や vim, Emacs などを愛する人もいるでしょう。

他にも、Linter（eslint）や書式の自動フォーマッタ（prettier）といったツールも活用しています。

- エディタ（IDE）で TypeScript の型チェックが走っていること
- エディタ（IDE）で補完ができること
- エディタ（IDE）でリファクタリングができること
- エディタ（IDE）で自動で Linter / フォーマッタ が走ること

プロトタイピングではこういった足回りの整備が重要です。これについてもこれまでに説明してきたことと同様に、同時にやらないように注意しましょう。

Linter やフォーマッタの重要性と非重要性

Linter や書式の自動フォーマッタの導入は必須です。もし入ってないプロジェクトがあれば、最初にこれらを導入できないか検討すべきです。もし導入できないようであれば、転職を検討すべきサインかもしれません。

tslint の開発が終了したので、2020 年の JavaScript/TypeScript 開発では eslint 一択です。もしかしたら 2022 年頃には Rome 一択という時代が来ているかもしれませんが……。

[*4] 分離できない場合は、自分がそのシステムの把握をしきれてないサインだと思った方がいいかもしれません。把握できてないことを認めつつ放置するのも、システムを分解して最小要素が何か追求するのも、それぞれ選択肢として考えてもいいでしょう。

書式の自動フォーマットは 2020 年時点では prettier 一択です。

さて linter やフォーマッタを導入するときに話題になることが、その設定ですが、細かい設定のあれこれ自体はほとんど意味がありません。

残念ながら細かい設定は、まずリーダビリティ（読みやすさ）やメンテナビリティ（開発しやすさ）には貢献しません。

設定をすることよりも、導入することで得られるメリットの方が遙かに大きいことと、細かい設定は大抵、自転車置き場の屋根議論と呼ばれるような「それは好みだね」となって結論がでません。

好みであれば、チームリーダーなりテックリードなり、誰かがエイヤで決めちゃって良いのです。

問題がなければデフォルトの推奨ルールを使えばいいでしょう。

方針としては、ゆるめな方向で設定するといいでしょう。無駄に linter の示すやり方に沿って、リーダビリティを落とすよりはよほどいいです。

1.2 プロトタイピングの始め方

プロトタイピングをするときに、どこを起点にするのか？[*5]は、その人が得意なやり方及び、状況に合わせたやり方に合わせるといいでしょう。

- デザインツールでデザインから作る（Figma など）
- HTML + CSS で画面モックから作る
- 動きやパーツから作る
- 関数から作る（TDD など）
- データから作る（SQL や TDD など）
- 型定義から作る

ある一方から作る必要はありません。二面作戦が有効なケースやそれが得意な人もいるでしょう。

[*5] プロトタイピングに限りません。新規にコードを書くときにどこから書くか？ という話です。

1.2.1 デザインから作る

デザインスキルを持っているひとは Figma[*6] などのツールを使うといいでしょう。頑張れば Figma で画面遷移なども含めて設計が可能です。場合によっては、Figma で作り上げたものをいじることでプロトタイピングが完了してる可能性もあります。

ただし、デザインツールのデータは、そのままではアプリケーションとして動くわけではないため、いつかは必ずアプリケーションに落とし込む必要があります。この落とし込むという工程に罠が潜んでいることもしばしばです。デザインだけでは気づかなかった問題、たとえば、バックエンドとの通信によって生じる問題もあり得るでしょう。

この作り方に関しては、筆者自身があまりデザインツール慣れしていないのもあり、本書では取り上げません。

1.2.2 画面モックから作る

HTML + CSS にある程度慣れているなら、画面モックを作るという選択肢があります。出来あがった HTML + CSS を分割してコンポーネントに落とし込んでいけば、段階的に動きを付けていけます。

HTML + CSS で作るときに最も役立つツールはウェブブラウザの持つ developer console です。表示している画面の DOM 構造の確認や、動的な書き換えなども行えるため、HTML + CSS をいじる際に重宝します。

誰かがデザインした資料や、デザインツールのデザイン画面があれば、それを HTML + CSS に落とし込むことになりますが、そういったものがないケースもあります。その場合、デザインツールに慣れていなければ、まずはペンと紙、あるいはタブレットなどを使い、手書きで雑に絵を描いてみましょう。

1.2.3 動き中心で作りはじめるなら

デザインを最低限のものにして画面遷移や動き中心で作り始めるというやり方もあります。

たとえば、最小限のボタンや `input` などを配置して、JavaScript で動きを付けます。

このやり方は最速で動く物に到達できるため、プロトタイピングを見せたい相手が許容

[*6] https://www.figma.com/

できるなら、デザインは最小限であることを伝えておいて、動きだけを見せることもできるでしょう。

ただ、どうしても完成品とのイメージの差は大きくなりがちなので、早めに、CSS の装飾などを付けるべきでしょう[*7]。

UI ライブラリを使っても問題の無い案件であれば、UI ライブラリを使えばどこかで見たことがあるような画面になってしまいますが、それなりの見た目をそこそこの労力で組み立てられます[*8]。

最小限の HTML で作るか、UI ライブラリを使うかはさておいて、動き中心で作る場合、具体的な工程としてはコンポーネントを作る事になるでしょう。

トップダウンが得意なひとは、最初に `TodoApp` のようなコンポーネントを作り、中では最低限の構造を表現できる HTML を作りはじめて、そこから分割をしていくといいでしょう。逆に必要なパーツがある程度見えている人はボトムアップでやってもいいでしょう。パーツを作って、組み合わせて `TodoApp` を作るのです。

厳密にどこから始めるべきということもないので、自分の手の付けやすいところから始めてかまいません。

1.2.4 型定義から始める

昨今のウェブフロントエンドでは以前と比べて TypeScript が当たり前になりました。そのため、簡単に型から作れるようになりました。

- React コンポーネント引数（props）の型定義
- 様々なデータ構造の型定義
- 関数の型定義

これらを駆使するのです。

特にやりとりするデータが分かっている場合は、そのデータを表現する型を書き始めるといいでしょう。

▼リスト 1.1　記事の著者

[*7] 人間はある程度デザインに縛られる生き物なので、CSS を全くつけてないようなイケてないデザインの UI はウケが悪い傾向があります。

[*8] UI ライブラリを使ってプロトタイピングをするときは、その UI ライブラリに日頃から慣れておく必要があります。また、チートシートなんかを手元に用意しておくと、とてもはかどります。

```
1: type Author = {
2:   id: number
3:   name: string
4: }
```

▼リスト 1.2　記事

```
1: type Article = {
2:   author: Author
3:   tags: string[]
4:   subject: string
5:   content: string
6: }
```

この TypeScript の型定義は記事の著者、記事を表現する型です。ここで重要なことは、この 2 つの型にそれぞれ `Author` や `Article` といった名前がついていて、その名前でアクセスできるということです。

最初から完璧である必要はありません。名前でアクセスできさえすれば、中身を必要に応じて、追加・削除・変更が可能ですし、それによって影響を受ける範囲は全部 TypeScript コンパイラが教えてくれます。

データの型があれば、コンポーネントの引数の型として渡すこともできます。コンポーネントに `id` `name` `tags` `subject` `content` を渡すのと、`author` `article` を渡すのであれば、後者の方が圧倒的にメンテナンスしやすいでしょう。

型を作ってから作成する型定義駆動に慣れると、適切な型さえ最初に定義すれば、それらをやりとりするだけで、開発ステップが軽量になっていきます。

React コンポーネントのプロトタイプを作る場合、DOM 構造が頻繁に変わりうるため、スナップショットテストを書いても恩恵が少ない、などという問題がありますが、型定義をしっかりしていると、そういったプロトタイピング中のコード変化に対応しやすいです。

ただし、ある程度モノが出来てきて、安定すべきコードであると判断したときには、スナップショットテストや画像回帰テストを検討すべきです[*9]。また、コンポーネント本体はともかく、そこから呼び出している関数などはユニットテストを書く方が望ましく、またユニットテストが書きやすい設計を心がける方が、メンテナンスしやすいコードになり

[*9] 本書では解説していませんが、Web フロントエンドでのテストでは、スナップショットテストや画像回帰テストと呼ばれるものが定番です。興味のある方は検索してみてください。

ます。

型もテストも、両方うまく使い分けるのが望ましいです。

1.2.5 組み合わせる

ここまでに紹介してきたテクニックは、組み合わせられるものばかりです。

たとえば「動き中心で作りはじめるなら」で説明したように、React コンポーネントを作るところから始めるのであれば、コンポーネント引数の型定義と平行で始めることができます。

型を駆使したプロトタイピングは、精密さと手早さを両立できるため、オススメできるやり方の 1 つです。型定義さえしっかりしてあれば、多少テキトーに直感でコードを書いても大体なんとかなるからです。足りないものはエラーとして現れてきますから。

イマドキは VSCode が優秀ですし、型定義さえしっかりしてあれば、大抵なんとかしてくれます。

プロトタイピングはゼロイチなのか？

プロトタイピングとはゼロイチな手法なのでしょうか？

大抵の場合は、新規の設計・実装になるはずですが、必ずしもゼロイチとは限りません。

たとえばある既存プロダクトに新しい機能を作るとき、プロトタイプを作って動作検証して本番コードを作る工程の方が、机上設計して実装するよりも早いのであれば、プロトタイピングの出番です。

ある程度の規模感だったり、プロトタイピングに慣れていれば、机上設計をしてから実装するよりも、開発工数を縮めることができる可能性は高まります。

まっさらな状態で作る場合は、考え方としてはゼロイチかもしれませんが、それでも参考にできるものは多いはずです。プログラミングにおいて、真の意味で無からの創造をするケースは頻出しません。過去の知識・経験、公式サイト、サンプルコード、GitHub のソース、ブログの記事、何かしら参考にしているはずです。

それはそれとしても、新規コードベースを作成するのが得意な人と、既存のコードを改修するのが得意な人がいます。前者であればプロトタイピングに既に慣れているか、少しの練習でプロトタイピングを物にできるかもしれません。

改修が得意な人が、無理をしてプロトタイピングをやるべきとはいいませんが、チャレンジしておくと、何かしら技術の幅が広がるのではないでしょうか。

1.2.6 レイアウトに慣れておく

何かしらのウェブアプリや GUI を開発する場合、大抵ネックとなるのがレイアウトです。

ヘッダ・サイドバー・フッターみたいな構造、スクロールしない領域とスクロールする領域、そういったものをデザインすることがとても多い割には、ウェブ技術のレイアウトは、CSS でもとても面倒で厄介です。そしてより悪いことに、画面の見た目のうちレイアウトは重要な要素です。

UI ライブラリのレイアウト機能を使う[*10]という割り切り方もありますが、HTML + CSS でプロトタイピングをするのであれば、いわゆる Flexbox（CSS Flexible Box Layout Module[*11]）や CSS Grid レイアウト[*12]を覚えておくことをオススメします。

Flexbox と Grid レイアウトはそれぞれ得手・不得手に違いがあります。Flexbox は一次元レイアウト、Grid レイアウトは二次元レイアウトに適しています。

Flexbox はとても現実的な手段です[*13]。多少癖があり面倒くさいところもありますがさっくり使えます。

Flexbox と Grid レイアウトをそれぞれ使い分けできると良いでしょう。時間があるうちに、どちらも試しておき、よくあるパターンをまとめたチートシートや、サンプル集を手元に用意しておくといいでしょう。

1.3 バックエンドのプロトタイピング

バックエンドのプロトタイピングのうち、型定義の話は既にしてあるので省略します。

[*10] UI ライブラリのレイアウト機能も、実際には Flexbox などを使って実現されています。

[*11] https://www.w3.org/TR/css-flexbox-1/

[*12] https://www.w3.org/TR/css-grid-1/

[*13] IE でも Polyfill を使えば Flexbox なら普通に使えますが、Grid レイアウトは一部機能に問題があるようです。

1.3.1 関数やクラスから作る

TDD（テスト駆動開発）をすると良いでしょう。テスト駆動開発は名前に「開発」とついていて、実装にもまたがっていますが、設計のための技法です。

たとえば関数を作るとして、その関数のインターフェース（入出力）はどうあるべきか？ を実際に、実コードとテストをそれぞれ書きながら決めるための技法だからです。

まさにプロトタイピングと同じです。実際に動くコードや、動くテストを書きながら、あるべきインターフェースについて決めるのです。

ビジネスロジックがある程度見えているような場合は特に TDD のような手法が有効です。入力データと、それに対応する出力データを定義すれば、ビジネスロジックが具体化できる、というような、ボトムアップから全体像を描ける人に向いています。

TDD では 100% を目指すこともできますが、完璧ではないがちゃんと動く物を手早く作るための、設計技法として使うのが望ましいでしょう。ここらへんは求めるもの、段階によって使い分けると開発力を大きく上げることができます。

1.3.2 データから作る

SQL 慣れしてる人は、こういうパターンのアプリケーションならこういうデータが必要だ、というデータモデリングから入るといいでしょう。

特に本書で紹介している Hasura GraphQL Engine であれば SQL さえ作ってしまえば、GraphQL API も自動で用意されるため、データ指向のプロトタイピングが行えます。GraphQL には、GraphiQL[*14] という、実際の GraphQL クエリを投げてその結果を受けとる GUI ツールがあり、Hasura Console にも組み込まれています。

1.4 容易に改修できるようにする

プロトタイピングは、手早く作ることと、フィードバックを受けて改修しなければいけないため、容易に改修できるようにしなければいけません。

[*14] Graph と QL の間に i があります

1.4.1 型はとても重要

型さえがっちり書いておけば、改修は容易になります。ちょっとやそっとの改修程度では壊れないようになります。

1.4.2 JSDoc を書く

不完全でもいいので JSDoc を書きましょう。最低限、その関数・コンポーネントの意図、引数の制約[*15]などを書いておくべきです。

JSDoc があれば、チームメンバーがコードを触るときに効率が段違いに改善されますが、一人でプロトタイプを作る場合でも、別の場所からそのコードを呼び出すときに、エディタの機能でドキュメントが表示されて便利です。

ローカルのみで使われてるものに JSDoc を付ける優先度は低いですが、色々なところから参照されているものなんかは優先的に JSDoc を付けるべきです。

1.4.3 作り捨てられる気持ちを持つ

1・2 日で作れるわ、くらいの気持ちで作り、維持しましょう。いつでも捨てられるという気持ちがある方が心の平穏を保てますし、なによりフィードバックをくれる人にも「すぐ作り直せるんで、根本的な指摘でも OK ですよ」とアピールしておかないと、プロトタイピングの意味がありません。

もし 1・2 日で作れる範囲を超えている場合は、プロトタイピング自体を軽く済むように仕様やスコープ（範囲）を調整した方がいいかもしれません。

一度書いたコードを消したくない、消す・書き換えることに、心の抵抗を感じる場合は、何かしらコード供養の方法を考えてもいいかもしれません。どこかにメモしておく、リポジトリにブランチを切ってコミットしておくなどです。

1.4.4 常時、リファクタリングをする

プロトタイプを作る工程では常にリファクタリングをしましょう。

リファクタリングとは、あるコードの外から見た振る舞いを変更せずに、中のコードだ

[*15] 型情報のようなものはどうせ TypeScript で型アノテーションを付けてるはずなので不要ですが、型で表現できない情報は自然言語で記述するのが望ましいでしょう。

けを書き換えて、コードをよりメンテナンスしやすい綺麗な形にすることです。

外から見た振る舞いとは、たとえば関数であれば、入力と出力です。副作用があればそれも振る舞いです。GUI コンポーネントであれば、人間が見た時の見た目や、ボタンを押したときの挙動などを指します。

なぜリファクタリングというテクニックが開発されたかというと、人は最初から完璧なモノを作れないため、最初から完璧を目指さない、途中でより良い形を模索するためです。

プロトタイピングで生み出されるコードは、何度も書き換えられるという性質上、リファクタリングを怠ると、いとも簡単に、誰も触りたくないようなコードになってしまいます[*16]。より良いプロトタイピングとリファクタリングは切っても切り離せないものです。

1.4.5 デザインドキュメントを書く

いわゆる設計書です。これも完全なものを書く必要はありません。簡単に Markdown[*17]で書いてリポジトリに `design.md` や `design_doc.md` あたりの名前で入れておくといいでしょう。

- 背景（なぜ作るのか？）
- ゴール
- 選択理由（なぜそれを選んだ？ なぜそれを選ばない？[*18]）
- 用語集
- 大まなか構造
- 大まかな設計思想

などをざっくり書いておくというものです。

図はあると役立つかもしれませんが、必須ではありませんし、殴り描きで十分です。

1.4.6 どこかの段階でユニットテストを書く

プロトタイピングは作り捨てる気持ちで手早く作るものですが、プロトタイプとして作ったものが最終的に生き残るということも往々にしてあります。

[*16] プロトタイピングに限らない話ですが、特にプロトタイピングだと特にそうなりやすいです。
[*17] Excel を頑張る必要はありません。エビデンスも必要ありません。
[*18] ソースコードでは表現が極めて難しいため、自然言語で記述する必要があります。

そこで大切になってくるのが、リファクタリングとユニットテストで、前者に関しては既に説明済みなので、ここではユニットテストの大切さを解説します。

TDD でやっていない場合、プロトタイプの序盤でユニットテストを書くのは無意味になりがちですが、プロトタイプもある程度固まってきたら、ユニットテストを書いていくべきでしょう。

まだフロントエンドでは、ユニットテストが普及しているとは言いがたいですが、たとえばバリデーションスキーマ、一部のカスタムフック関数、ユーティリティ関数などといったもののユニットテストは書きやすいはずです。ある程度仕様が定まりつつあるコンポーネントのスナップショットテストや画像回帰テストなんかも有用でしょう。

1.4.7 誰でもアクセスできるようにデプロイしておく

プロトタイピングは、作ったものを実際に触ってもらわないと意味がありません。

ついつい「リポジトリに入れてあるからビルドして動かして試してみて！」といいたくなる気持ちはありますが、エンジニアですら、それを面倒くさがる人はいます。

そのためプロトタイピング中は開発・試験用の環境にデプロイするのが望ましいでしょう。コミットしたら自動で CI/CD が走ってデプロイまでしてくれれば理想形です。

ただし、本書ではページの都合上デプロイに関しては省略しています。

次善策としては、ngrok[*19] などの、ローカルサーバーをグローバルに公開できるサービスがあります。ただし、なぜか重いというような謎のトラブルがあったり、ローカルサーバーを起動しっぱなしにするのは問題があることも多いでしょう。

1.5 経験の積み方

これもプロトタイピングに固有の話ではなく、プログラミング全般に通じるものですが、プロトタイピングにおいても重要なので、ここに書いておきます。

1.5.1 最初から最後までの工程をやり遂げる

プロジェクトの作成から、実運用なりまでを一度やり遂げておくといいでしょう。序盤から終盤までに何をすればいいのかを把握できますし、序盤までもっていくために、最初のうちにしておいた方がいいことを肌感覚として掴めるようになります。

[*19] https://ngrok.com/

1.5.2 時間的余裕のあるうちに、技術を練習しておく

プロトタイピングが一週間でできあがったとしても、実際にはそれはその一週間だけで、出来あがったものではありません。常日頃から、技術について習熟しているからこそできるのです。

この本でも取り上げている、React, Next, Hasura, GraphQL, SQL などのように、プロトタイピングに使えるフレームワーク、ライブラリ、仕組み、概念など、常日頃から習熟しておきましょう。フロントエンドのコードも注意深く書かなければ、容易に速度が遅くなりがちです。どういうときに遅くなるのか？ どうすれば速度が改善されるのか？ という経験値を詰んでおけば、最初からアンチパターンを避けてコーディングできるようになるはずです。

暇のあるときにプロトタイピングやリファクタリング、TDD などの練習をしておくのもいいでしょう。

1.5.3 ふりかえりを活用する

ふりかえりといっても、KPT だの PDCA だのをやる必要性は全くありません。ふりかえりの達人になることが目的ではありませんし「同時に複数のことをやらない」の原則を思い出しましょう。

その日やったことを自然言語でメモに書き出す、図にする、得た資料リンク集（公式サイトや参考にしたブログなど）を作る、感想を書く、そんな程度で十分です。もちろん、これら全部をやる必要すらありません。

「今日はうまくいった。やったー！」だけでも十分です。

ふりかえりをより効果的にやりたい場合、Fun Done Learn というやり方もあります。

- Fun は楽しかったこと
- Done はやりとげたこと
- Learn は学びだったこと

この 3 つを意識して、メモを残すだけでも効果はばつぐんです。今日のプロトタイピングや技術調査は 3 つの項目のどれに偏っていたか？ まんべんなく達成感があったか？ などをふりかえることができます。

第 2 章

要素技術

この章では、Next.js, Hasura GraphQL Engine などなど、プロトタイピングに必要な説明をしていきます。

2.1 Next.js

Next.js（以後 Next）は、Vercel が開発する React フレームワークです。

特徴は React のコンパイルなど面倒を引き受けてくれる軽量なフレームワークであることです。この性質はプロトタイピングにとても向いています。

Vue.js 向けのフレームワークである Nuxt は元々 Next にインスパイアされて生み出されたものです。こちらも日本ではとても人気がありますが、方向性が全く違っていて、Nuxt は手厚いフレームワークです。Next はフレームワークと呼ぶには素っ気ないですが、Nuxt はまさにフレームワークと呼べるものです。

Next は日本ではあまり利用事例を聞かないかもしれませんが、世界的に見るとシェアは決して低くありません。少なくとも Nuxt の倍以上のシェアをもちます[*1]。

2.1.1 Next の主な機能

もう少し詳しく説明すると、Next は Node.js 上で動くサーバーアプリです。

[*1] The State of JavaScript 2019 によると、Next のシェア率は 24% で、Nuxt は 11% だそうです。React が 80% で、Vue が 46% と比べると、React ほどの、世界的なシェアはまだ無いですが、関心を持ってる人は多くいるようです。

- React コンパイルの面倒な部分を引き受けてくれる
- フロントエンドとバックエンドを透過的に扱える
- 必要なファイルを HTTP などでサービスする
- サーバーサイドレンダリングができる
- 静的サイト生成ができる
- AMP 生成ができる
- page ルーティング機能を持つ
- ほか、プラグインによる拡張ができる

などの機能を持ちます。

ビルド（**React** コンパイル）

TypeScript や React 他をプロダクション向けにビルドしようとすると Webpack の設定などが大変です。Next はそこをカバーしてくれます。

フロントエンドとバックエンドが透過的に扱える

Next はユニバーサルフレームワークです。フロントエンドとバックエンドでコードの共通化が可能です。

プロトタイピングをするときに、Firebase などの BaaS を前提としてフロントエンドのみで行う事例も多いでしょう。ただ、それではどうしてもできることが限定されてしまいます。Next ならバックエンドにも踏み込んだプロトタイピングが可能です。

create-react-app

カジュアルに React を使う手段の 1 つに create-react-app というツールがあります。これらは React を軽く試したい場合に有用で、筆者も度々これを使って記事を書いたりしてきましたが、ビルドに使う Webpack の設定が実質、不可能という致命的な欠点があります。

幾つかのライブラリは Webpack の設定をいじることを前提としているものがありますし、本格的なプロダクト開発では Webpack を触る事も多いです。

Next なら簡単に Webpack の設定を変更できます。

2.1.2 Next を使う

Next を使うには create-next-app という Next のプロジェクトを作成してくれるツールを使えば簡単にセットアップできます。

```
# yarn の場合
$ yarn create next-app <アプリケーション名>

# npm/npx の場合
$ npx create-next-app <アプリケーション名>
```

このコマンドを叩くことにより、アプリケーション名のディレクトリが作られ、その中に必要なファイルやパッケージがインストールされます。

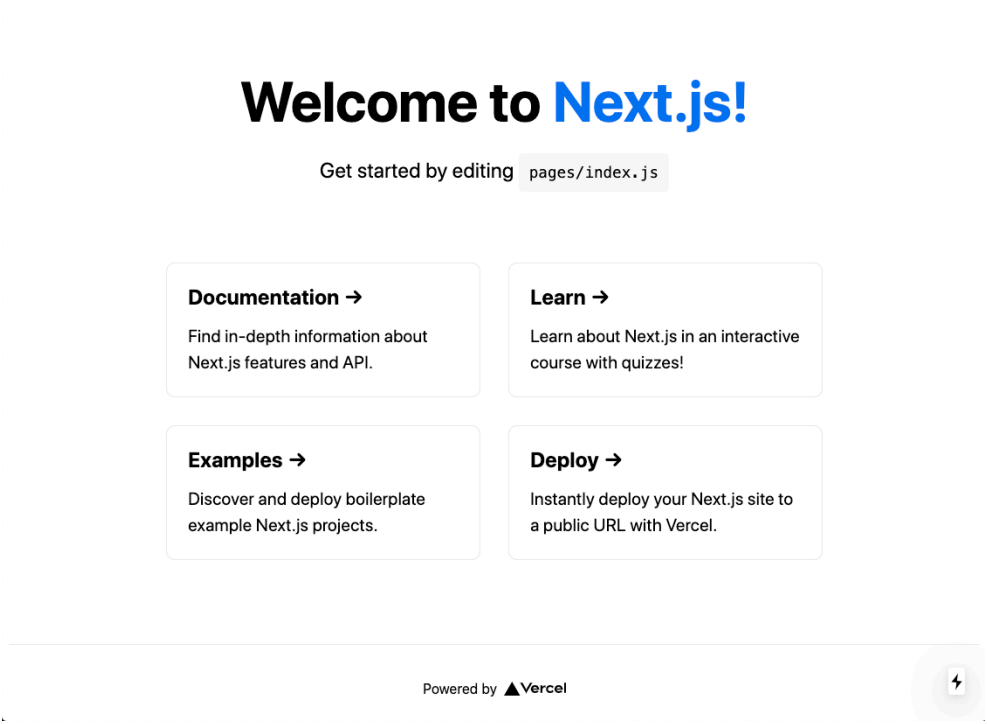
```
# yarn で sample-app というアプリケーションを作る場合
$ yarn create next-app sample-app
$ cd sample-app
```

2.1.3 起動してみよう

sample-app ディレクトリで yarn dev もしくは npm run dev というコマンドで、開発用サーバーが立ち上がります。

```
1: $ yarn dev
2: ...
3: ready - started server on http://localhost:3000
4: event - compiled successfully
5: event - build page: /
6: wait  - compiling...
7: event - compiled successfully
```

http://localhost:3000 にアクセスすると、Next.js のデフォルトページが表示されます。



▲図 2.1　Next.js デフォルト画面

これで Next.js のインストールはできました。いま表示されている画面は `pages/index.js` がソースコードです。

▼リスト 2.2　pages/index.js

```
 1: import Head from 'next/head'
 2: import styles from '../styles/Home.module.css'
 3:
 4: export default function Home() {
 5:   return (
 6:     <div className={styles.container}>
 7:       <Head>
 8:         <title>Create Next App</title>
 9:         <link rel="icon" href="/favicon.ico" />
10:       </Head>
11:
```

```
12:       <main className={styles.main}>
13: 省略
14:       </main>
15:
16:       <footer className={styles.footer}>
17: 省略
18:       </footer>
19:     </div>
20:   )
21: }
```

やたら長いので抜粋です。

<Head>....</Head> は HTML ヘッダーを示すためのものです。よく React アプリでは HTML ヘッダを記述するために react-helmet[*2] というパッケージが使われますが、Next の標準機能で、HTML ヘッダーを記述できます。ここでは Head によって、タイトルと、ファビコンが設定されています。

import styles from '../styles/Home.module.css' では CSS Modules を読み込んでいます。

たとえば styles.container や styles.main などは、

▼リスト 2.3　styles/Home.module.css

```
 1: .container {
 2:   min-height: 100vh;
 3:   padding: 0 0.5rem;
 4:   display: flex;
 5:   flex-direction: column;
 6:   justify-content: center;
 7:   align-items: center;
 8: }
 9:
10: .main {
11:   padding: 5rem 0;
12:   flex: 1;
13:   display: flex;
14:   flex-direction: column;
15:   justify-content: center;
16:   align-items: center;
17: }
```

として定義されています。

[*2] https://github.com/nfl/react-helmet

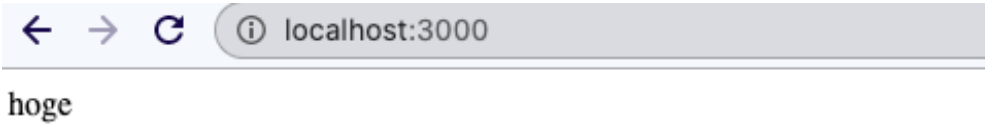
2.1.4 編集してみよう

pages/index.js を編集してみましょう。

▼リスト 2.4 pages/index.js

```
1: export default function Home() {
2:   return <div>hoge</div>
3: }
```

create-next-app の厚意を無に帰すような変更ですがばっさり削ってしまいましょう。



▲図 2.2 hoge だけ表示される

開発サーバーを起動している間は、ファイルを編集すれば、その更新に応じて画面も切り替わります。

2.1.5 Next で覚えておくこと

https://nextjs.org/docs/getting-started の公式ドキュメントを一通り読んでおくと良いでしょう。さほどドキュメント量があるわけでもないので、片っ端から読むとしてもそんなに大変ではないはずです。

とは言え、本書で全部を解説するには紙面の限りもあるので、簡潔に説明していきます。

Pages

https://nextjs.org/docs/basic-features/pages

Next のプロジェクトディレクトリで、pages/ もしくは src/pages/ 以下に配置したファイルは、ページルーティングが行われます。拡張子は、.js .jsx .ts .tsx などが対応しています。

第 3 章

ローカルで開発しよう

ローカルで開発をする場合、Hasura GraphQL Engine と PostgreSQL は、Docker（docker-compose）で動かすことをオススメします。

- Next.js をローカルで動かす[*1]
- Hasura GraphQL Engine と PostgreSQL を Docker で動かす

この章は **2020** 年 **9** 月時点のものなので、利用してるライブラリ・フレームワークのバージョンが上がることで、セットアップ手順が変わる可能性があります。なるべくその時の公式ドキュメントを参考にしてください。

3.1 Next

https://nextjs.org/docs/api-reference/create-next-app

Next は create-next-app でインストールします。この章では `yarn` で説明しますが、`npm` ユーザーの人は適宜読み替えてください。

```
$ yarn create next-app sample-app
$ cd sample-app
$ yarn add -D typescript @types/node @types/react @types/react-dom
```

[*1] Mac や Linux ならローカルで問題なく動くことがほとんどですが、Windows の場合は、なるべく WSL2 を使ってください。

ここでは、TypeScript を手動でインストール[*2]しています。
TypeScript の設定を tsconfig.json に書きます。

```
$ $EDITOR tsconfig.json
```

▼リスト 3.1　tsconfig.json

```
 1: {
 2:   "compilerOptions": {
 3:     "baseUrl": ".",
 4:     "paths": {
 5:       "@/*": ["src/*"]
 6:     },
 7:     "target": "es2020",
 8:     "skipLibCheck": true,
 9:     "strict": true,
10:     "noEmit": true,
11:     "esModuleInterop": true,
12:     "module": "esnext",
13:     "moduleResolution": "node",
14:     "resolveJsonModule": true,
15:     "isolatedModules": true,
16:     "jsx": "preserve",
17:     "lib": ["dom", "dom.iterable", "esnext"],
18:     "allowJs": true,
19:     "forceConsistentCasingInFileNames": true
20:   },
21:   "include": ["next-env.d.ts", "**/*.ts", "**/*.tsx"],
22:   "exclude": ["node_modules"]
23: }
```

compilerOptions に　@/ で src/ をアクセスできるようにエイリアスの設定を追加しておきます。

```
$ rm -rf pages/ styles/
$ mkdir -p src/pages
```

[*2] yarn create next-app sample-app --examples --with-typescript で、TypeScript インストール済みのプロジェクトが生成されますが、不要なサンプルとなるものが多くてどうせ全部消すため、手動で入れています。Next.js + TypeScript がよく分からないときに参考にするという目的な一度は動かしてみてもいいかもしれません。

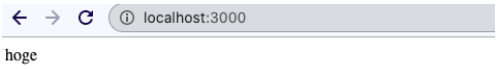
pages/ は、JavaScript で記述されていることと、pages/ という非標準的なディレクトリ構成ではなく src/pages/ に配置したいこと styles/ にあるスタイルシートも不要なので削除します。

一応動作確認と、作成の起点とするために、src/pages/index.tsx を作成します。

▼リスト 3.2　src/pages/index.tsx

```
1: import { NextPage } from 'next'
2:
3: const Index: NextPage = () => {
4:   return <div>hoge</div>
5: }
6:
7: export default Index
```

NextPage は、Next の pages/ で使うコンポーネントの型です。React.FC をベースにしつつ、Next 固有の拡張をしているものです。



▲図 3.1　hoge だけ表示される

3.2　jest

https://jestjs.io/docs/ja/getting-started
jest は今もっとも使い勝手の良いテストフレームワーク（テストランナー）です。

```
$ yarn add jest @types/jest ts-jest -D
```

3.3　prettier, eslint...

https://prettier.io/ https://eslint.org/
prettier と eslint は併用して使うものなので、ここでまとめて設定します。

```
$ yarn add -D eslint eslint-config-prettier eslint-plugin-jest \
  eslint-plugin-prettier eslint-plugin-react prettier \
  @typescript-eslint/eslint-plugin @typescript-eslint/parser \
  eslint-plugin-react-hooks
```

まずは prettier の設定ファイルである .prettierrc を書きます。

```
$ $EDITOR .prettierrc
```

▼リスト 3.3　.prettierrc

```
1: {
2:   "tabWidth": 2,
3:   "singleQuote": true,
4:   "semi": false,
5:   "trailingComma": "all",
6:   "printWidth": 80
7: }
```

これは本書で採用している書式です。

もちろん、ご自身の好みに応じて自由に設定していいでしょう。どんな設定だとしても機械的に自動でフォーマットさせるので、人間が書式について悩むことから開放されます。

次に、eslint の設定である .eslintrc.json を書きます。

```
$ $EDITOR .eslintrc.json
```

▼リスト 3.4　.eslintrc.json

```
 1: {
 2:   "root": true,
 3:   "plugins": ["jest", "react"],
 4:   "env": {
 5:     "es2020": true,
 6:     "jest/globals": true
 7:   },
 8:   "extends": [
 9:     "eslint:recommended",
10:     "plugin:prettier/recommended",
```

```
11:     "plugin:@typescript-eslint/eslint-recommended",
12:     "plugin:@typescript-eslint/recommended",
13:     "plugin:react-hooks/recommended"
14:   ],
15:   "rules": {
16:     "no-mixed-operators": "error",
17:     "no-console": "off",
18:     "no-undef": "off",
19:     "react/jsx-uses-vars": 1,
20:     "@typescript-eslint/explicit-function-return-type": "off",
21:     "@typescript-eslint/explicit-module-boundary-types": "off",
22:     "@typescript-eslint/no-non-null-assertion": "off"
23:   },
24:   "parserOptions": {
25:     "project": "./tsconfig.json"
26:   },
27:   "parser": "@typescript-eslint/parser"
28: }
```

少々複雑です。eslint を jest / React(React Hooks) / prettier / typescript とそれぞれに対応させるための設定です。

ここまでの設定で eslint / prettier の設定が完了しました[*3]。

3.4 Hasura GraphQL Engine

`wget https://raw.githubusercontent.com/hasura/graphql-engine/stable/install-manifests/docker-compose/docker-compose.yaml` コマンドを実行すると、Hasura GprahQL Engine と PostgreSQL を動かすための `docker-compose.yaml` ファイルを取得できますが、一部書き換えをします。

```
$ $EDITOR docker-compose.yaml
```

```
1:   environment:
2:     HASURA_GRAPHQL_ENABLE_CONSOLE: "false" # set to "false" to disable console
```

これにより、Hasura GraphQL Engine が起動しても、自動でコンソールが立ち上がら

[*3] コマンドラインで eslint を動かしてもいいですが、VSCode の eslint 拡張などを入れておくと便利です。

ないようになります。これは後ほど説明する、自動マイグレーションのために必要なものです。

設定が完了したので起動をします。Docker でサーバーを動かすとき、バックグラウンドで動かすデーモンモード[*4]という起動方法があります[*5]。

```
# 別ターミナルで
$ docker-compose up

# 同じターミナルならデーモンモードで起動
$ docker-compose up -d
```

3.4.1 自動マイグレーションの準備

自動マイグレーションを行います。

```
$ yarn add -D hasura-cli
$ yarn hasura init hasura
```

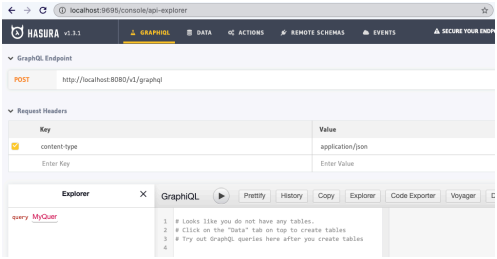
これで、マイグレーション情報が保存されるディレクトリが作成されました。

[*4] UNIX 系 OS には元々、デーモンというバックグラウンドで動く便利な仕組み・概念があります。ちなみに、綴りは demon ではなく daemon です。

[*5] 筆者はデーモンとして起動するよりも、別ターミナルを開いてフォアグラウンドで起動する方が好みです。-d を付けてデーモンモードで起動すると、起動したことを忘れてしまったままターミナルを閉じて、どこのディレクトリの docker-compose を立ち上げたのかわからなくなることがあるためです……。

```
# 別ターミナルでコンソールを起動する
$ yarn hasura console --project hasura
```

さて、Hasura GraphQL Engine がちゃんと動いてるか確認しましょう。



▲図 3.2　Hasura コンソール

Hasura コンソールの起動を確認ができれば、ここでの作業は完了です。データのセットアップや実際のマイグレーションについては、次の章で解説します。

3.4.2　コード生成

GraphQL コード生成では、プラグインを色々つかうこともあって、インストールするパッケージは多くなってしまいます。

```
$ yarn add @apollo/react-hooks @apollo/client graphql
$ yarn add -D @graphql-codegen/cli @graphql-codegen/typescript \
 @graphql-codegen/typescript-operations \
 @graphql-codegen/typescript-react-apollo
```

つぎに設定ファイルを作成します。GraphQL のスキーマ定義は、サーバー側の定義とクライアント側の定義に分かれています。

```
$ $EDITOR codegen.yml
```

▼リスト 3.6　codegen.yml

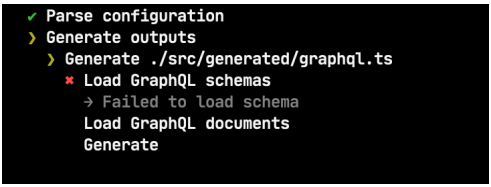
```
schema: http://localhost:8080/v1/graphql/
documents:
  - ./src/**/*.graphql
overwrite: true
generates:
  ./src/generated/graphql.ts:
    plugins:
      - typescript
      - typescript-operations
      - typescript-react-apollo
    config:
      skipTypename: false
      withHooks: true
      withHOC: false
      withComponent: false
```

schema は、サーバー側のスキーマ定義であり、サーバー側ができること全てを表現したものです。documents は、クライアント側のスキーマ定義であり、実際にクライアント側が欲しい情報を指定したものです。

documents は、.graphql 以外にも .ts などを指定することも可能ですが、筆者の考えではコード生成をする限りは .graphql ファイルにスキーマ定義をすべきだというものなので、./src/**/*.graphql という設定にしています。

さて、現時点ではまだ Hasura コンソールを使ってサーバー側のスキーマ定義をしていませんし、クライアント側の src/**/*.graphql も共に存在しないため、コード生成コマンドを実行してもエラーになります。

```
$ yarn gql-gen
```



▲図 3.3　コード生成に失敗

あと、現時点ではまだ生成されていませんが、src/generated/graphql.ts が自動生成されるはずなので、eslint / prettier の対象外に指定します。

```
$ $EDITOR .eslintignore .prettierignore
```

▼リスト 3.7　.eslintignore

```
1: src/generated
```

ここまで設定すれば、あとは Hasura コンソールで RDB をセットアップして src/ 下にクライアント側の GraphQL クエリを作成するだけです。この章は汎用的に使える手順として記述してるため、いったんここまでにします。

3.4.3 Apollo Client 初期化

https://www.apollographql.com/docs/react/

Apollo Client は、最もメジャーな GraphQL クライアントです。

新しい API 仕様を採用している @apollo/client が推奨されていますが、少し困ったことに Apollo Client には過去使われていた apollo-boost というのもあって、そちらの方がまだまだ情報が多かったり、API 仕様が @apollo/client になって複雑になってしまったという問題があります。

とは言え、新規で書くならなるべく新しい API 仕様を採用する方が後々困ったことになりづらいため、今回は @apollo/client を使います。

▼リスト 3.8　src/pages/_app.tsx

```
 1: import { AppProps } from 'next/app'
 2: import { ApolloClient, InMemoryCache, HttpLink } from '@apollo/client'
 3: import { ApolloProvider } from '@apollo/react-hooks'
 4:
 5: import 'minireset.css'
 6:
 7: const createApolloClient = () => {
 8:   return new ApolloClient({
 9:     link: new HttpLink({
10:       uri: 'http://localhost:8080/v1/graphql',
11:     }),
12:     cache: new InMemoryCache(),
13:   })
14: }
15:
16: const MyApp = ({ Component, pageProps }: AppProps) => {
17:   const client = createApolloClient()
```

```
18:
19:   return (
20:     <ApolloProvider client={client}>
21:       <Component {...pageProps} />
22:     </ApolloProvider>
23:   )
24: }
25:
26: export default MyApp
```

この初期化のおまじないが必要です。

また minireset.css という、リセット CSS を当てています。

```
$ yarn add minireset.css
```

これで大体の準備は整いました。

3.5 README.md

README.md も create-next-app が生成したものであるため、必要に応じて、プロジェクトに合わせた README に書き換えてしまいましょう。

3.6 git にコミットしましょう

create-next-app で作成されたディレクトリでは最初から git が設定されている状態です。そこで、ここまでの成果を git にコミットしましょう。

なんなら、自分のプロトタイピング用に、このコミットした git を GitHub の private リポジトリにでも格納しておくといいかもしれません。そこから clone してプロトタイピングをすればセットアップをすっ飛ばせます。

ただしその場合、パッケージのアップデートなどを考慮しなければいけません。プロジェクトのパッケージをアップデートしてくれるツールを活用するといいでしょう[*6]。

[*6] コマンドラインで手動でやるなら npm-check-updates を使うといいかもしれません。定期的にアップデートをしてくれる renovate をセットアップ出来れば良いかもしれませんが、多少面倒かもしれません。

```
$ $EDITOR .gitignore
```

.gitignore の末尾に .env と src/generated を追加しておきます。

▼リスト 3.9　.gitignore

```
1: .env
2: src/generated
```

```
$ git add .
$ git commit -m 'setup done'
```

https://github.com/erukiti/prototyping-sample/tree/4cf6b0733421cb3d0dc14a3cb5fb63141b35b063

```
commit 4cf6b0733421cb3d0dc14a3cb5fb63141b35b063
Author: SASAKI Shunsuke <erukiti@gmail.com>
Date:   Tue Sep 1 02:10:34 2020 +0900

    setup done

commit ecc5b526d8ed9d45c9bbb3479e11adfaa5d12460
Author: SASAKI Shunsuke <erukiti@gmail.com>
Date:   Tue Sep 1 01:28:03 2020 +0900

    Initial commit from Create Next App
```

https://github.com/erukiti/prototyping-sample/tree/4cf6b0733421cb3d0dc14a3cb5fb63141b35b063 は、筆者のサンプルリポジトリの URL のコミットハッシュに対応したものです。このコミットハッシュに対応したソースコードを一通り読めるので、本書の補完用にご覧ください。

次の章以後でも都度、コミットログと GitHub の URL を貼り付けています。

さて、まだ DB にデータを作成してないため、自動マイグレーションやコード生成を試せませんが、次の章で詳しく説明いたします。

第 4 章

ブログサービスをプロトタイピングする（閲覧画面を作る）

この章からは、実際にプロトタイピングの工程を、生々しくお伝えしていきます。

プロトタイピングの題材として、ウェブアプリとしてブログサービスを作ることにします。

シチュエーションとしては、貴方はあるチームで、ブログサービスの新規開発を行うことになったとします。ターゲット層は特に定めていないものとします。

チームでは、企画者やデザイナーが、いまからブログサービスを立ち上げるならどういう仕組みやデザインにすればいいか、机上の空論を並べ立てようとしていますが、貴方は「このままだといつまでも終わらないプロジェクトになってしまう」という予感に突き動かされたので「とりあえずプロトタイプを作るから、それを元に議論を始めよう」と提案します。

まだメンバーの稼働状況は、全員がスタートしているわけでもない状況なので、少しの猶予があるとしても、プロトタイプを手早く作れなければ、机上での議論をひたすら繰り返すことになり、いわゆるサンクコストもかさんで、どんどん沼に突っ込んでいくことになりかねません。

もちろん、これはフィクション[*1]です。

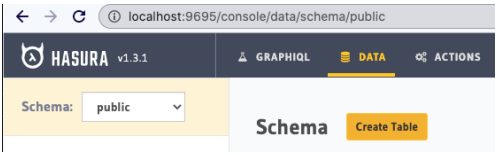
この章では、閲覧画面をプロトタイピングします。

[*1] ここ以後の章では、幾つかの機能やデザインなどを否定している場面もありますが、実際のところそれを否定することが正しいかどうかは分かりません。単なる事例として紹介しただけなので、別の結論になることもあるでしょう。著者もそれを否定する意図で書いている訳ではありません。

まずは、第 3 章「ローカルで開発しよう」を元にプロジェクトをセットアップして「.git にコミットしましょう」までを進めてください[*2]。そこから始めます。

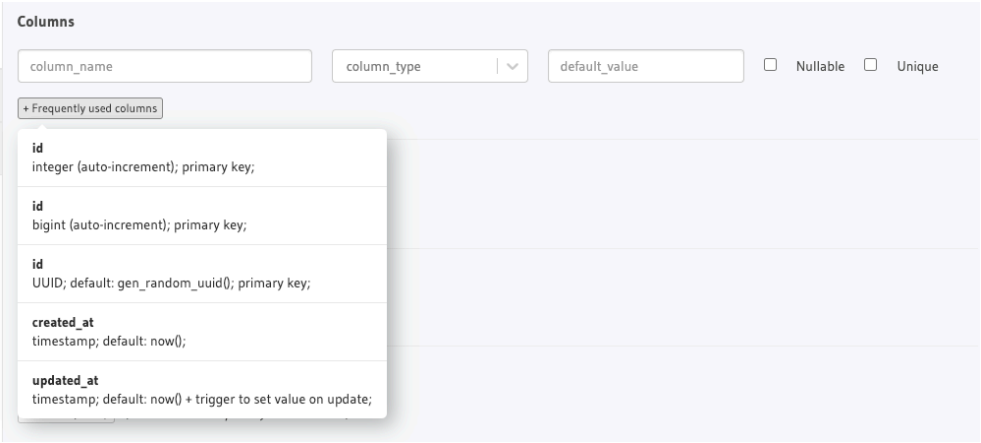
4.1 テーブル作成

Hasura コンソールの DATA タブからテーブルを作成します。



▲図 4.1　テーブル作成

まずユーザーテーブルを作成しましょう。カラム作成時には便利な機能があります。`+ Frequently used columns` というボタンです。よく使われるプライマリ ID や `created_at` や `updated_at` カラムなどを簡単に追加可能です。



▲図 4.2　Frequentry used columns

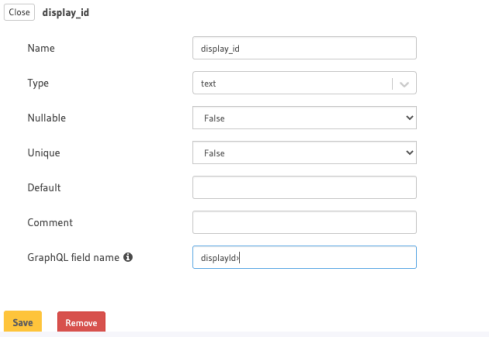
[*2] https://github.com/erukiti/prototyping-sample/tree/4cf6b0733421cb3d0dc14a3cb5fb63141b35b063

これを使い、id UUID; を追加します。

カラム名	型	初期値	Nullable	Unique
id	UUID	gen_random_uuid()		
display_id	Text			Unique
display_name	Text			
created_at	Timestamp	now()		
updated_at	Timestamp	now()		

ここまで設定したら一番下にある Add Table ボタンをクリックします。少し時間がかかりますが、必ず、完全に登録されるまでお待ちください[*3]。

作成後画面で `display_id` の Edit をクリックします。



▲図 4.3　display_id 編集画面

この一番下にある `GraphQL field name` に `displayId` を追加して、Save ボタンを押します。

`GraphQL field name` は、DB のカラムはスネークケースにしつつも、GraphQL の名前をキャメルケースにするために設定します[*4]。この設定は必ずしもやらないといけないものではありません。設定しない場合は GraphQL のカラム名に `display_id` というスネークケースが登場してしまうだけです。API から取得したパラメータがスネークケースの場合、JavaScript のコードでスネークケースを扱わないといけなくなります。

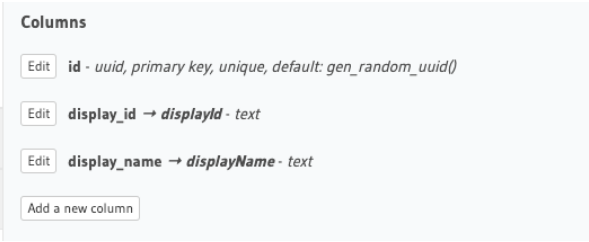
[*3] バージョンによるかもしれませんが、登録が完了するまえにボタンを連打したりすると、データが壊れたり、謎の状態になることもあるのでご注意を！

[*4] 正直なところ、この程度のことは全自動でやるオプション設定とか欲しいところです。

1. 我慢する
2. カラム名をキャメルケースに変換するユーティリティでも使う
3. あらかじめ GraphQL field name を設定しておいて最初からキャメルケースにしておく

この問題は DB では一般的にキャメルケースを使わずスネークケースを使う、JS では一般的にキャメルケースを使うという、境界面が衝突したからこそ生じる問題です。どこでそれを吸収するか？ という選択で、今回は 3 番を採用しています。

同様に display_name や created_at updated_at にも GraphQL field name を付与しましょう。



▲図 4.4　変更後

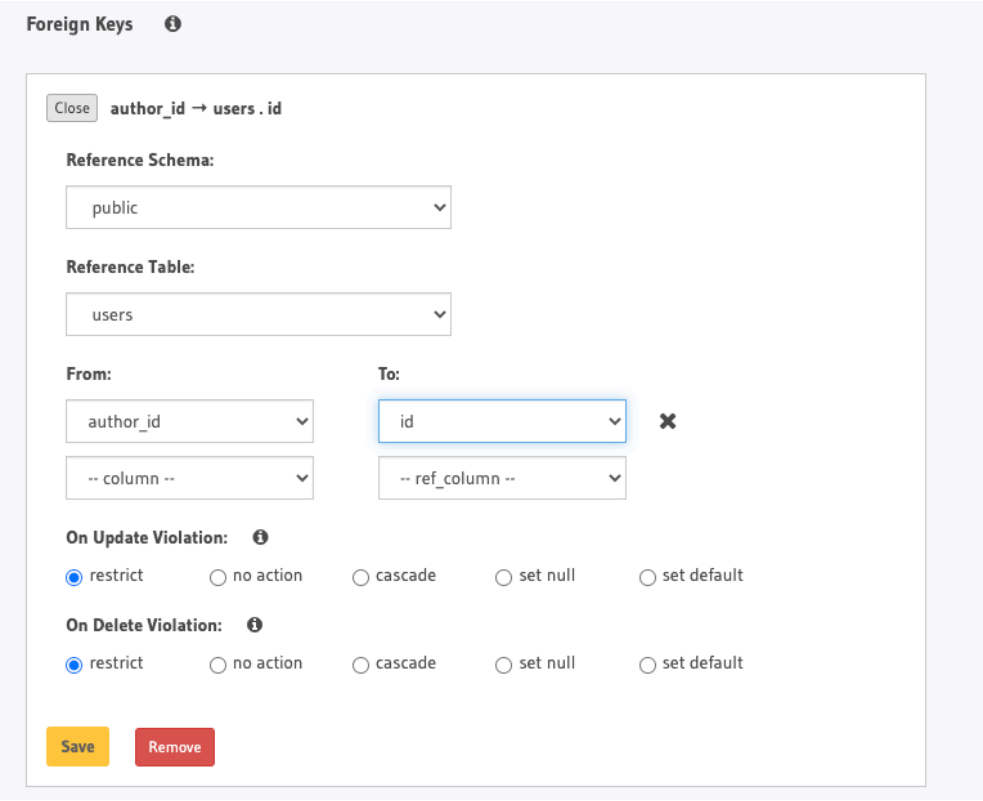
完了後はこのようになっているはずです。

次は、記事のための articles テーブルを作成しましょう。画面上部の DATA タブをクリックしてから Create Table ボタンを押します。

id と created_at と updated_at は、それぞれ Frequently used columns ボタンから追加します。

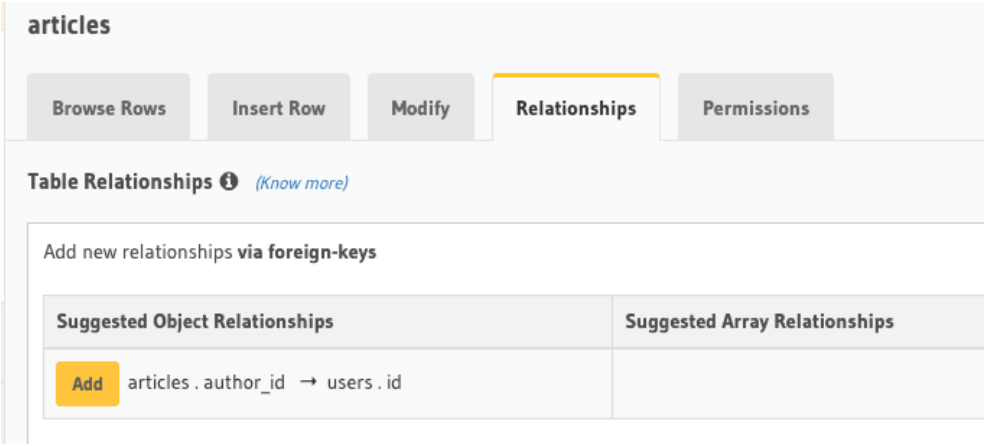
カラム名	型	初期値	Nullable	Unique
id	UUID	gen_random_uuid()		
subject	Text			
content	Text			
author_id	UUID			
created_at	Timestamp	now()		
updated_at	Timestamp	now()		
published_at	Timestamp		Nullable	

Foreign Keys を登録します。
Reference Table に users を設定し From に author_id を、To に id を設定します。



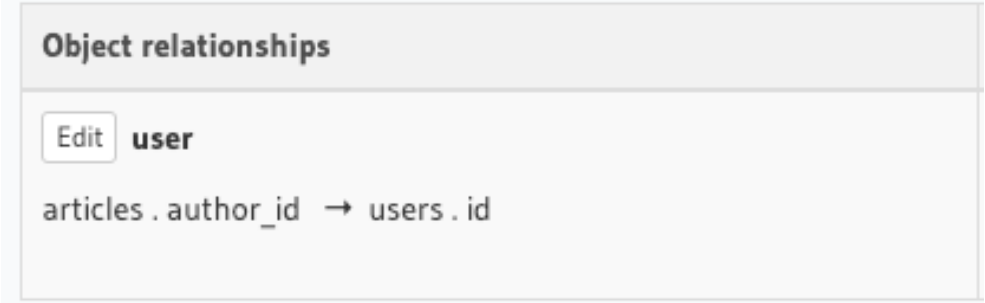
▲図 4.5　Foreign Keys

Save し、作成後に、Relationships タブをクリックして Suggested Object Relationships の Add ボタンを押します。



▲図 4.6　Relationships

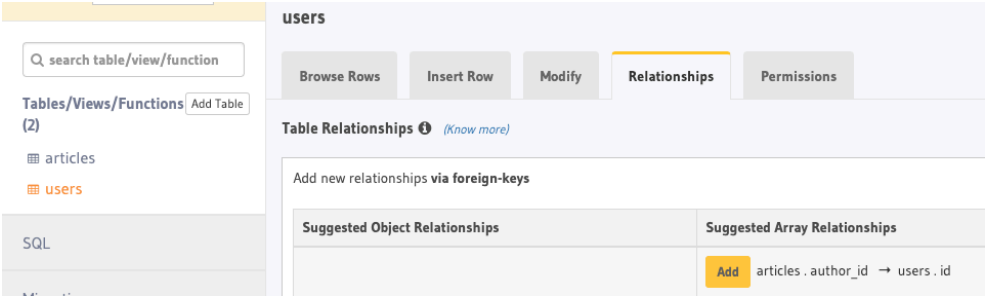
これは、Foreign Keys を元に、GraphQL 的に結合できる条件を設定できるものです。



▲図 4.7　設定後

設定するとこのようになります[*5]。
同様に、users テーブルのリレーションも張ります。

[*5] これも自動でやってくれればありがたいところなのですが……。



▲図 4.8　users テーブルのリレーション

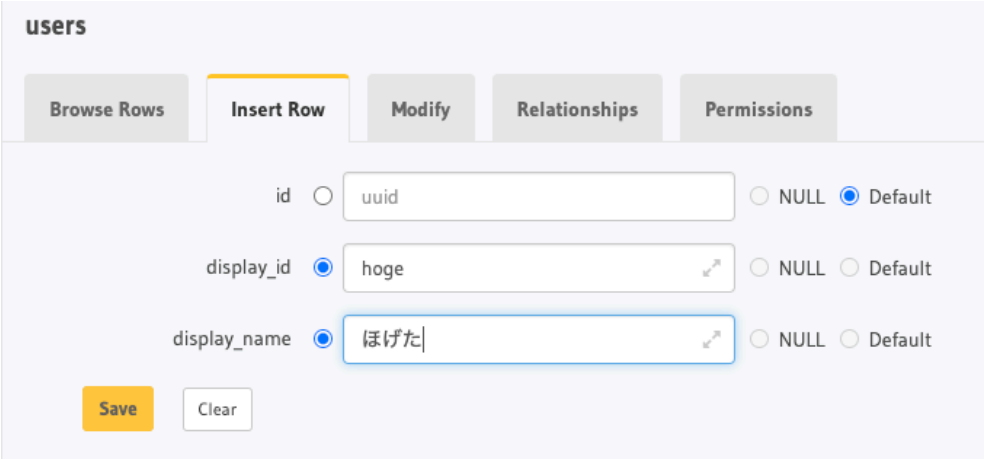
さて、articles から users へのリレーションは、Object relationships でしたが、users から articles へのリレーションは、Array relationshps になりました。

▲図 4.9　Array relationships

これは users 1 レコードに対して articles が複数紐付く 1:n の関係だからですね。

4.1.1　ダミーデータ作成

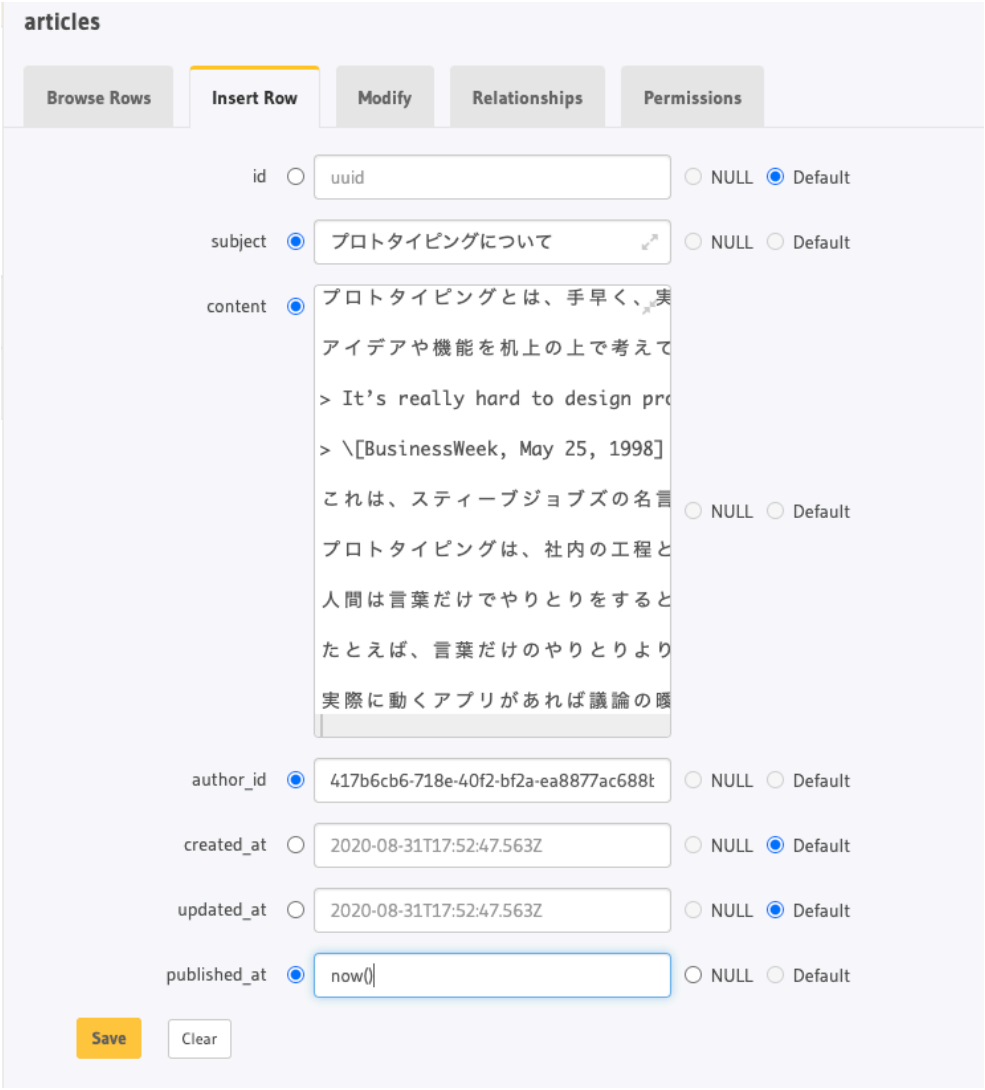
ここまでで、必要なテーブルが出来たので、ダミーデータを入れてみましょう。まずは `users` からです。



▲図 4.10　サンプルユーザー

筆者が実行したときには、ID は、`417b6cb6-718e-40f2-bf2a-ea8877ac688b` になりましたが、もちろん、これは実行するたびにランダムな値に変わるので、皆さんが実行するときは別の値になっているはずです。

次に `articles` に記事のデータを入れてみましょう。



▲図 4.11　記事

- subject には「プロトタイピングについて」
- content には、はじめにの原稿から抜粋したもの

- author_id はユーザーの UUID である 417b6cb6-718e-40f2-bf2a-ea8877ac688b
- published_at には now()

をそれぞれ設定したものです。
この記事の UUID は 415863a0-76d0-4926-a113-5420682775a5 になりました。

4.1.2 GraphQL query を試してみる

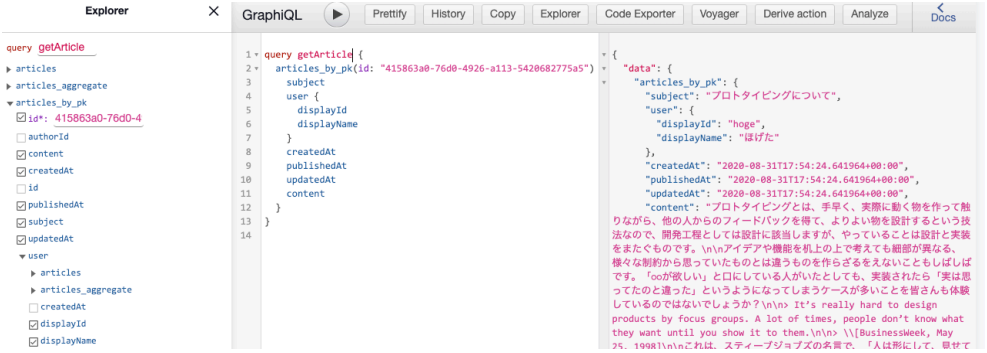
画面上部の GRAPHIQL タブに移動します。

```
query getArticle {
  articles_by_pk(id: "415863a0-76d0-4926-a113-5420682775a5") {
    subject
    user {
      displayId
      displayName
    }
    createdAt
    publishedAt
    updatedAt
    content
  }
}
```

このようなクエリを投げると

```
{
  "data": {
    "articles_by_pk": {
      "subject": "プロトタイピングについて",
      "user": {
        "displayId": "hoge",
        "displayName": "ほげた"
      },
      "createdAt": "2020-08-31T17:54:24.641964+00:00",
      "publishedAt": "2020-08-31T17:54:24.641964+00:00",
      "updatedAt": "2020-08-31T17:54:24.641964+00:00",
      "content": "プロトタイピングとは、手早く、実際に動く物を作って触りながら、他の
人からのフィードバックを得て、よりよい物を設計するという技
```

というような結果が返ってきました。



▲図 4.12　GraphiQL

どうやらうまくいったようです。

4.1.3　git commit する

さて、テーブル定義などが完了したので、いったん git に commit します。`git status` をみると、`hasura/migrations/` の下に、いくつも `down.sql` と `up.sql` が作成されています。

これらは、テーブルを作成、変更するたびに生成されるマイグレーション用 SQL 文ですが、1 つのマイグレーションデータにまとめる migrate squash[*6]（以後、squash） ができます。

```
$ yarn hasura --project hasura migrate status
...
VERSION        NAME                                                SOURCE STATUS  DAT
1598894684038  create_table_public_users                            Present        Pre
1598895613389  create_table_public_articles                         Present        Pre
1598896703138  alter_table_public_users_alter_column_display_id     Present        Pre
1598897074786  alter_table_public_users_add_column_created_at       Present        Pre
1598897083395  alter_table_public_users_add_column_updated_at       Present        Pre
1598897176691  alter_table_public_users_alter_column_created_at     Present        Pre
1598897184632  alter_table_public_users_alter_column_updated_at     Present        Pre
```

[*6] https://hasura.io/docs/1.0/graphql/core/migrations/migrations-setup.html#step-6-squash-migrations-and-add-checkpoints-to-version-control

VERSION はタイムスタンプです。一番、数値の若い 1598894684038 まで squash してみましょう。実行するコマンドは yarn hasura --project hasura migrate squash --name "create-users-articles" --from 1598894684038 です。

```
$ yarn hasura --project hasura migrate squash --name "create-users-articles" --from
WARN This command is currently experimental and hence in preview, correctness of sq
INFO Created '1598899352025_create-users-articles' after squashing '1598894684038'
INFO The following migrations are squashed into a new one:
1598894684038
1598895613389
1598896703138
1598897074786
1598897083395
1598897176691
1598897184632
1598899339908

INFO Do you want to delete these migration source files? (y/N)
```

ということなので、Y を押して Enter で squash されます。migrate status を見ると、create-users-artciles という名前の squash されたマイグレーションができていますが、DATABASE STATUS が Not Present になっています。

```
$ yarn hasura --project hasura migrate status
VERSION        NAME                   SOURCE STATUS  DATABASE STATUS
1598894684038  -                      Not Present    Present
1598895613389  -                      Not Present    Present
1598896703138  -                      Not Present    Present
1598897074786  -                      Not Present    Present
1598897083395  -                      Not Present    Present
1598897176691  -                      Not Present    Present
1598897184632  -                      Not Present    Present
1598899352025  create-users-articles  Present        Not Present
```

これは migrate squash によって新規に作成されたマイグレーションであり、サーバー側ではそれは実行されていないため Not Present というステータスになります。

認識してもらいつつも、実行自体はさせないために hasura migrate apply --skip-execution --version 1598899352025 というコマンドを実行する必要があります[*7]。

[*7] 個人的には、サーバー側からみるとそうかもしれないけど、融通効かせてくれてもいいんじゃないかなーなんて思ってしまったりします……。

実行するコマンドは yarn hasura --project hasura migrate apply --skip-execution --version 1598899352025 です。

```
$ yarn hasura --project hasura migrate apply --skip-execution --version 159889935202
INFO migrations applied
$ yarn hasura --project hasura migrate status
VERSION        NAME                   SOURCE STATUS  DATABASE STATUS
1598894684038  -                      Not Present    Present
1598895613389  -                      Not Present    Present
1598896703138  -                      Not Present    Present
1598897074786  -                      Not Present    Present
1598897083395  -                      Not Present    Present
1598897176691  -                      Not Present    Present
1598897184632  -                      Not Present    Present
1598899352025  create-users-articles  Present        Present
```

さて、ここから git にコミットしましょう。

https://github.com/erukiti/prototyping-sample/tree/92dd7a82b05dee456682908f8c0e1563347ef2e8

```
commit 92dd7a82b05dee456682908f8c0e1563347ef2e8
Author: SASAKI Shunsuke <erukiti@gmail.com>
Date:   Tue Sep 1 03:50:10 2020 +0900

    users と articles テーブルを作成
```

4.2 閲覧ページの URL を決める

ここまででテーブルやデータが出来たので、ブログ閲覧ページを実際に作っていきます。

さて、ブログ閲覧ページの URL に必要な要素は何か？ を考えます。

- ユーザーを識別する ID
- 記事を識別する ID

これら 2 つの ID が必要でしょう。

そこで /[userId]/[articleId] でアクセスできるとしましょう[*8]。

[*8] じつは今回の実装では userId がどんな値でも通るようになっています……。ちゃんと userId と articleId 両方で where 条件を組み立てるようにしなければいけません。

```
$ mkdir -p src/pages/[userId]/[articleId]
```

src/pages/[userId]/[articleId]/get-article.graphql に GraphQL クエリを定義しましょう。

▼リスト 4.4　src/pages/[userId]/[articleId]/get-article.graphql

```
query getArticle($id: uuid!) {
  articles_by_pk(id: $id) {
    subject
    user {
      displayId
      displayName
    }
    createdAt
    publishedAt
    updatedAt
    content
  }
}
```

このようなファイルを作っておけば、やっとコード生成が動くようになったので yarn gql-gen を実行します。

```
$ yarn gql-gen
```

さて、じつは 1 つトラップがあります。Timestamp や UUID というのは Hasura が勝手に定義した型（GraphQL のスカラー型）であり、gql-gen からすると、不明な型なので any を割り当ててしまうのです。

src/generated/graphql.ts の先頭の方に type Scalars という定義があります。

```
export type Scalars = {
  ID: string;
  String: string;
  Boolean: boolean;
  Int: number;
  Float: number;
  timestamptz: any;
  uuid: any;
};
```

困ったものですね。これをそのまま取り扱うわけにはいかないため、せめて any ではなく string を割り当てたいものです。

codegen.yml の末尾に、

```
      scalars:
        timestamptz: string
        uuid: string
```

という定義を追加する必要があります。

▼リスト 4.7　codegen.yml

```
schema: http://localhost:8080/v1/graphql/
documents:
  - ./src/**/*.graphql
overwrite: true
generates:
  ./src/generated/graphql.ts:
    plugins:
      - typescript
      - typescript-operations
      - typescript-react-apollo
    config:
      skipTypename: false
      withHooks: true
      withHOC: false
      withComponent: false
      scalars:
        timestamptz: string
        uuid: string
```

この設定変更をした状態でもう一度 gql-gen を実行すると、

```
export type Scalars = {
  ID: string;
  String: string;
  Boolean: boolean;
  Int: number;
  Float: number;
  timestamptz: string;
  uuid: string;
};
```

型定義が正常になりました。これはほんとめんどくさいトラップなのですが、現状では気をつけるしかありません。

この GraphQL を実際に叩くページを作成してみましょう。

▼リスト 4.9　src/pages/[userId]/[articleId]/index.tsx

```
 1: import { useRouter } from 'next/router'
 2: import { NextPage } from 'next'
 3:
 4: import { useGetArticleQuery } from '@/generated/graphql'
 5:
 6: const ArticlePage: NextPage = () => {
 7:   const router = useRouter()
 8:   const { articleId } = router.query
 9:
10:   const { loading, error, data } = useGetArticleQuery({
11:     variables: {
12:       id: articleId as string,
13:     },
14:   })
15:
16:   if (loading) {
17:     return <p>...loading</p>
18:   }
19:   if (error) {
20:     return <p>{error.toString()}</p>
21:   }
22:   return <div>{JSON.stringify(data)}</div>
23: }
24:
25: export default ArticlePage
```

`import { useGetArticleQuery } from '@/generated/graphql'` によって、先ほど作成したクエリを実際に叩くための React Hooks 関数をインポートします。

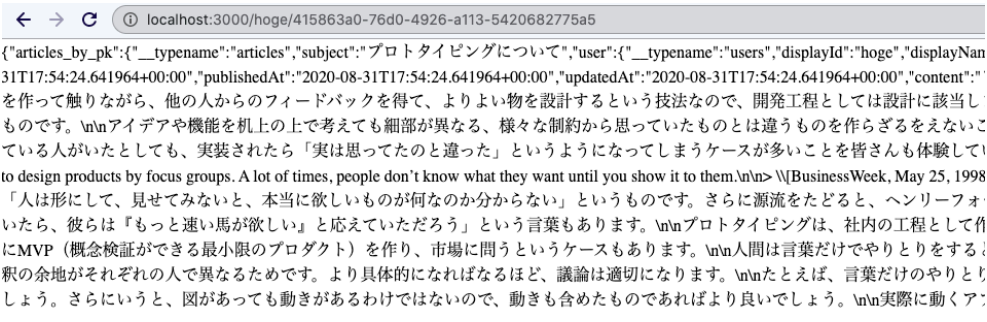
`useGetArticleQuery` 関数の引数には、`{ variables: { id: ... }}` という形で `id` を指定します。これは記事の ID `articleId` であり、Next の動的ルーティングによって、`const router = useRouter()` および `const { articleId } = router.query` のコードで `articleId` が取得できます。

戻り値は `loading error data` となっていて、アクセス中（ローディング中）は `loading` フラグが立っているため、ローディング用の処理を差し込みましょう。プロトタイピングでは優先度は低いため `<p>...loading</p>` を描画しています。

`error` は API アクセス時に生じたエラーが返ってきます。GraphQL にまつわるエラー

やネットワークにまつわるエラーが返ってくるはずです。これもプロトタイピングでは重要ではないため、確認するために {error.toString()} を描画しています。

さて loading も error も無い場合は、data に何が入っているか確認するために {JSON.stringify(data)} を描画しています。



▲図 4.13　これで JSON データを見た結果

さて、これで一度コミットしておきましょう。

https://github.com/erukiti/prototyping-sample/tree/07147a696b806143f6fcbd4aaeb1842a5486f432

```
commit 07147a696b806143f6fcbd4aaeb1842a5486f432
Author: SASAKI Shunsuke <erukiti@gmail.com>
Date:   Tue Sep 1 03:53:35 2020 +0900

    記事のJSONを表示
```

4.2.1　少しずつ見た目を整えていく

さきほど作成した index.tsx を修正して、見た目を整えましょう。

articleId による記事検索がヒットしなかった場合など、data や data.articles_by_pk が空の場合には <Error statusCode={404} /> で 404 ページを描画するようにしましょう。

▼リスト 4.11　Error を import する

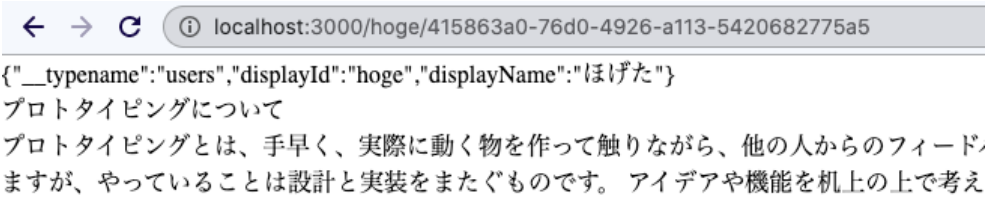
```
1: import Error from 'next/error'
```

ArticlePage の return {JSON.stringify(data)} を書き換えましょう。

▼リスト 4.12　ArticlePage を書き換える

```
 1:   if (!data || !data.articles_by_pk) {
 2:     return <Error statusCode={404} />
 3:   }
 4:
 5:   const { user, subject, content } = data.articles_by_pk
 6:
 7:   return (
 8:     <div>
 9:       <div>{JSON.stringify(user)}</div>
10:       <div>{subject}</div>
11:       <div>{content}</div>
12:     </div>
13:   )
```

あとは少しずつ表示をやりはじめます。



▲図 4.14　subject と content を表示

さて、デザインとしては世界的によくあるような見た目にしましょうか。

▼リスト 4.13　src/pages/[userId]/[articleId]/index.module.css

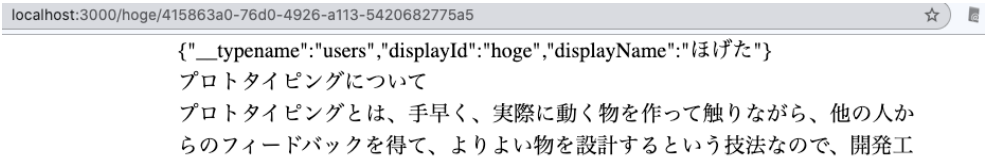
```
1: .contentContainer {
2:   width: 700px;
3:   margin: 0 auto;
4:   font-size: 20px;
```

```
  line-height: 30px;
}
```

▼リスト 4.14　ArticlePage の div を書き換える

```
return (
  <div className={styles.contentContainer}>
    <div>{JSON.stringify(user)}</div>
    <div>{subject}</div>
    <div>{content}</div>
  </div>
)
```

まずは幅を狭めて中央に表示します。ブログではあんまり横にキツキツにすると読みづらすぎるのです。また、デフォルトではフォントサイズも小さいですし、行間も詰まっていて読みづらいので広げましょう。



▲図 4.15　ブログを表示するためのコンテナを作った

もう少し行間をいじらないといけないかもしれないですが、ひとまずはこんなところでしょう。

▼リスト 4.15　src/pages/[userId]/[articleId]/index.tsx

```
import { useRouter } from 'next/router'
import { NextPage } from 'next'
import Error from 'next/error'

import { useGetArticleQuery } from '@/generated/graphql'

import styles from './index.module.css'

const ArticlePage: NextPage = () => {
  const router = useRouter()
  const { articleId } = router.query
```

```
12:
13:   const { loading, error, data } = useGetArticleQuery({
14:     variables: {
15:       id: articleId as string,
16:     },
17:   })
18:
19:   if (loading) {
20:     return <p>...loading</p>
21:   }
22:   if (error) {
23:     return <p>{error.toString()}</p>
24:   }
25:
26:   if (!data || !data.articles_by_pk) {
27:     return <Error statusCode={404} />
28:   }
29:
30:   const { user, subject, content } = data.articles_by_pk
31:
32:   return (
33:     <div className={styles.contentContainer}>
34:       <div>{JSON.stringify(user)}</div>
35:       <div>{subject}</div>
36:       <div>{content}</div>
37:     </div>
38:   )
39: }
40:
41: export default ArticlePage
```

https://github.com/erukiti/prototyping-sample/tree/c70037e2bf3f8bb1be67a6442fa53f2ced89c492

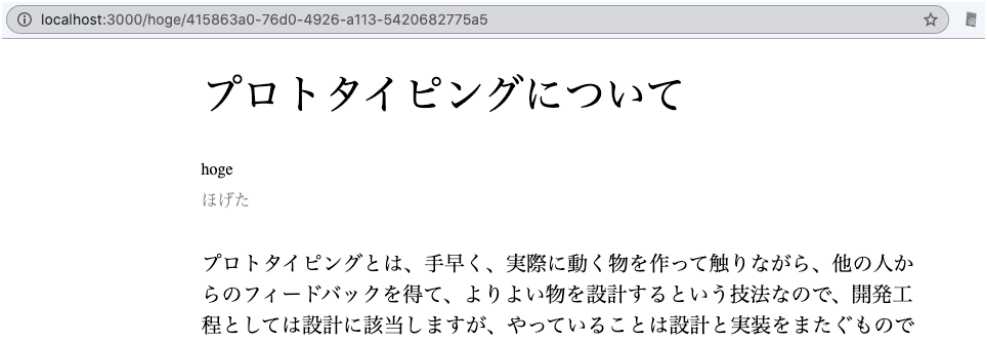
```
commit c70037e2bf3f8bb1be67a6442fa53f2ced89c492
Author: SASAKI Shunsuke <erukiti@gmail.com>
Date:   Tue Sep 1 04:03:40 2020 +0900

    見た目を整える1
```

このままではまだブログっぽさが全然ありません。ユーザー情報、タイトル、本文をそれぞれいじってみましょう。

▼リスト 4.17　src/pages/[userId]/[articleId]/index.module.css

```
.subject {
  margin-top: 32px;
  font-size: 40px;
  line-height: 48px;
}

.userContainer {
  margin-top: 32px;
  display: flex;
}

.userId {
  font-size: 16px;
}

.userName {
  font-size: 16px;
  opacity: 0.5;
}

.content {
  margin-top: 32px;
}
```

```
  return (
    <div className={styles.contentContainer}>
      <h1 className={styles.subject}>{subject}</h1>
      <div className={styles.userContainer}>
        <div>
          <div className={styles.userId}>{user.displayId}</div>
          <span className={styles.userName}>{user.displayName}</span>
        </div>
      </div>
      <div className={styles.content}>{content}</div>
    </div>
  )
```



▲図 4.16　タイトル、ユーザー情報、本文をそれっぽく

もちろんまだ色々と足りないですが、なんとなくそれっぽくなってきたように見えます。

https://github.com/erukiti/prototyping-sample/tree/6f7d9ddcdf2b487194bd9e198ed4d7197da69b2d

```
commit 6f7d9ddcdf2b487194bd9e198ed4d7197da69b2d
Author: SASAKI Shunsuke <erukiti@gmail.com>
Date:   Tue Sep 1 04:06:28 2020 +0900

    見た目を整える2
```

さて、分かっていたことですが、ID と名前が表示されるだけではまだ寂しいです。最近のメディアでアイコンがないというのは、なかなかあり得ない話でしょう。

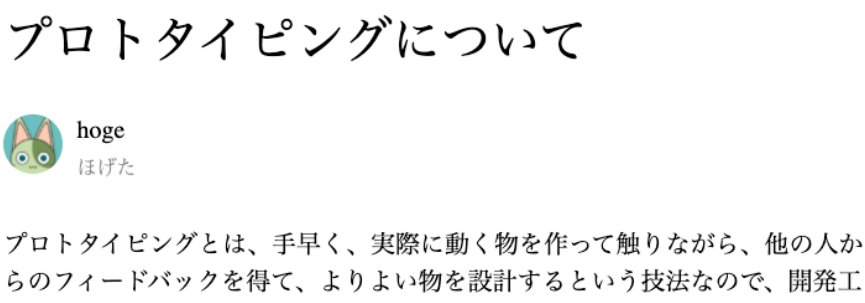
アイコンや画像など、ファイル置き場が必要になりますが、いったんアイコンは決め打ちで public に置いてしまいましょう。化け猫アイコンメーカー[*9]からダウンロードしてきた png 画像を、/public/profile.png として置きます。

[*9] http://neutralx0.net/tool/bnmk.html

```
 1: .userContainer {
 2:   margin-top: 32px;
 3:   display: flex;
 4:   align-items: center;
 5: }
 6:
 7: .userText {
 8:   margin-left: 12px;
 9: }
10:
11: .userIcon {
12:   width: 48px;
13:   height: 48px;
14:   border-radius: 24px;
15: }
```

```
1:       <div className={styles.userContainer}>
2:         <div>
3:           <img className={styles.userIcon} src="/profile.png" />
4:         </div>
5:         <div className={styles.userText}>
6:           <div className={styles.userId}>{user.displayId}</div>
7:           <span className={styles.userName}>{user.displayName}</span>
8:         </div>
9:       </div>
```

この変更の結果



▲図 4.17　丸形アイコンを追加

のようになりました。どんどんそれっぽくなっていきます。
アイコンは元々四角形の普通のアイコンですが、CSS によってサイズ調整と丸くして

います。width と height を 48px に固定して、border-radius をその半分である 24px にすると、丸いアイコンになります。

https://github.com/erukiti/prototyping-sample/tree/1f7de127d5da01f33ba3dee339c2cbcb13d5814c

```
commit 1f7de127d5da01f33ba3dee339c2cbcb13d5814c
Author: SASAKI Shunsuke <erukiti@gmail.com>
Date:   Tue Sep 1 04:13:24 2020 +0900

    見た目を整える3
```

4.2.2 フォーマットをどうするか

さて、ブログっぽい画面をつくることはできましたが、現時点では本文のフォーマットが定まっていません。技術者向けならば Markdown を採用しておけば間違いはありませんが、今回のターゲット層は技術者だけではありません。

ならば、リッチテキストを採用すべきか？ もちろん、ユーザーの利便性を考えると、Note や Medium くらいのリッチさは欲しいでしょう。いま、ブログ界隈に参戦するのであれば、最低限あれくらいのリッチさがないと戦えないでしょう。実際にはもっとユーザーに沿った使い勝手の良い UI を考えるべきかもしれません。

ただ、どこを重視してプロトタイピングをするか？ という点を忘れてはいけません。

UI を先に議論したいのであればリッチテキストエディタを採用すべきです。ウェブ上のエディタ作成は様々な理由により修羅の道なので自作はオススメできませんが、いい感じのエディタを自作できればそれで優位性も得られるかもしれません。

今回は入力 UI を論点にしていない空気感だったとします。その場合、必ずしもリッチテキストエディタにこだわる必要はありません。そこで、フォーマットをほぼ考慮しない。まずは空行による、文の分割のみを考慮する。という割り切りをしてみましょう。

コンテンツの描画は、再利用できる可能性が高いことや、独立性があるべきであることから、コンテンツ描画用のコンポーネントを作成します。

▼リスト 4.23　src/components/article/index.tsx

```
1: import React from 'react'
2:
3: type Props = {
4:   content: string
5: }
6:
7: export const Article: React.FC<Props> = ({ content }) => {
8:   return <div>{content}</div>
9: }
```

▼リスト 4.24　src/pages/[userId]/[articleId]/index.tsx

```
1: import { Article } from '@/components/article'
```

▼リスト 4.25　src/pages/[userId]/[articleId]/index.tsx

```
1: //    <div className={styles.content}>{content}</div>
2: //    を書き換える
3:       <div className={styles.content}>
4:         <Article content={content} />
5:       </div>
```

これで、コンテンツの中身のレンダリングをコンポーネントに外だしできました。

さて、あとは content を分割したり、スタイルをつけていきましょう。

▼リスト 4.26　src/components/article/index.module.css

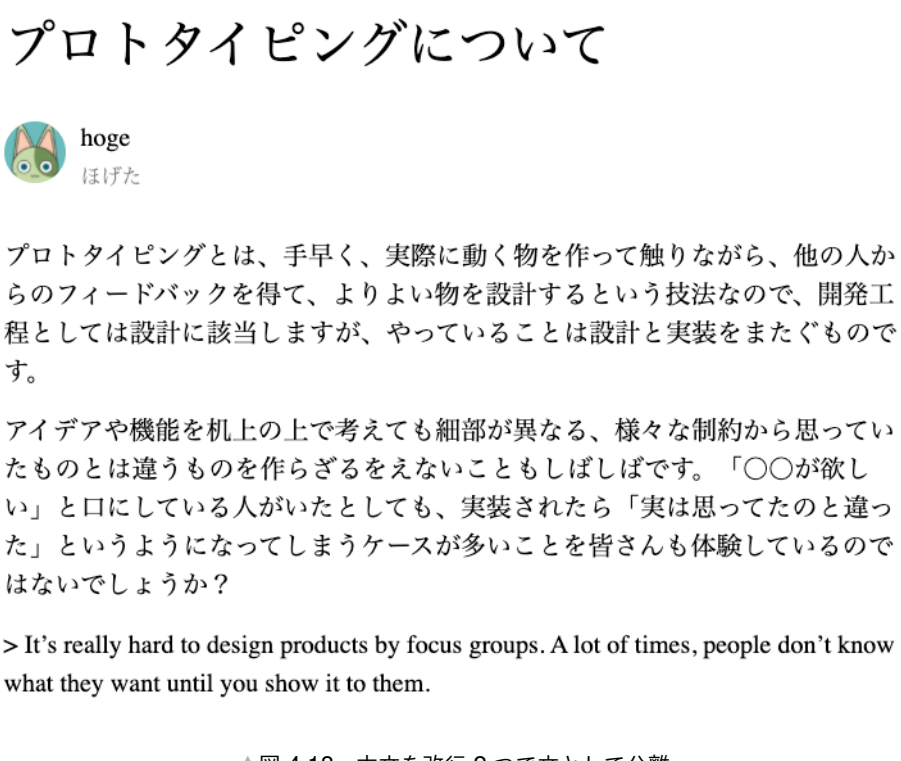
```
1: .paragraph {
2:   margin-top: 1rem;
3: }
```

▼リスト 4.27　src/components/article/index.tsx

```
1: import React from 'react'
2:
3: import styles from './index.module.css'
4:
5: const Paragraph: React.FC<{ p: string }> = ({ p }) => {
6:   return <div className={styles.paragraph}>{p}</div>
7: }
8:
9: type Props = {
```

```
10:   content: string
11: }
12:
13: export const Article: React.FC<Props> = ({ content }) => {
14:   return (
15:     <>
16:       {content.split('\n\n').map((p, i) => (
17:         <Paragraph p={p} key={i} />
18:       ))}
19:     </>
20:   )
21: }
```

これで、文ごとに分割されるようになり、よりブログっぽくなりました。



▲図 4.18　本文を改行 2 つで文として分離

https://github.com/erukiti/prototyping-sample/tree/572e0791dc94ba9b6

7c390b182bf450b500aefb5

```
commit 572e0791dc94ba9b67c390b182bf450b500aefb5
Author: SASAKI Shunsuke <erukiti@gmail.com>
Date:   Tue Sep 1 04:19:55 2020 +0900

    見た目を整える4
```

さて、ここで少しリファクタリングをします。と言ってもさっき書いた Paragraph を別のファイルに切り出すだけですが。

▼リスト 4.29　src/components/article/paragraph.tsx

```
 1: import React from 'react'
 2:
 3: import styles from './index.module.css'
 4:
 5: type Props = {
 6:   p: string
 7: }
 8:
 9: export const Paragraph: React.FC<Props> = ({ p }) => {
10:   return <div className={styles.paragraph}>{p}</div>
11: }
```

▼リスト 4.30　src/components/article/index.tsx

```
 1: import React from 'react'
 2:
 3: import { Paragraph } from './paragraph'
 4:
 5: type Props = {
 6:   content: string
 7: }
 8:
 9: export const Article: React.FC<Props> = ({ content }) => {
10:   return (
11:     <>
12:       {content.split('\n\n').map((p, i) => (
13:         <Paragraph p={p} key={i} />
14:       ))}
15:     </>
16:   )
17: }
```

https://github.com/erukiti/prototyping-sample/tree/141d8d16881f2bfd3

15c94fea3d2721916fca84a

```
commit 141d8d16881f2bfd315c94fea3d2721916fca84a
Author: SASAKI Shunsuke <erukiti@gmail.com>
Date:   Tue Sep 1 04:21:01 2020 +0900

    リファクタリング
```

4.2.3 必要な機能は何？

さて、それっぽくなってくると、寂しい部分がさらに浮き上がってきます。

- 投稿日時（公開日時）
- 何分でこの記事を読めるかの目安
- Twitter link button
- Fav/Star
- サイトヘッダ（サイトのロゴ、トップページへの遷移、ログインステータスなど）
- コメント機能
- タグ
- サイドバー

などなど。

ただし、これらのうち、どれが本当に必要なものなのでしょうか？

投稿日時（公開日時）は必須でしょう。

何分でこの記事を読めるかの目安は日本のブログサービスでは見かけませんが、海外だとついています。

Twitter link button なんかは割と、どこのサービスでも見かけるため、必須と考えても良さそうですが、どこに配置すべきか？ というデザイン上の議論の分かれるポイントです。

Fav/Star に該当する機能は要注意です。「この記事が好きである」という up vote としての事例と、「この記事を後で読み返したい」という stock / pinned の事例の二種類に分かれるからです。

たとえば、そういった分離がない事例としては、Medium や Note などが挙げられます。分離されてる事例では、Qiita や dev.to が挙げられます。Qiita はサービスの途中でストックという概念を分離したため、少し混乱が生じていたようです。

もし今からブログサービスを設計するならば、これらの概念をどう取り扱うべきか考慮すべきでしょう。つまり、議論が分かれるポイントになるものです。

サイトヘッダは、どのサービスでも同じようなものがあるため、混乱を避けるためにも必須と考えるべきでしょう。

コメント機能は、大抵のサービスにはありますが、炎上対策や spam 対策など頭を悩ませる要因になりがちです。安易に設計すると困ったことになることに注意したいため、やはりこれも議論が分かれるポイントになるものです。

タグ機能は、Qiita や dev.to では全面に押し出している機能で、Note では機能としてはあるもののあまり積極的には押し出していません。そのため、これも議論が分かれるポイントでしょう。

サイドバーは、デザイン性が強く表れるポイントです。Note はとにかくスッキリして読みやすさを心がけています。Qiita や dev.to は少し密な詰め込みをしています。Medium は、拍手（Fav）やコメント、ポケットのアイコンだけをサイドに表示していて、Note のスッキリさに近いです。これらを考えるとやはり議論が分かれるポイントでしょう。

ここまで検討してきて、必須と議論が分かれるものがはっきりしました。

- 投稿日時
- サイトヘッダ

この 2 つは必須かつ、議論の余地があまり無いところです。あえていうなら、投稿日時を日本のフォーマットにすべきか？ 海外に合わせるか？ などは考慮すべきですが。

さて、まずは投稿日時を追加しましょう。

```
1:   const { user, subject, content, publishedAt } = data.articles_by_pk
2:   console.log(publishedAt)
3:   // 2020-08-31T17:54:24.641964+00:00
```

`data.articles_by_pk` から `publishedAt` を取り出して表示してみます。`console.log` の結果は `2020-08-31T17:54:24.641964+00:00` という文字列でした。これならば、JavaScript の `Date` のコンストラクタに喰わせればそのまま `Date` 型のオブジェクトを取得できます。

あと、`ID / name` を分けて表示していましたが、チームメンバーから `name @ID` のような表記にして欲しいというツッコミを受けたので、そのように修正してみましょう。

▼リスト 4.33　CSS を追加

```
.userId {
  font-size: 16px;
}

.publishedAt {
  font-size: 16px;
  opacity: 0.5;
}

.content {
  margin-top: 32px;
}
```

```
  const { user, subject, content, publishedAt } = data.articles_by_pk
  if (!publishedAt) {
    return <Error statusCode={404} />
  }
```

publishedAt は Nullable カラムであり、null のときは記事を非公開にしているとします。そのため 404 エラーにしています。

さて、新しく追加したスタイルも込みで、ユーザー情報を大幅に書き換えましょう。

▼リスト 4.35　元のコード

```
        <div className={styles.userText}>
          <div className={styles.userId}>{user.displayId}</div>
          <span className={styles.userName}>{user.displayName}</span>
        </div>
```

↓

▼リスト 4.36　styles.userText の中を書き換える

```
        <div className={styles.userText}>
          <div className={styles.userId}>
            {user.displayName} @{user.displayId}
          </div>
          <span className={styles.publishedAt}>
            {new Date(publishedAt).toLocaleString()}
          </span>
        </div>
```

▲図 4.19　ユーザー情報に投稿日時を追加

まだとても手抜きな感じですが、それっぽく投稿日時を表示できました。

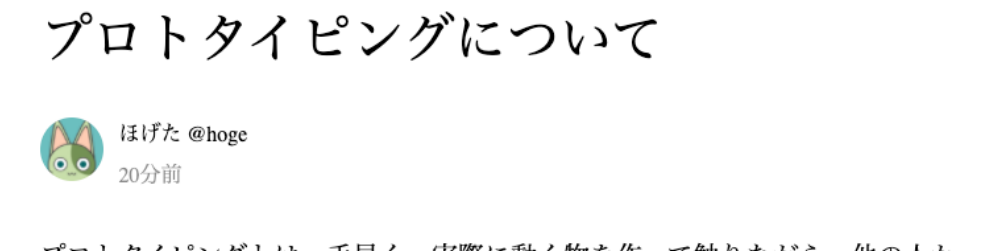
メンバーにここまで出来上がったものを見せたところ「日時はもうちょっと見せ方考えたいよねー」という話が持ち上がりました。

日時を見せるやり方は色々ありますが、JavaScript の `Date` 型は扱いがとにかく面倒です。moment のようなライブラリを使うのが一般的ですが、日付操作程度で、あの大きさのライブラリを使うのかーという貧乏精神が働いてしまうところもあります。

プロトタイプだから、あの程度のサイズなら問題ない、などの判断で moment を使ってもいいのですが、いずれにせよ、Date 型のオブジェクトに対して、何らかの求める結果を得られるようにしたい、ということで、どういうやり方を採用するにせよ、関数化しておくとよいのではないか？ と考えるところです。

それはさておき、見せ方を考えたいよねーと言ってるタイミングで関数を作ることを考えても仕方ありません。直接文字を埋め込みながら「こういう見せ方？」と問いかけるようにしてみましょう。

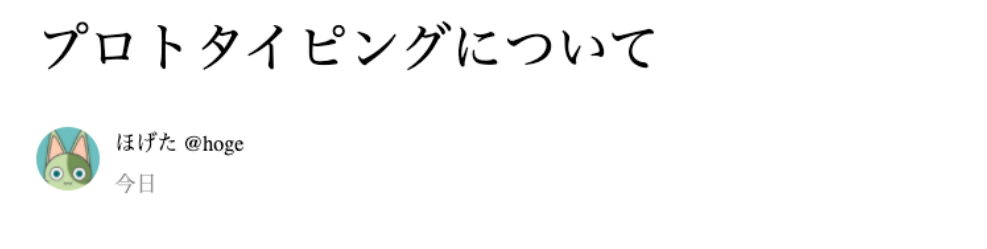
```
1:            <span className={styles.publishedAt}>20分前</span>
```



▲図 4.20　20 分前

実際のところ、これはピンと来ません。

```
1:            <span className={styles.publishedAt}>今日</span>
```



▲図 4.21　今日

何か面白さを感じますが、一つ議論が起こりました。「今日っていつよ？」「0 時 0 分からを今日と呼ぶのが正しいけど、それだと時間帯によってはすぐ今日が終わっちゃう」「24 時間経過したかどうかを争点にしてもいいけど、それだと今日という表現は正しくないよね」など。

海外のサービスでは投稿日時の時間をあまり気にしないものがありますが、日本人の感覚だからなのか、日時は注目したいポイントです。

少し考え方を変えて日時自体は変えずに、文言を付加してみてはどうでしょうか。

```
1:          <span className={styles.publishedAt}>
2:            {new Date(publishedAt).toLocaleString()} ☆投稿すぐ
3:          </span>
```

▲図 4.22　文言付加バージョン

文言や記号はともかく、日時と別に、付加情報があるのはいいかも？ という議論になりました。「New みたいなマーク付けるのはよくあるし、いいかも」「そういえば古い記事には、この記事は一年前の記事です。みたいな付加情報を付けるケースもあるよね」

とりあえず、日時ばかりを議論し続けても意味がないので、日時について大まかな方向性は決まったので、細かいフォーマットや付加情報は後ほど決めるということになりました。

4.2.4　TDD で日付のフォーマット処理をプロトタイピングする

大筋が決まったということで、リファクタリングも兼ねて、関数とそのテストを記述するため、TDD で作成しましょう。

まずはフォーマットした後の情報を型定義します。

```
1: export type FormattedDate = {
2:   datetime: string
3:   isNew: boolean
4:   // FIXME: n年前の記事です、のような情報は後でここに追加する
5: }
```

FormattedDate という型に情報を入れているため、情報を追加したければ後で追加できるので、今の時点で不要なものを入れない仕様にしましょう。適切にコードを書いていれば、そういった改修をするコストは大きくありません。

さて、TDD ということで、まず空の関数を書きます。

```
1: export const formatDate = (d: Date, now: Date): FormattedDate => {
2:   return {
3:     datetime: '',
4:     isNew: false,
5:   }
6: }
```

そして、テストコードを書きます。

```
 1: import { formatDate } from './date'
 2:
 3: describe('formatDate', () => {
 4:   const now = new Date('2020-08-25 17:58:53')
 5:   test('作成してすぐ', () => {
 6:     expect(formatDate(new Date(now), now)).toEqual({
 7:       datetime: '2020/8/25 17:58:53',
 8:       isNew: true,
 9:     })
10:   })
11: })
```

さて、jest を起動するのですが、TDD で開発しているときは特にそうですが、watch モードで起動するとテスト・改修のサイクルが捗ります。

```
$ yarn jest --watch
```

```
FAIL src/utils/date.test.ts
  formatDate
    × 作成してすぐ (12 ms)

  ● formatDate › 作成してすぐ

    expect(received).toEqual(expected) // deep equality

    - Expected  - 2
    + Received  + 2

      Object {
    -   "datetime": "2020/8/25 17:58:53",
    -   "isNew": true,
    +   "datetime": "",
    +   "isNew": false,
      }

       4 |   const now = new Date('2020-08-25 17:58:53')
       5 |   test('作成してすぐ', () => {
    >  6 |     expect(formatDate(new Date(now), now)).toEqual({
         |                                            ^
       7 |       datetime: '2020/8/25 17:58:53',
       8 |       isNew: true,
       9 |     })

      at Object.<anonymous> (src/utils/date.test.ts:6:44)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        5.316 s
Ran all test suites related to changed files.
```

▲図 4.23　当然エラーになる

https://github.com/erukiti/prototyping-sample/tree/9ffe339f54ada0fd5