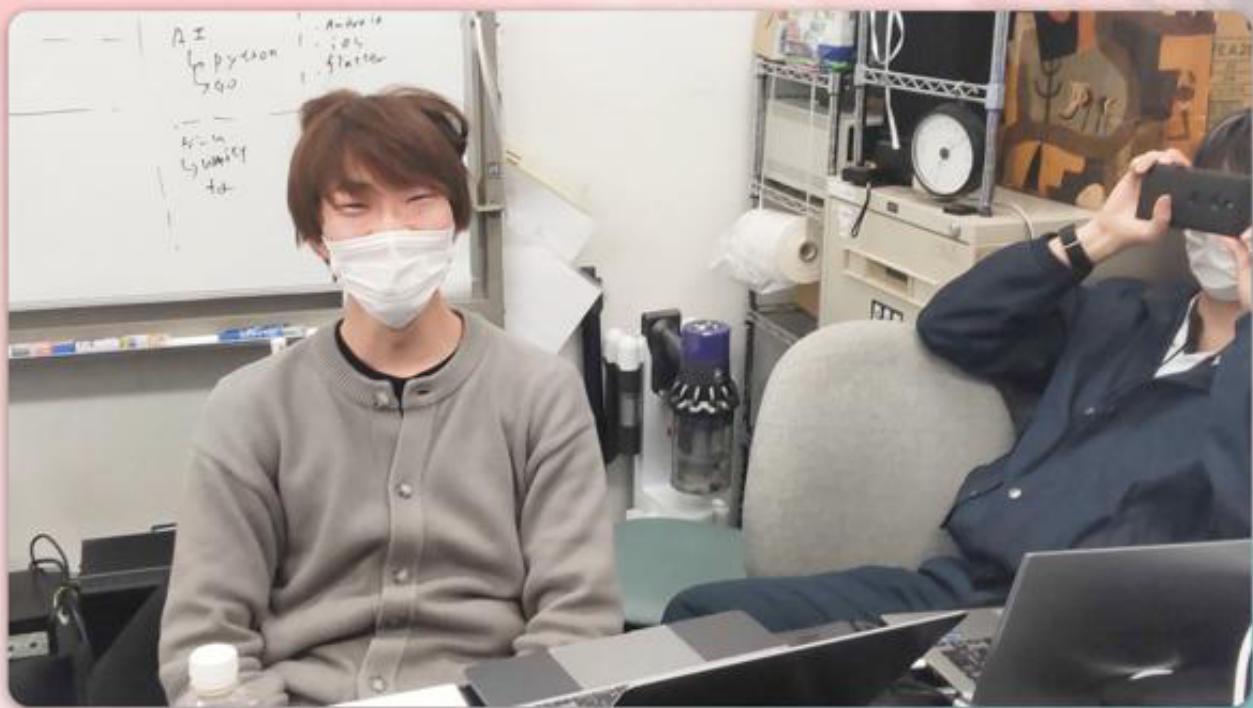


シス研の技術本 テスト作成 表紙



目次

第 1 章	シス研というサークルについて	3
1.1	はじめに	3
1.2	シス研とは	3
1.3	この本について	6
1.4	まとめ	6
第 2 章	DiscordBot を作ってみよう!	7
2.1	DiscordBot を作ってみよう	7
2.2	実行環境・使用技術	8
2.3	ローカル環境で Bot が動作するようにする	8
2.4	まとめ	11
第 3 章	リポジトリ作成後に設定しておきたいこと	12
3.1	はじめに	12
3.2	ファイルの設定	13
3.3	インフラ回り	16
3.4	おわりに	17
第 4 章	Gin × Neo4j × Docker で最短経路を返してくれる API サーバを建てる	19
4.1	Gin × Neo4j × Docker で最短経路を返してくれる API サーバを建てる	19
4.2	今回扱うデータの図	20
4.3	ディレクトリ構造	21
4.4	Neo4j に最初にインポートするファイル	22
4.5	Go のソースコード	29
4.6	実行方法	35
4.7	おわりに	35
第 5 章	Next.js 13 触ってみた	36
5.1	はじめに	36
5.2	プロジェクト作成からサーバ起動	36

5.8	終わりに	55
第 6 章	これは chapter	56
6.1	これは section	56
第 7 章	これは chapter	58
7.1	これは section	58
第 8 章	これは chapter	60
8.1	これは section	60

1

シス研というサークルについて

1.1 はじめに

初めまして！シス研会長の林です！はい、ここでみなさんシス研とはなんぞ？となっていると思うのでまずは自分が水先案内人となりましてシス研とこの本について解説していこうと思います。

1.1.1 どんな話をするのか

シス研ってどんなサークル？どんな活動をしているの？この本はどういったもの？といったものを紹介していきます。それでは、さっそく行ってみましょう！！！

1.2 シス研とは

シス研は正式名称を「システム工学研究会」と言い、愛知工業大学公認の情報系サークルです。歴史は長く、2023 年で創立 47 周年を迎え、かの Apple と同じ年となります！！

シス研ではハッカソン出場をはじめとしたチーム開発、ゲーム作成、インフラの構築、運用などを行っています。

1.2.1 どんな活動をしているの？

シス研の主な活動はチーム開発とインフラ整備です。サークル全体としての開発物などではなく、それぞれがチームを組んでハッカソンに出場したりしています。

インフラ面では、部室に物理サーバを持っており、そこでシス研のホームページや各種サービスを公開しています。^{*1*2}23年4月現在、大幅な工事を行なっておりごく一部のサービスのみ稼働しています（すみません）そのほかにもシス研主催のLT会・ハッカソンの開催、stech様^{*3}と共同でQiita Advent Calendarへの参加もしています。^{*4}



▲図1.1 部室の様子

^{*1} シス研ホームページ <https://set1.ie.aitech.ac.jp>

^{*2} シス研紹介ページ <https://welcome.sysken.net>

^{*3} stech様 HP <https://stech.careerselect.jp>

^{*4} Advent Calendar 2022 <https://qiita.com/advent-calendar/2022/stech-ait-advent>

1.2.2 2021～2022 年度の活動実績

- 2021 愛工大大学祭 工科展 最優秀賞
- 2022 技育博 参加
- 2022 Geekcamp vol8 優秀賞
- 2022 愛工大大学祭 工科展 瑞若賞
- 2022 技育展 出展
- 2022 愛工大大学祭 模擬店 最優秀賞
- 2022 HackU 春・夏 参加
- 2022 Geekcamp アドバンス 登壇
- 長期休暇中の LT 会、ハッカソン主催
- 各種勉強会の開催

シス研の設備

- ブレードサーバ、ネットワーク機器
- デスクトップ PC
- iMac,MacBook
- iPhone,iPad
- Android 端末各種
- Raspberry Pi
- はんだ等の電子工作セット
- その他多数...



▲ 図 1.2 ブレードサーバ



▲ 図 1.3 ネットワーク機器



▲ 図 1.4 タブレット端末



▲ 図 1.5 Raspberry Pi

1.3 この本について

この本はシス研のメンバーが経験したこと、取り組んだことのアウトプットを目的としたものです。この本を通じて皆さんにはシス研のメンバーは具体的にどのような活動をしているのか知ってもらいたいと思います。

また、シス研として本を出すのは今回が初めてなのでどうか温かい目で見ていただけると幸いです。

1.4 まとめ

ここまでお話を来てきましたが簡単にでもシス研について知ってもらうことはできただでしょうか？うまく伝えることができていたらとても嬉しいです。

次のページからはメンバーの記事本編になります！シス研初めての本をよろしくお願ひします！！

2

DiscordBot を作ってみよう!

2.1 DiscordBot を作ってみよう

2.1.1 はじめに

はじめまして、suda です。私は Discord で人とチャットをしている時に同じ会話が頻繁に続き、これ Bot で返事をするようにしたら返事をする手間が省けるし面白いのでは？と思い Bot を作ることにしました。発想がひどいって！？まあでも自分の発想したものを形にすることが面白いことだと思うので今回はそこには目を瞑りましょう…もちろん自分が送ったメッセージに対して Bot に返答させることもできるので、自分だけのオリジナル DiscordBot を作ってみましょう！

2.1.2 何を作るのか

Discord のサーバで特定のメッセージが来たら、特定のメッセージを返す Discord の Bot を作ります。サンプルプログラムを参照したい方は以下の URL からご覧下さ

い。^{*1} 例えば自分が「仕事終わった」と言うと Bot が「お疲れ様」と返してくれます。

2.2 実行環境・使用技術

- Python 3.10.8

2.3 ローカル環境で Bot が動作するようにする

まずはローカル環境で Bot が動作するようにしてみます。

2.3.1 Bot の作成・管理をする

初めに、機能などはまだついていない Bot を Discord のポータルサイトから作成します。Discord の Bot の作り方(メモ)という記事の「1.Discord 上の Bot の作成」を見ながら Bot を作成してみて下さい。^{*2}

2.3.2 ファイルの作成

Bot を実行する Python ファイルを作ります。

```
mkdir message_discord_bot  
cd message_discord_bot  
touch main.py
```

^{*1} 今回作る DiscordBot のサンプルプログラム https://github.com/sudamichiyo/Discord_Bot_sampleprogram

^{*2} Discord の Bot の作り方(メモ)<https://note.com/exteoi/n/nf1c37cb26c41>(参照 2023.3.29)

2.3.3 discord.py の準備

ここからは discord.py のドキュメントを見ながら環境構築をしていきます。³ Python で Discord の API を操作するために必要なライブラリをインストールします。先ほど作成したディレクトリにアクセスして、以下のコマンドで discord.py をインストールします。

```
python -m pip install -U discord.py
```

次に、先ほど作成した main.py を以下のソースコードに書き換えます。

```
1 import discord
2
3 class MyClient(discord.Client):
4     async def on_ready(self):
5         print(f'Logged on as {self.user}!')
6
7     async def on_message(self, message):
8         print(f'Message from {message.author}
9             : {message.content}')
10
11 intents = discord.Intents.default()
12 intents.message_content = True
13
14 client = MyClient(intents=intents)
15 client.run('my token goes here')
```

ここで、以下のボットに関する 2 つの設定を Discord のポータルサイトから設定してください。

³ discord.py ドキュメント <https://discordpy.readthedocs.io/ja/latest/intro.html#basic-concepts> (参照 2023.3.29)

- ポータルサイトの「Bot」からトークンを取得する
- ポータルサイトの「Bot」の「MESSAGE CONTENT INTENT」を有効にする

'my token goes here' は取得した Bot のアクセストークンを書きます。以上の設定が終わったところで python3 main.py を実行すると、Bot のサーバが立ち上がりまします。Bot サーバ起動後に Bot のいるサーバでメッセージを投げると、コマンドライン上に「書いた人」と「メッセージ」がそのまま出力されます。

2.3.4 環境変数の設定

ソースコードに直接トークンを書いてしまうと、Github でソースコードをホスティングするときにトークンキーが他の人にバレてしまいます。これを防ぐために.env ファイルを作成して、その中に Discord のアクセストークンを書きます（下記参照）。

```
1 DISCORD_TOKEN='My token goes here'
```

Python の中で.env ファイルに書かれている変数を取得するために dotenv というライブラリを使用します。以下のようにインストールします。

```
pip install python-dotenv
```

インストール後に main.py に下記のコードを付け加えて下さい。

main.py の import discord と class MyClient の間に以下のコードを追加します。

```
1 import os  
2 from dotenv import load_dotenv  
3 load_dotenv()
```

そして、最後の行を以下のように書き換えて下さい。

```
1 client.run(os.environ['DISCORD_TOKEN'])
```

書き換えたあとに python3 main.py を実行すると、先ほどと同じようにメッセージの受け取りをしてくれるサーバーサイドアプリケーションが立ち上がります。

2.3.5 Bot が特定のワードに反応して、特定のメッセージを返答する機能をつける

プログラムを起動して正常にサーバーサイドアプリケーションがメッセージを受け取れるようになったら、Bot が特定のワードに反応して、特定のメッセージを返答する機能をつけていきます。Bot に機能をつけるには上記のソースコードの 8 行目と 10 行目の間に以下のコードを付け足していきます。

```
1 # メッセージを書いた人が Bot なら処理終了
2 if message.author.bot:
3     return
4 channel = message.channel
5 if message.content == '仕事終わった':
6     await channel.send('お疲れ様')
```

付け足したコードの解説をしていきます。

2,3 行目でメッセージを書いた人が Bot なら処理を終了させています。

4 行目でメッセージが投稿されたチャンネル取得しています。

5 行目の message.content はメッセージの内容で、今回の場合「仕事終わった」というメッセージをチャンネルに投稿すると、メッセージが投稿されたチャンネルに Bot が「お疲れ様」と返答します。

2.4 まとめ

今回は Discord の Bot の作り方を説明しました。上記の「仕事終わった」や「お疲れ様」に当たる部分を変えたりして自分好みに改良してみて下さい。

3

リポジトリ作成後に設定しておきたい こと

3.1 はじめに

こんにちは。hihumikan です。

本チャプターでは「リポジトリ作成後に設定しておきたいこと」をご紹介します。私自身が次回プロジェクト開始する際に、こういった設定や技術を使うだろうというものをまとめました。

ただし、これら全ての内容をプロジェクトに適用出来るものではないため、ご利用の環境や用途に合わせて利用していただければと思います。

3.1.1 対象読者

対象読者は、Git/GitHub を利用した開発を行う初学者の方を想定しています。

3.2 ファイルの設定

リポジトリ作成後に設定しておきたい「ファイルの設定」についてご紹介します。

3.2.1 .gitignore

.gitignore は、Git による管理から除外したいファイルやディレクトリを指定するための設定ファイルです。

例えば、MacOS の場合、ディレクトリ毎に.DS_Store というファイルが自動的に生成されます。このファイルは、ディレクトリの meta 情報を記録しますが、通常の開発において共有する必要がないため、Git の管理対象外としておくのが望ましいです。

また、.env などの環境変数を利用してプログラムを動かす場合、.env ファイルには、パスワードや秘密鍵などの情報が含まれているのが多いため、これも Git の管理対象外としておくのが望ましいです。

リスト 1.1 のようなテキストファイルを Git の管理下に置くだけで、Git から管理対象外として扱われます。

リスト 1.1 .gitignore

```
1 .DS_Store  
2 .env
```

リポジトリを共有する場合、.gitignore ファイルを置いておくだけで、開発メンバー全員が同じ設定を利用できるため、必要なファイルだけが Git の管理下に置かれるようになります。

3.2.2 .gitattributes

.gitattributes は、特定のファイルに対して Git の挙動を変更するための設定ファイルです。主に、改行コードやファイルの文字コードを指定するために利用されます。

改行コードを指定する理由としては、Windows と MacOS では改行コードが異なる点が挙げられます。Git で管理されているファイルの改行コードが一致しないと、

差分が発生してしまい、想定した挙動と異なる動作をする可能性があります。それを防ぐために、`.gitattributes` に改行コードを明示しておくことで、安全に開発を進められます。

リスト 1.2 のように、ファイルの拡張子に対して、改行コードを指定が出来ます。

リスト 1.2 `.gitattributes`

```
1 * text=auto  
2 *.sh text eol=lf
```

これも`.gitignore` と同様に`.gitattributes` 共有するだけで、開発メンバー全員が同じ設定を利用できます。

3.2.3 Makefile

Makefile は、コマンドをまとめて実行するための設定ファイルです。

利点として、開発メンバー全員が同じコマンドを実行出来る所にあります。環境構築やテストの実行など、手順が複雑な作業を一人一人が実行した場合、手順の違いによるエラーが発生する可能性があります。それらを防ぐために、Makefile にまとめておくことで、開発メンバー全員が同じコマンドを実行でき、人的ミスを防げます。

Makefile の例としては、リスト 1.3 のようなものです。

リスト 1.3 Makefile

```
1 up: ## API とデータベースを起動
2   docker compose -f docker-compose-db.yml -p db up -d
3   docker compose -f docker-compose-api.yml -p api up -d
4
5 build: ## サービスの構築
6   docker compose -f docker-compose-db.yml -p db build
7   docker compose -f docker-compose-api.yml -p api build
8
9 stop: ## サービスを停止
10  docker compose -f docker-compose-db.yml -p db stop
11  docker compose -f docker-compose-api.yml -p api stop
12
13 kill: ## サービスを強制停止
14  docker compose -f docker-compose-db.yml -p db kill
15  docker compose -f docker-compose-api.yml -p api kill
16
17 down: ## サービスの停止とコンテナの削除
18  docker compose -f docker-compose-db.yml -p db down
19  docker compose -f docker-compose-api.yml -p api down
20
21 restart: ## サービスの再起動
22  docker compose -f docker-compose-db.yml -p db restart
23  docker compose -f docker-compose-api.yml -p api restart
```

これらを実行する場合、シェルに

```
$ make up
```

と入力するだけで、長いコマンドであった 2,3 行目のコマンドが簡単に実行されます。リスト 1.3 の例は簡単なものですが、他にも Makefile 内に変数を定義できるため、変更が必要な箇所を変数に置き換えることで、コマンドの変更を容易に行えます。

3.3 インフラ回り

3.3.1 GitHub Actions

Github Actions は、GitHub 上で動作する CI/CD ツールです。簡単に言えば、GitHub リポジトリに関連するイベントに応じて、あらかじめ定義しておいたワークフローを仮想マシン上で実行するものです

用途としては、コードの静的解析やテストの実行、デプロイなどが挙げられます。その他にも、Pull Request に対して、テスト内容をコメントしてくれるなどの GitHub 上での機能を利用できます。

利用方法としては、リスト 1.4 のように、.github/workflows ディレクトリを作成し、その中に設定ファイルを作成します。

```
リスト 1.4 .github/workflows/pr.yml

1 on:
2   pull_request:
3     types: [opened]
4   name: Pull Request
5   jobs:
6     assignAuthor:
7       name: Assign author to PR
8       runs-on: ubuntu-latest
9     steps:
10       - name: Assign author to PR
11         uses: technote-space/assign-author@v1
```

上記の例では、Pull Request が作成された際に、Pull Request の作成者を Assignee

に設定する設定ファイルの例です。

その他にも、リスト 1.5 のように、ssh 鍵を設定することで、リモートサーバーにアクセスができます。

リスト 1.5 .github/workflows/deploy.yml

```
1 name:CI
2 on:
3   push:
4     branches:
5       - main
6 jobs:
7   deploy:
8     runs-on: ubuntu-latest
9     steps:
10    - uses: actions/checkout@v2
11    - name: Install SSH Key for Deploy
12      uses: appleboy/ssh-action@master
13      with:
14        key: ${{ secrets.SK }}
15        host: ${{secrets.SSH_HOST}}
16        username: ${{secrets.SSH_USERNAME}}
17        port: ${{secrets.SSH_PORT}}
18        script: |
19          git pull
```

この他にも、コードを自動で整形して commit してくれるツールなどの様々なツールが存在します。調べてみると面白いかもしれません。

3.4 おわりに

本チャプターでは「リポジトリ作成後に設定しておきたいこと」をご紹介しました。ご紹介したのは一部分ですが、これらを設定しておくことで、開発の効率化や、

開発メンバーのミスを防げます。ぜひ、設定してみてください。

4

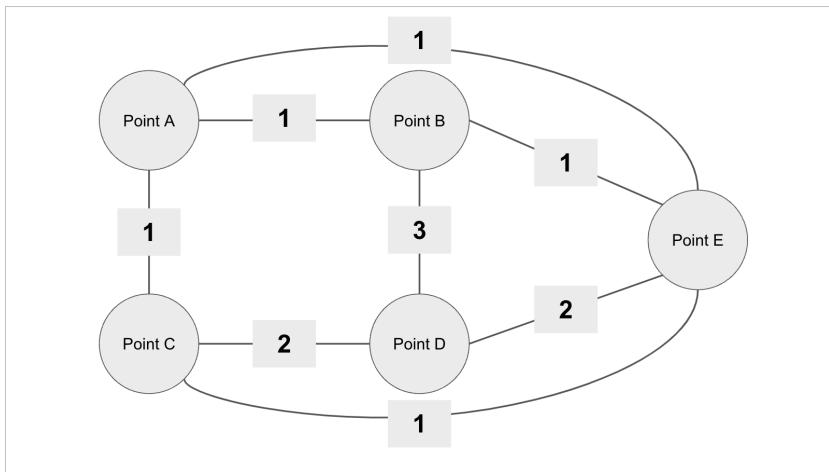
Gin × Neo4j × Docker で最短経路を返してくれる API サーバを建てる

4.1 Gin × Neo4j × Docker で最短経路を返してくれる API サーバを建てる

4.1.1 はじめに

このセクションでは、現在最もメジャーなグラフ DB である Neo4j と Github で多くのスターを獲得している Go 言語の Web フレームワーク、Gin を用いて API サーバーを構築していきます。また、実行環境統一のため Docker を用います。

4.2 今回扱うデータの図



4.3 ディレクトリ構造

```
ディレクトリ構造

├── build
│   ├── Docker
│   │   ├── go
│   │   │   └── Dockerfile
│   │   └── neo4j
│   │       ├── Dockerfile
│   │       └── volumes
│   │           ├── import
│   │           │   ├── done
│   │           │   ├── points.csv
│   │           │   └── route.csv
│   │           └── script
│   │               └── import_data.sh
│   └── docker-compose.yml
└── server
    ├── config
    │   ├── config.go
    │   └── environments
    │       └── neo4j.yml
    ├── controllers
    │   └── coordinate_controller.go
    ├── db
    │   └── neo4j.go
    ├── go.mod
    ├── go.sum
    ├── main.go
    ├── models
    │   └── coordinate.go
    ├── router
    │   └── router.go
    └── sample.http
```

4.4 Neo4j に最初にインポートするファイル

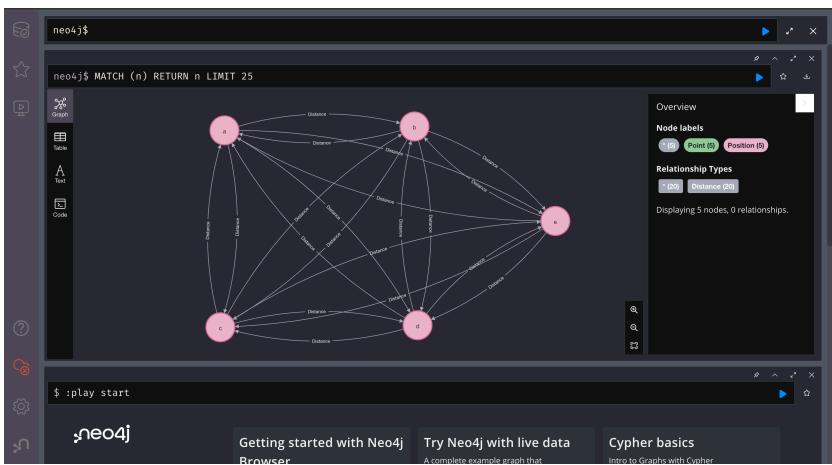
point.csv

```
1 point_id:ID,point_name,:LABEL
2 a,PointA,Point;Position
3 b,PointB,Point;Position
4 c,PointC,Point;Position
5 d,PointD,Point;Position
6 e,PointE,Point;Position
```

route.csv

```
:START_ID,:END_ID,:TYPE,cost:int  
b,a,Distance,1  
c,b,Distance,3  
d,c,Distance,2  
e,d,Distance,2  
c,a,Distance,1  
d,a,Distance,2  
e,b,Distance,1  
e,a,Distance,1  
c,e,Distance,1  
d,b,Distance,3  
a,b,Distance,1  
b,c,Distance,3  
c,d,Distance,2  
d,e,Distance,2  
a,c,Distance,1  
a,d,Distance,2  
b,e,Distance,1  
a,e,Distance,1  
e,c,Distance,1  
b,d,Distance,3
```

これらを Neo4j にインポートすることで、先ほどの図を表現することができます。



無事できてますね。

4.4.1 Docker 環境の設定

```
docker-compose.yml
```

```
'''  
1 version: "3"  
2 services:  
3   go:  
4     container_name: NEO4JAPI_GO  
5     build:  
6       context: ./docker/go  
7       dockerfile: Dockerfile  
8     stdin_open: true  
9     tty: true  
10    volumes:  
11      - ../server:/server  
12    ports:  
13      - 8080:8080  
14    networks:  
15      app_net:  
16        ipv4_address: 192.168.0.1  
17    depends_on:  
18      - "neo4j"  
19  
20  neo4j:  
21    container_name: NEO4JAPI_NEO4J  
22    build:  
23      context: ./Docker/neo4j  
24      dockerfile: Dockerfile  
25    restart: always  
26    ports:  
27      - 57474:7474  
28      - 57687:7687  
29    volumes:          25  
30      - ./Docker/neo4j/volumes/data:/data  
31      - ./Docker/neo4j/volumes/logs:/logs  
32      - ./Docker/neo4j/volumes/conf:/conf  
33      - ./Docker/neo4j/volumes/import:/import  
34      - ./Docker/neo4j/volumes/script:/script  
35    environment:
```

volumes はデータベースに入れるデータ、ログをバインドするための場所を指定しています。environment ではユーザーとパスワード、最初に読み込んでもらうシェルスクリプトの場所を指定します。

go 関連の Dockerfile。

Dockerfile

```
1 # go バージョン
2 FROM golang:1.19.3-alpine
3 # アップデートと git のインストール
4 RUN apk add --update && apk add git
5 # app ディレクトリの作成
6 RUN mkdir /server
7 # ワーキングディレクトリの設定
8 WORKDIR /server
9 # ホストのファイルをコンテナの作業ディレクトリに移行
10 ADD . /server
11 # main.go を実行
12 CMD ["go", "run", "main.go"]
```

Neo4j 関連の Dockerfile。

Dockerfile

```
```
\begin{verbatim}
1 # go バージョン
2 FROM golang:1.19.3-alpine
3 # アップデートと git のインストール
4 RUN apk add --update && apk add git
5 # app ディレクトリの作成
6 RUN mkdir /server
7 # ワーキングディレクトリの設定
8 WORKDIR /server
9 # ホストのファイルをコンテナの作業ディレクトリに移行
10 ADD . /server
11 # main.go を実行
12 CMD ["go", "run", "main.go"]
```

csv ファイルを読ませるためのシェルスクリプト。

```
import_data.sh
```

```
1 #!/bin/bash
2 set -euC
3
4 # EXTENSION_SCRIPT はコンテナが起動するたびにコールされるため、
5 # import 処理が実施済かフラグファイルの有無をチェック
6 if [-f /import/done]; then
7 echo "Skip import process"
8 return
9 fi
10
11 # データを全削除
12 echo "delete database started."
13 rm -rf /data/databases
14 rm -rf /data/transactions
15 echo "delete database finished."
16
17 # CSV データのインポート
18 echo "Start the data import process"
19 neo4j-admin import \
20 --nodes=/import/points.csv \
21 --relationships=/import/route.csv
22 echo "Complete the data import process"
23
24 # import 処理の完了フラグファイルの作成
25 echo "Start creating flag file"
26 touch /import/done
27 echo "Complete creating flag file"
```

## 4.5 Go のソースコード

### 4.5.1 config ディレクトリ

```
config.go

1 package config
2
3 import (
4 "github.com/spf13/viper"
5)
6
7 var n *viper.Viper
8
9 func init() {
10 n = viper.New()
11 n.SetConfigType("yaml")
12 n.SetConfigName("neo4j")
13 n.AddConfigPath("config/environments/")
14 }
15
16 func GetNeo4jConfig() *viper.Viper {
17 if err := n.ReadInConfig(); err != nil {
18 return nil
19 }
20 return n
21 }
```

このファイルで neo4j.yaml の環境変数を読み込みます。

### neo4j.yml

```
1 neo4j:
2 user: neo4j
3 password: admin
4 uri: neo4j://192.168.176.1:57687
```

Neo4j と接続するための環境変数です。

#### 4.5.2 db ディレクトリ

neo4j.go

```
1 package db
2
3 import (
4 "log"
5
6 "neo4japi/server/config"
7
8 "github.com/neo4j/neo4j-go-driver/v4/neo4j"
9)
10
11 func GetDriverAndSession() neo4j.Session {
12 n := config.GetNeo4jConfig()
13 dr, err := neo4j.NewDriver(n.GetString("neo4j.uri"), neo4j.BasicAuth(n.Get
14 if err != nil {
15 log.Fatal(err)
16 }
17 ses := dr.NewSession(neo4j.SessionConfig{AccessMode: neo4j.AccessModeRead})
18 return ses
19 }
20
```

このファイルで Neo4j と接続し、セッションを返すようにします。

#### 4.5.3 models ディレクトリ

coordinate.go

```
1 package models
2
3 import (
4 "fmt"
5 "log"
6
7 "neo4japi/server/db"
8
9 "github.com/neo4j/neo4j-go-driver/v4/neo4j"
10)
11
12 type Route struct {
13 Position string `json:"point"`
14 }
15
16 func FindRoute(fr, to string) []*Route {
17 var r []*Route
18 ses := db.GetDriverAndSession()
19 defer ses.Close()
20 cyp := fmt.Sprintf(`
21 MATCH (from:Position {point_name: "%s"}), (to:Position {point_name: "%s"})
22 path=allShortestPaths ((from)-[distance:Distance*]->(to))
23 WITH
24 [position in nodes(path) | position.point_name] as name,
25 REDUCE(totalMinutes = 0, d in distance | totalMinutes + d.cost) as 所
要時間
26 RETURN name
27 ORDER BY 所要時間
28 LIMIT 10;
29 `, fr, to)
30
31 _, err := ses.ReadTransaction(func(transaction neo4j.Transaction) (interface{}, error) {
32 result, err := transaction.Run(cyp, nil)
33 if err != nil {
34 return nil, err
35 }
36 var routes []*Route
37 for result.Next() {
38 var route Route
39 err = result.Decode(&route)
40 if err != nil {
41 return nil, err
42 }
43 routes = append(routes, &route)
44 }
45 return routes, nil
46 })
47
48 if err != nil {
49 log.Println("Error finding route: ", err)
50 return nil
51 }
52
53 return r
54 }
```

出発地点と目的地を受け取ることで Neo4j に Cypher というクエリ言語を用いて最短経路を導出してもらいます。

#### 4.5.4 controllers ディレクトリ

coordinate\_controller.go

```
1 package controllers
2
3 import (
4 "net/http"
5
6 "github.com/gin-gonic/gin"
7 "neo4japi/server/models"
8)
9
10 func RouteSearch(c *gin.Context) {
11 fr := c.Query("fr")
12 to := c.Query("to")
13 res := models.FindRoute(fr, to)
14 c.JSON(http.StatusOK, res)
15 }
```

GET リクエストで受け取ったパラメータを models で作成した関数に渡します。

#### 4.5.5 router ディレクトリ

```
router.go

1 package router
2
3 import (
4 "github.com/gin-gonic/gin"
5 "neo4japi/server/controllers"
6)
7
8 func Init() {
9 r := gin.Default()
10 r.GET("/coordinate", controllers.RouteSearch)
11 r.Run()
12 }
```

ルーティング先を定義します。

#### 4.5.6 実行ファイル

```
coordinate_controller.go

1 package main
2
3 import (
4 "neo4japi/server/router"
5)
6
7 func main() {
8 router.Init()
9 }
```

## 4.6 実行方法

```
docker compose up
```

このコマンドを実行することで Neo4j サーバーと Gin サーバーが立ち上がりります。

### 4.6.1 実行確認

```
sample.http
```

```
1 GET http://localhost:8080/coordinate?fr=PointB&to=PointE
```

今回は Vscode の拡張機能である REST Client を用いて実行確認を行います。

ここで、 coordinate?fr=PointB&to=PointE の部分を自分の好きな地点にしてリクエストを送ると、最短経路が返されます。

## 4.7 おわりに

このチャプターではグラフ DB と Go 言語を用いた API サーバーの建て方を説明しました。Gin、Neo4j は奥が深いので、もし気になった方はぜひ自分で調べて触ってみてください。

# 5

## Next.js 13 触ってみた

### 5.1 はじめに

10月後半に行われた Next.js Conf 2022 で発表された Next.js 13 を実際に触ってみたのでその内容を書きます。当初は全体を網羅して書く予定だったのですが量が多くなったの少し絞ってます。対象読者としてある程度 React や Next.js を触っている人を対象としています。

### 5.2 プロジェクト作成からサーバ起動

TypeScript と ESLint を入れるか聞かれるので入れる。

```
$ npx create-next-app@latest --ts
```

pages ディレクトリを削除。

```
$ rm -rf pages
```

app ディレクトリを作る。

```
$ mkdir app
```

app ディレクトリは実験段階の機能なので, next.config.js を変更する。

next.config.js

```
const nextConfig = {
 reactStrictMode: true,
 swcMinify: true,
 experimental: {
 appDir: true,
 },
};
```

app/page.tsx を作り、以下のようにする。

app/page.tsx

```
export default function Page() {
 return <h1>Hello, Next.js!</h1>;
}
```

ローカルサーバ起動。

```
$ npm run dev
```

ブラウザで <http://localhost:3000/> にアクセスすると、Hello, Next.js! が表示される。



▲ 図 5.1 Hello,Next.js

### 5.3 Layout と Head

サーバを起動すると自動的に app ディレクトリに layout.tsx、head.tsx というファイルが作られていました。

next/head を使って各ページファイルに head を定義していたのが head.tsx に書けるようになったみたいですね。

head.tsx

```
export default function Head() {
 return (
 <>
 <title></title>
 <meta content="width=device-width,initial-scale=1"
 name="viewport" />
 <link rel="icon" href="/favicon.ico" />
 </>
)
}
```

```
)
}
```

ページの共通レイアウトを定義するファイル。

```
export default function RootLayout({
 children,
}: {
 children: React.ReactNode
) {
 return (
 <html>
 <head />
 <body>{children}</body>
 </html>
)
}
```

今までには\_app.tsx などに下記のようにレイアウトを定義していました。

```
_app.tsx
```

```
<Layout>
 <Component {...pageProps} />
</Layout>
```

この方法だとページごとにレイアウトを変えられないです。なのでページごとにレイアウトを変えたい場合は getLayout を用いる必要がありました。Next.js 13 ではレイアウトを変えたいページがあるディレクトリに layout.tsx を置けばページごと

にレイアウトを変えられるようになりました。

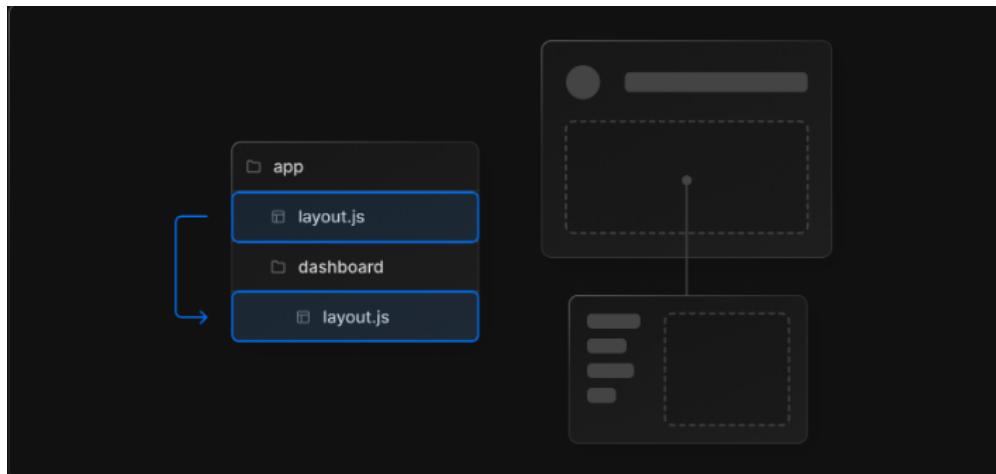
ここではダッシュボードページのレイアウトの場合を考えます。app ディレクトリの中で dashboard ディレクトリを作成して以下のように layout.tsx を配置するだけです。

app/dashboard/layout.tsx

```
export default function DashboardLayout({
 children,
}: {
 children: React.ReactNode;
}) {
 return <section>{children}</section>;
}
```

少し疑問に思ったのが dashboard ディレクトリの page.tsx では RootLayout は呼ばれず DashboardLayout のみが呼ばれるのかと思っていました。しかし、実際には入れ子構造で呼び出されるようです。

公式の画像がわかりやすいので貼っておきます。



▲ 図 5.2 layout.js の適用範囲

次の機能に行く前に dashboard ディレクトリに page.tsx も作っておきます。

app/dashboard/layout.tsx

```
export default function DashBoardPage() {
 return <h1>DashBoard Page</h1>;
}
```

違いがわかりやすいように他のファイルも変更します。

app/layout.tsx

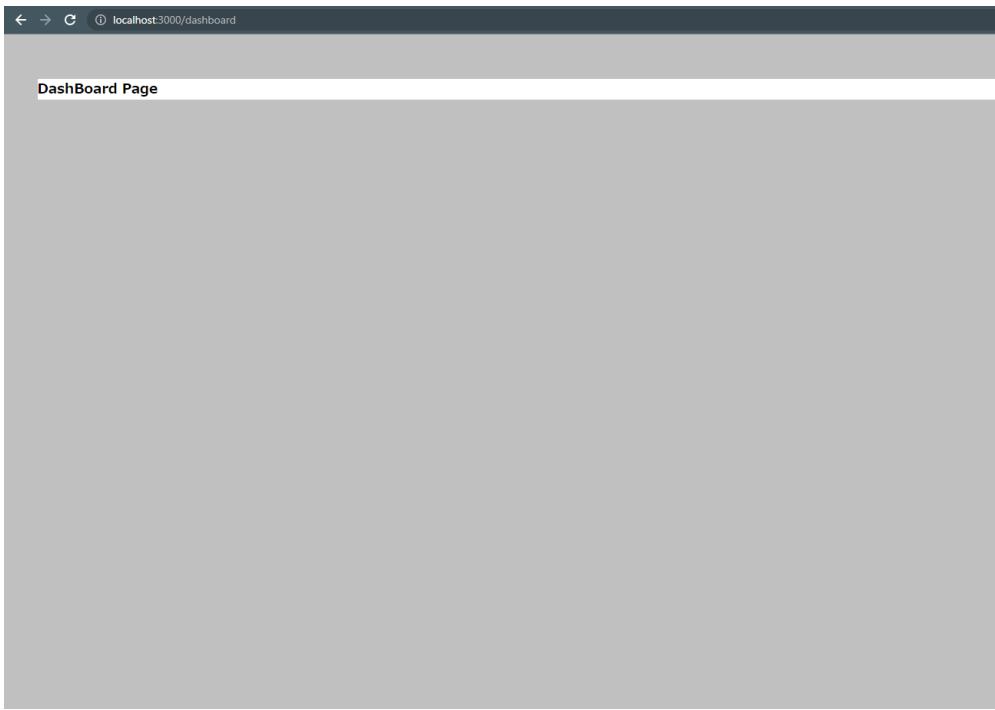
```
export default function RootLayout({
 children,
}: {
 children: React.ReactNode;
}) {
 return (
 <html>
 <head />
 <body
 style={{
 backgroundColor: '#COCOCO',
 padding: '50px',
 }}
 >
 {children}
 </body>
 </html>
);
}
```

```
}
```

app/dashboard/layout.tsx

```
export default function DashboardLayout({
 children,
}: {
 children: React.ReactNode;
) {
 return (
 <section
 style={{
 backgroundColor: 'white',
 }}
 >
 {children}
 </section>
);
}
```

ページの見た目が画像のようになってればOK。



▲ 図 5.3 DashBoard ページ

## 5.4 React Server Components

React Server Components(RSC) は React18 で追加された機能でクライアントとサーバ側が協調してアプリケーションをレンダリングできる機能です。これによりコンポーネントごとに最適なレンダリング方法を選択できるようになります。例えばデータの取得はサーバ側で行い、ユーザの操作によって変わる部分はクライアント側でレンダリングできます。これも公式ドキュメントの図がわかりやすいので貼っておきます。

What do you need to do?	Server Component	Client Component
Fetch data. Learn more.	✓	⚠
Access backend resources (directly)	✓	✗
Keep sensitive information on the server (access tokens, API keys, etc)	✓	✗
Keep large dependencies on the server / Reduce client-side JavaScript	✓	✗
Add interactivity and event listeners ( <code>onClick()</code> , <code>onChange()</code> , etc)	✗	✓
Use State and Lifecycle Effects ( <code>useState()</code> , <code>useReducer()</code> , <code>useEffect()</code> , etc)	✗	✓
Use browser-only APIs	✗	✓
Use custom hooks that depend on state, effects, or browser-only APIs	✗	✓
Use <a href="#">React Class components</a>	✗	✓

▲ 図 5.4 Sever Component と Client Components の違い

また SSR との違いとしてクライアント側の JavaScript の量を減らせる点です。SSR の場合ハイドレーション（サーバ側で生成した DOM とクライアントで生成した DOM を合成する）というステップがありページを早く表示できてもクライアント側でも同じ処理が走るため JavaScript の量は同じでした。RSC はサーバ側でレンダリングした後残りをクライアント側でレンダリングします。これによってクライアント側に送信される JavaScript の量を減らせます。

## 5.5 サーバーコンポーネントでデータを取得

app ディレクトリ内のコンポーネントはデフォルトだとサーバコンポーネントになっています。以下のコードはサーバサイドで qiita の記事リスト取得して表示するものです。dashboard/page.tsx を書き換えます。

dashboard/page.tsx

```
type Article = {
 id: number;
 title: string;
};

async function getArticle(): Promise<Article[]> {
 const res = await fetch('https://qiita.com/api/v2/items?'
 page=1&per_page=24');

 if (!res.ok) {
 throw new Error('Failed to fetch data');
 }

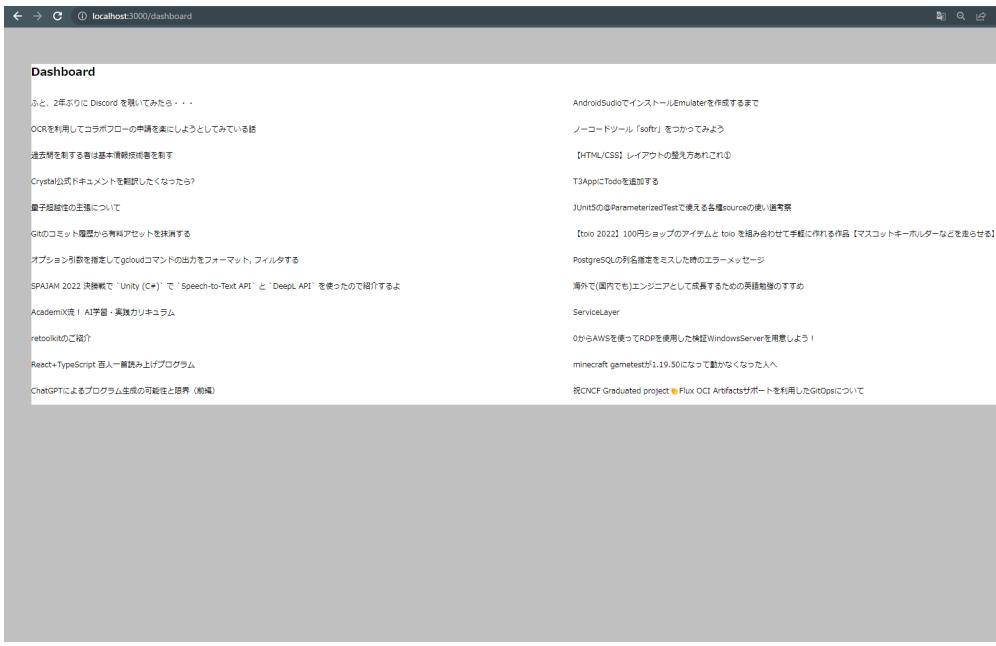
 return res.json();
}

export default async function DashBoardPage() {
 const articles = await getArticle();

 return (
 <div>
 <h1>Dashboard</h1>
 </div>
);
}
```

```
<div
 style={{
 display: 'flex',
 flexDirection: 'column',
 flexWrap: 'wrap',
 height: '50vh',
 }}
>
{articles?.map((article) => (
 <div
 key={article.id}
 style={{
 display: 'flex',
 gap: '10px',
 }}
 >
 <p>{article.title}</p>
 </div>
)));
</div>
</div>
);
}
```

画像のように表示される。



▲ 図 5.5 API からデータ取得後のページ

サイトにカーソルを合わせて右クリックしてページのソースを表示をクリックしてみましょう。あらかじめデータが入った状態でサーバから送られてくるのでページソースに記事データが表示されています。



▲ 図 5.6 ページのソース表示

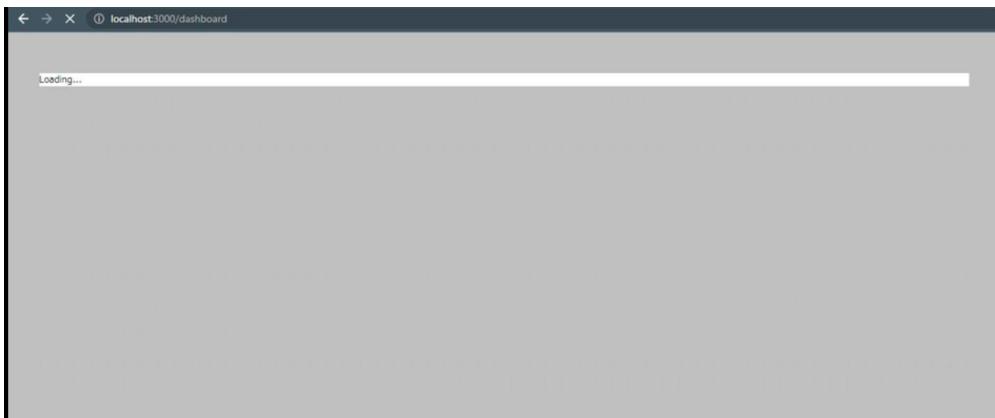
## 5.6 ローディング UI 表示

次はローディング UI を表示する機能です。dashborad ディレクトリに loading.tsx を作成します。

### loading.tsx

```
export default function Loading() {
 return <p>Loading...</p>;
}
```

データの取得が終わり、page コンポーネントがレンダリングされるまでの間は Loading コンポーネントが表示されます。



▲ 図 5.7 ローディング UI

これは React18 で追加された Suspense という機能が使われています。Suspense について説明するとコンポーネントが表示されるまでの状態を指定できるコンポーネントです。非同期的なコンポーネントの場合レンダリングに時間がかかるためその間に何を表示させるかを Suspense を使うと指定できるようになります。

具体的な使用例を上げます。下のコードはデータフェッチライブラリ React Query を使ったデータ取得と表示のサンプルです。

```
import { QueryClient, QueryClientProvider, useQuery } from
 'react-query';
```

```
const queryClient = new QueryClient();

export default function App() {
 return (
 <QueryClientProvider client={queryClient}>
 <Example />
 </QueryClientProvider>
);
}

export function Loading() {
 return <p>Loading...</p>;
}

function Example() {
 const { isLoading, data } = useQuery('repoData', () =>
 fetch('https://api.github.com/repos/tannerlinsley/react-
query').then(
 (res) => res.json()
)
);

 if (isLoading) return <Loading />;

 return (
 <div>
 <h1>{data.name}</h1>
 <p>{data.description}</p>
 </div>
);
}
```

```
);
}
```

Suspense を使えば下のコードに置き換えられます。

```
import { Suspense } from 'react';
import { QueryClient, QueryClientProvider, useQuery } from '
 react-query';

const queryClient = new QueryClient();

export default function App() {
 return (
 <QueryClientProvider client={queryClient}>
 {/* コンポーネントでラップ suspense */}
 <Suspense fallback={<Loading />}>
 <Example />
 </Suspense>
 </QueryClientProvider>
);
}

export function Loading() {
 return <p>Loading...</p>;
}

function Example() {
 const { data } = useQuery('repoData', () =>
 fetch('https://api.github.com/repos/tannerlinsley/react-
```

```
query').then(
 (res) => res.json()
)
);
//ローディングプロパティによる表示分岐の削除

return (
 <div>
 <h1>{data.name}</h1>
 <p>{data.description}</p>
 </div>
);
}
```

```
<Layout>
<Header/>
<SideNav/>
<Suspense fallback={<Loading />}>
<DashBoardPage />
```

```
</Suspense>
</Layout>
```

ディレクトリ内に loading.tsx を配置すると、自動的にこのコードを記述した動作が実現されます。

## 5.7 エラーハンドリング

次にエラーハンドリングを見ていきます。dashboard ディレクトリに error.tsx を作成します。

dashboard/error.tsx

```
'useClient';

import { useEffect } from 'react';

export default function Error({
 error,
 reset,
}: {
 error: Error;
 reset: () => void;
}) {
 useEffect(() => {
 // Log the error to an error reporting service
 console.error(error);
 }, [error]);
}

return (

```

```
<div>
 <p>Something went wrong!</p>
 <button onClick={() => reset()}>Reset error boundary</
 button>
</div>
);
}
```

公式ドキュメントによると error.tsx はクライアントコンポーネントである必要があり use client; と書けばクライアントコンポーネントして利用するできるようです。loading.tsx と同じようにファイルを置いておけば自動的に Page をネストしてエラーハンドリングをしているようです。

普通に書いた場合

dashboard/error.tsx

```
<Layout>
 <Header/>
 <SideNav/>
 <Suspense fallback={<Loading />}>
 <DashBoardPage />
 </Suspense>
</Layout>
```

ただこれを見てわかる通り page をラップしてるので同階層のコンポーネントのエラーハンドリングはできない感じですね。したいならより上の階層で ErrorBoundary を使う必要がありそうです。

実際にコードを書き換えてエラーを出してみます。存在しない URL 'hoge' を指定し getArticle でしていたエラーハンドリングを削除しています。

## dashboard/page.tsx

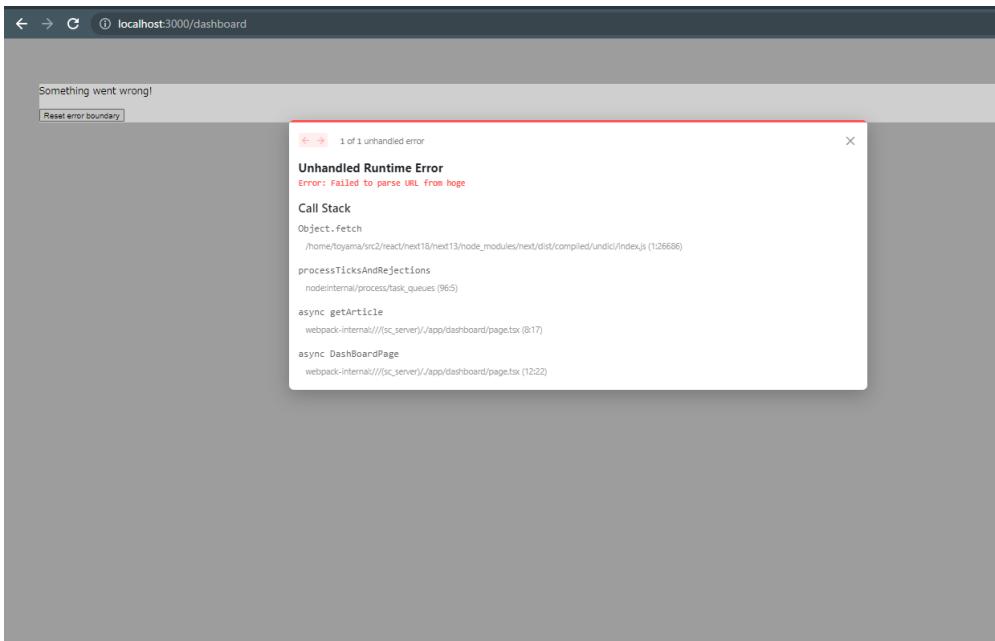
```
'useClient';

import { useEffect } from 'react';

export default function Error({
 error,
 reset,
}: {
 error: Error;
 reset: () => void;
}) {
 useEffect(() => {
 // Log the error to an error reporting service
 console.error(error);
 }, [error]);
}

return (
 <div>
 <p>Something went wrong!</p>
 <button onClick={() => reset()}>Reset error boundary</button>
 </div>
);
}
```

エラーの内容とエラーページが表示されていますね。



▲ 図 5.8 エラーハンドリング画面

## 5.8 終わりに

感想としては、Next.js 13 の機能は React18 の Suspense や RSC などの機能に合わせたアップデートというのを強く感じました。

# 6

これは chapter

## 6.1 これは section

我輩は猫である<sup>\*1</sup>。

どこで生れたかとんと見当がつかぬ。何でも薄暗いじめじめした所でニヤーニヤー泣いていた事だけは記憶している。吾輩はここで始めて人間というものを見た。しかもあとで聞くとそれは書生という人間中で一番禰惡な種族であったそうだ。この書生というのは時々我々を捕えて煮て食うという話である。

```
1 /* ここにはソースコードを書く */
2 #include<stdio.h>
3
4 int main(void)
5 {
```

---

<sup>\*1</sup> こんな感じで脚注を書く

```
6 printf("Hello, World!\n");
7 return 0;
8 }
9 /* breakable を付けるとこんな感じで改行にも対応できる */
```

```
ここにはコマンドを書く
$ echo "Hello, World!"
```

図表はキャプションを付けたときに、先頭に「▲」や「▼」を付けるようにした。

▼ 表 6.1 表のサンプル

日本	hoge	fuga	piyo
アメリカ	foo	bar	baz



▲ 図 6.1 画像のサンプル

これはコラム

コラムも隨時挟めるようにした。

tcolorbox は title を指定するといい感じにタイトル付きの枠で囲ってくれる。

# 7

これは chapter

## 7.1 これは section

我輩は猫である<sup>\*1</sup>。

どこで生れたかとんと見当がつかぬ。何でも薄暗いじめじめした所でニャーニャー泣いていた事だけは記憶している。吾輩はここで始めて人間というものを見た。しかもあとで聞くとそれは書生という人間中で一番禰惡な種族であったそうだ。この書生というのは時々我々を捕えて煮て食うという話である。

```
1 /* ここにはソースコードを書く */
2 #include<stdio.h>
3
4 int main(void)
5 {
```

<sup>\*1</sup> こんな感じで脚注を書く

```
6 printf("Hello, World!\n");
7 return 0;
8 }
9 /* breakable を付けるとこんな感じで改行にも対応できる */
```

```
ここにはコマンドを書く
$ echo "Hello, World!"
```

図表はキャプションを付けたときに、先頭に「▲」や「▼」を付けるようにした。

▼ 表 7.1 表のサンプル

日本	hoge	fuga	piyo
アメリカ	foo	bar	baz



▲ 図 7.1 画像のサンプル

これはコラム

コラムも隨時挟めるようにした。

tcolorbox は title を指定するといい感じにタイトル付きの枠で囲ってくれる。

# 8

これは chapter

## 8.1 これは section

我輩は猫である<sup>\*1</sup>。

どこで生れたかとんと見当がつかぬ。何でも薄暗いじめじめした所でニヤーニヤー泣いていた事だけは記憶している。吾輩はここで始めて人間というものを見た。しかもあとで聞くとそれは書生という人間中で一番獰惡な種族であったそうだ。この書生というのは時々我々を捕えて煮て食うという話である。

```
1 /* ここにはソースコードを書く */
2 #include<stdio.h>
3
4 int main(void)
5 {
```

<sup>\*1</sup> こんな感じで脚注を書く

```
6 printf("Hello, World!\n");
7 return 0;
8 }
9 /* breakable を付けるとこんな感じで改行にも対応できる */
```

```
ここにはコマンドを書く
$ echo "Hello, World!"
```

図表はキャプションを付けたときに、先頭に「▲」や「▼」を付けるようにした。

▼ 表 8.1 表のサンプル

日本	hoge	fuga	piyo
アメリカ	foo	bar	baz



▲ 図 8.1 画像のサンプル

これはコラム

コラムも隨時挟めるようにした。

tcolorbox は title を指定するといい感じにタイトル付きの枠で囲ってくれる。