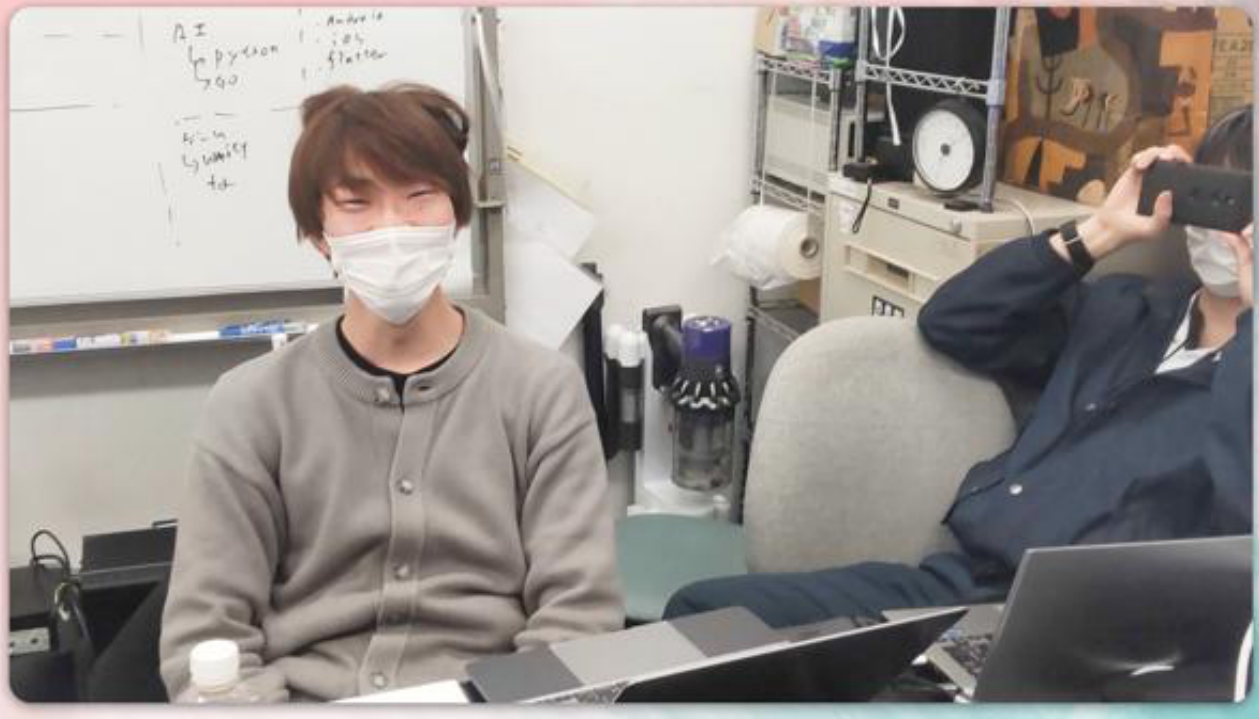
シス研の技術本
テスト作成
表紙



目次

1 これは chapter

1.1 これは section

我輩は猫である[*1]。

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獰悪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  {
```

[*1] こんな感じで脚注を書く

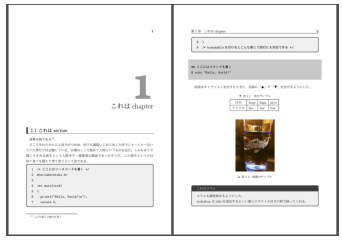
```
    printf("Hello, World!\n");
    return 0;
}
/* breakable を付けるとこんな感じで改行にも対応できる */
```

```
## 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 1.1　表のサンプル

日本	hoge	fuga	piyo
アメリカ	foo	bar	baz



▲ 図 1.1　画像のサンプル

これはコラム

コラムも随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。

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 これは chapter

3.1 これは section

我輩は猫である[*1]。

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獰悪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  {
```

[*1] こんな感じで脚注を書く

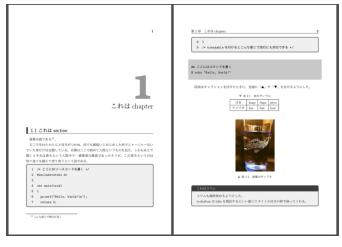
```
6    printf("Hello, World!\n");
7    return 0;
8  }
9  /* breakable を付けるとこんな感じで改行にも対応できる */
```

```
## 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 3.1　表のサンプル

日本	hoge	fuga	piyo
アメリカ	foo	bar	baz



▲ 図 3.1　画像のサンプル

これはコラム

コラムも随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。

4

これは chapter

4.1 これは section

我輩は猫である[*1]。

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獰悪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  {
```

[*1] こんな感じで脚注を書く

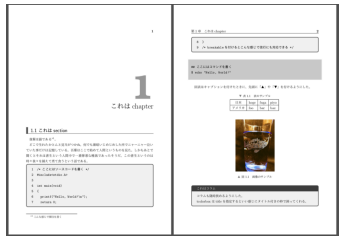
```
    printf("Hello, World!\n");
    return 0;
}
/* breakable を付けるとこんな感じで改行にも対応できる */
```

```
## 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 4.1　表のサンプル

日本	hoge	fuga	piyo
アメリカ	foo	bar	baz



▲ 図 4.1　画像のサンプル

これはコラム

コラムも随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。

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か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.

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

app/page.tsx を作り, 以下のようにする.

```
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.

```
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.

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

この方法だとページごとにレイアウトを変えることができません. 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 を置くことでページごとにレイアウトを変えることができるようになりました.

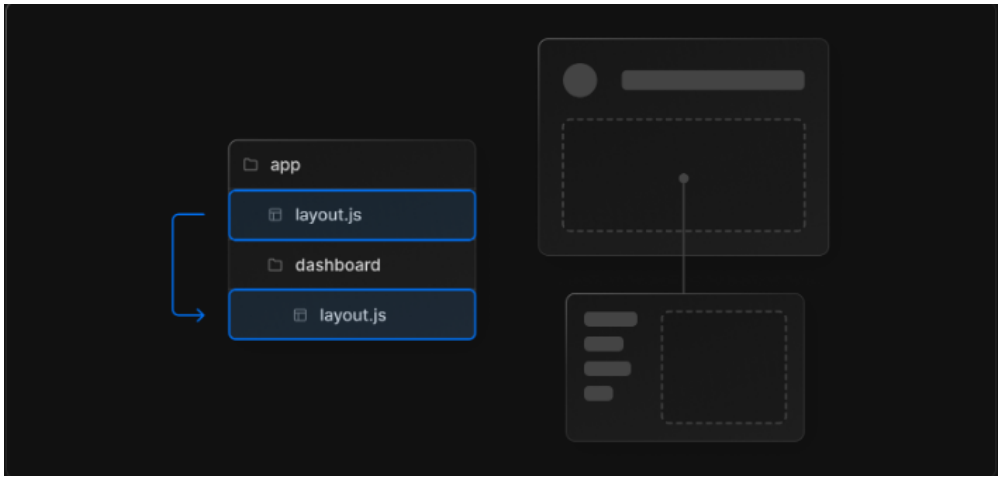
ここではダッシュボードページのレイアウトの場合を考えます. app ディレクトリ

の中で dashbord 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.

```
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. しかしそんなことはなく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.

```
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

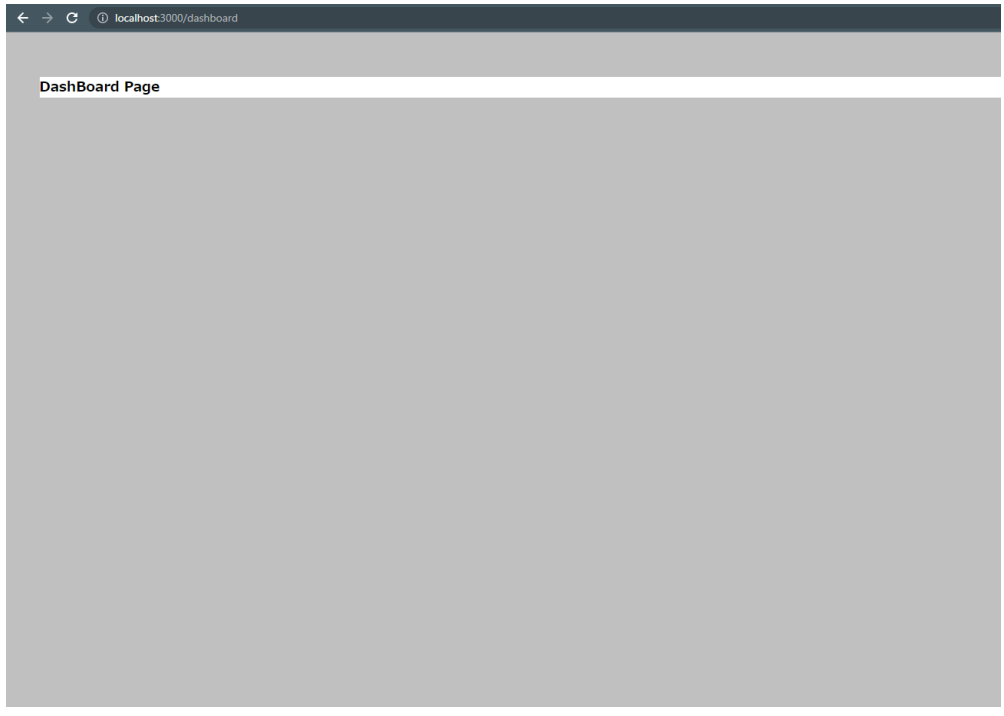
app/layout.tsx

```
export default function RootLayout({
    children,
  }: {
children: React.ReactNode;
}) {
return (
<html>
<head />
<body
style={{
    backgroundColor: '#C0C0C0',
    padding: '50px',
  }}
>
  {children}
</body>
</html>
);
}
```

app/dashboard/layout.tsx

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

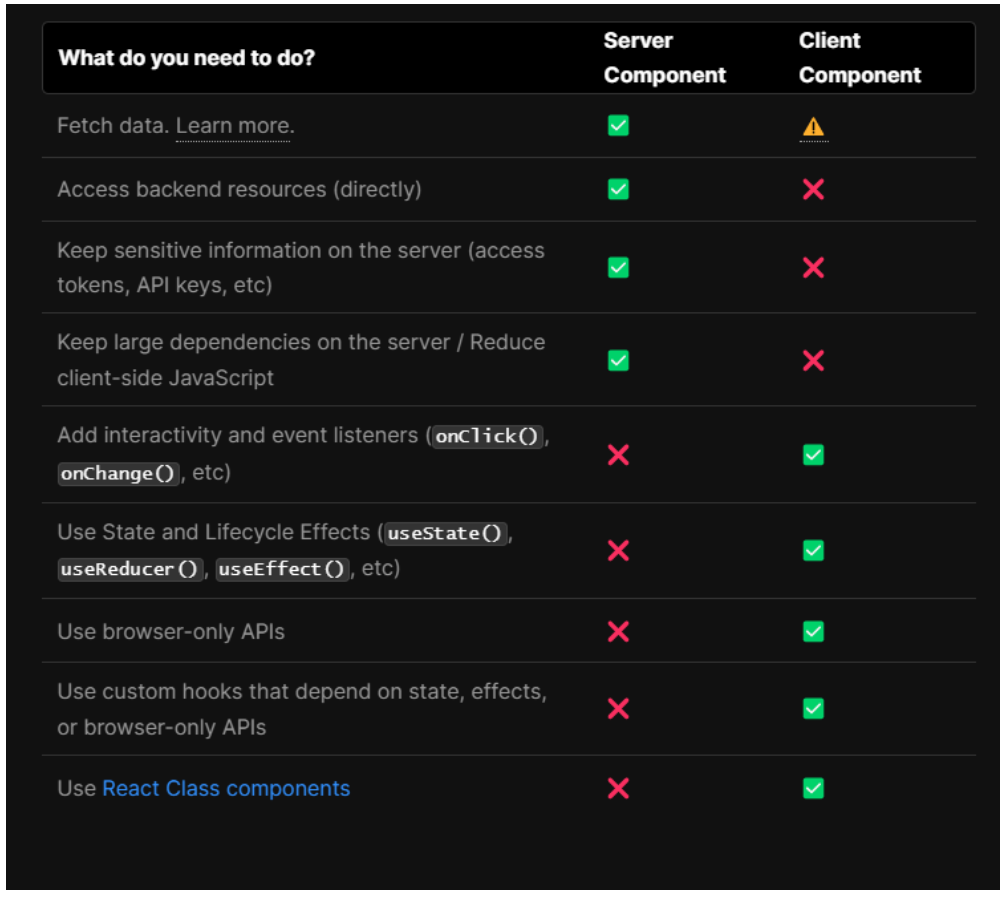
ページの見た目が画像のようになってれば OK



▲ 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するといったことができます. 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.



▲ 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 の量を減らすことができます.

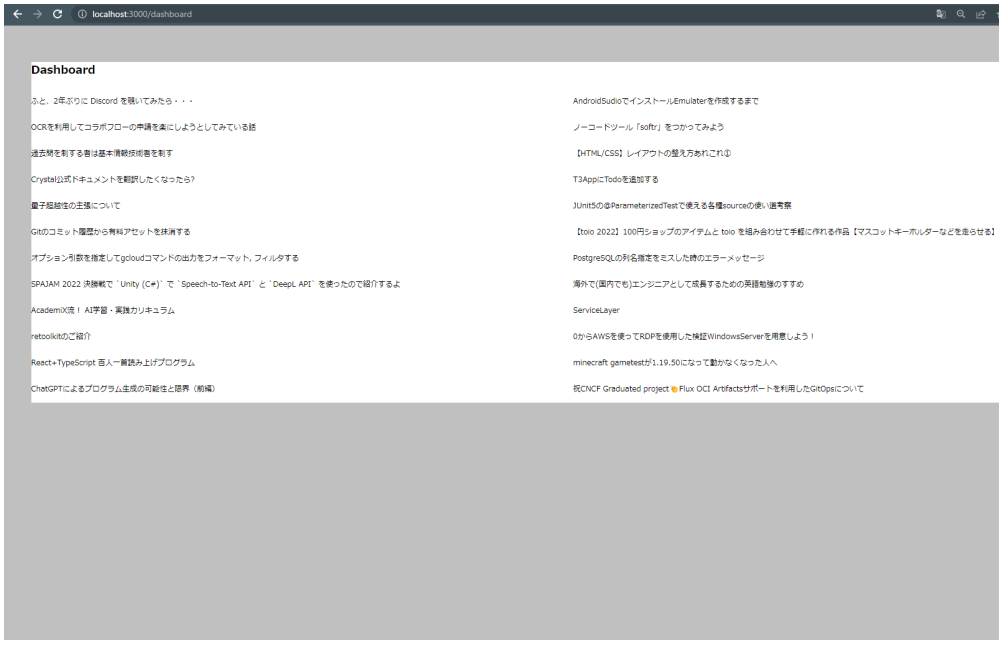
5.5 サーバーコンポーネントでデータを取得

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.

```
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
```

```
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.

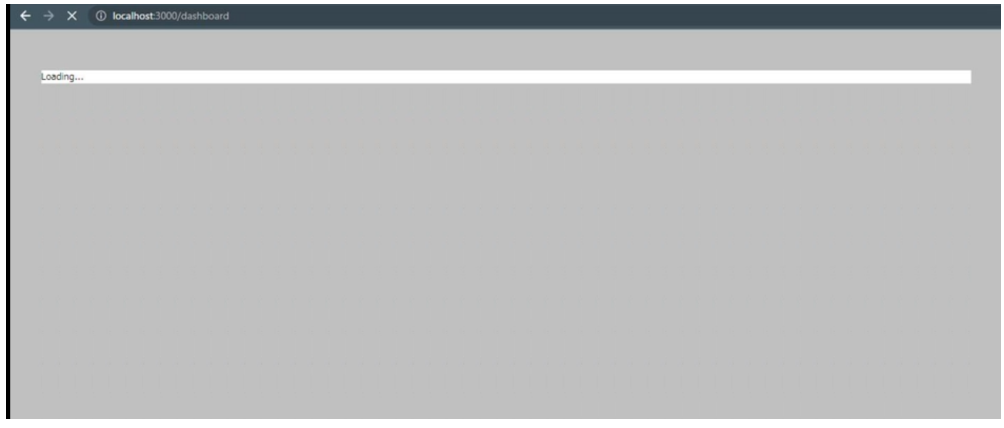
▲ 図 5.6　ページのソース表示画面

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

```
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てます. 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することが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ることができます

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

const queryClient = new QueryClient();

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

変更点は Example コンポーネントを Suspense コンポーネントでラップしているのと,Example コンポーネントの中の isLoading プロパティを使った表示切替の部分が消えているところです.Suspense のいいところはデータ取得をするコンポーネントの中で表示を切り替える処理を書く必要がないことです. これによってより宣言的なコードになりました. またコンポーネントの責務の観点から見てもローディング完了時の表示だけでよくシンプルになっています.

Next.js 13 では loading.tsx を置いてあげると Next.js 側がそれを読み取り Page コンポーネントを Suspense コンポーネントでラップしてくれるようです. 普通に書くと以下のようになります.

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

ディレクトリ内に loading.tsx を置いておけば自動的にこのコードを書いた動きに

なるということですね.

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.

```
'use client';

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; と書くことでクライアントコンポーネントして利用することができるようです. 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

```
    <Layout>
    <Headr/>
    <SideNav/>
    <ErrorBoundary fallback={<Loading />}>
      <DashBoardPage />
    </ErrorBoundary>
  </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.

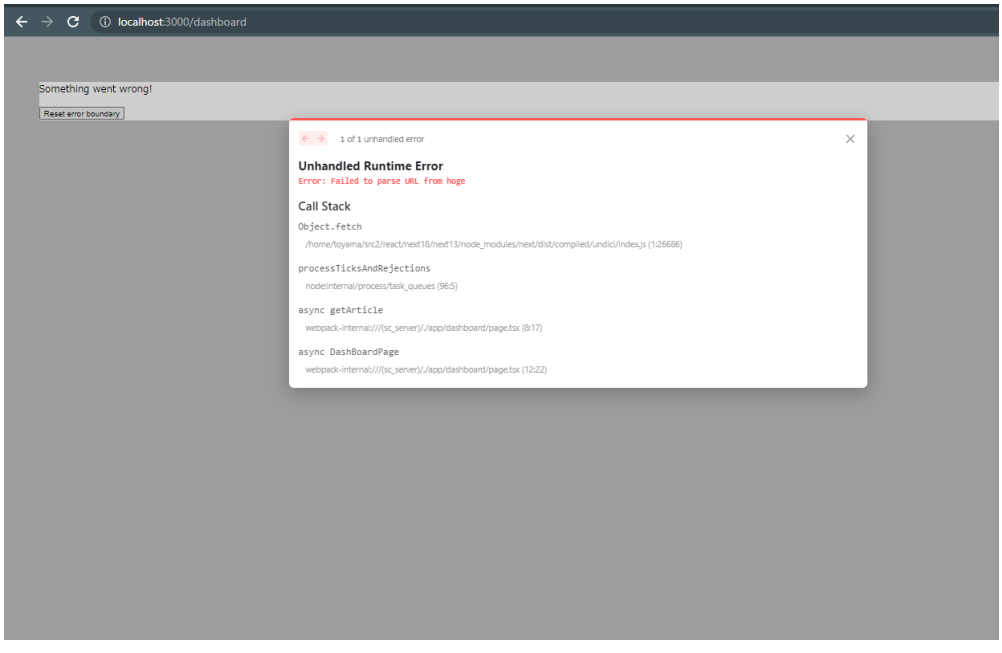
実際にコードを書き換えてエラーを出してみます. 存在しない URL'hoge' を指定し getArticel でしていたエラーハンドリングを削除しています.

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

```
async function getArticle(): Promise<Article[]> {
    const res = await fetch('hoge');

    return res.json();
  }

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

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

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 Supense 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[*1]。

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獰悪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  {
```

[*1] こんな感じで脚注を書く

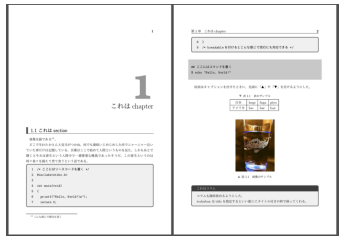
```
    printf("Hello, World!\n");
    return 0;
}
/* breakable を付けるとこんな感じで改行にも対応できる */
```

```
## 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

コラムも随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[*1]。

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獰悪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  {
```

[*1] こんな感じで脚注を書く

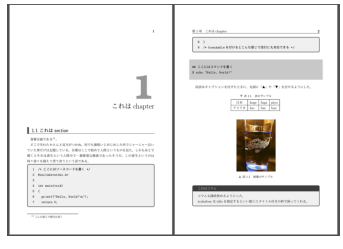
```
    printf("Hello, World!\n");
    return 0;
}
/* breakable を付けるとこんな感じで改行にも対応できる */
```

```
## 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

コラムも随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[*1]。

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獰悪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  {
```

[*1] こんな感じで脚注を書く

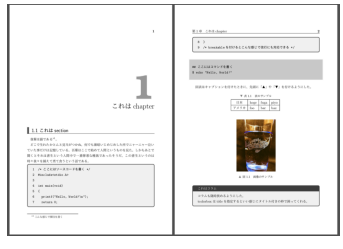
```
    printf("Hello, World!\n");
    return 0;
}
/* breakable を付けるとこんな感じで改行にも対応できる */
```

```
## 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

コラムも随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。