



SPA

Hands-on

Service Dev Engineer TAIKEN

ハンズオン資料のダウンロードのお願い

SPAハンズオンに参加頂き、ありがとうございます。

本日使用するハンズオン資料のダウンロードをお願いします。

ダウンロードURLはZoomのチャットで連絡します。

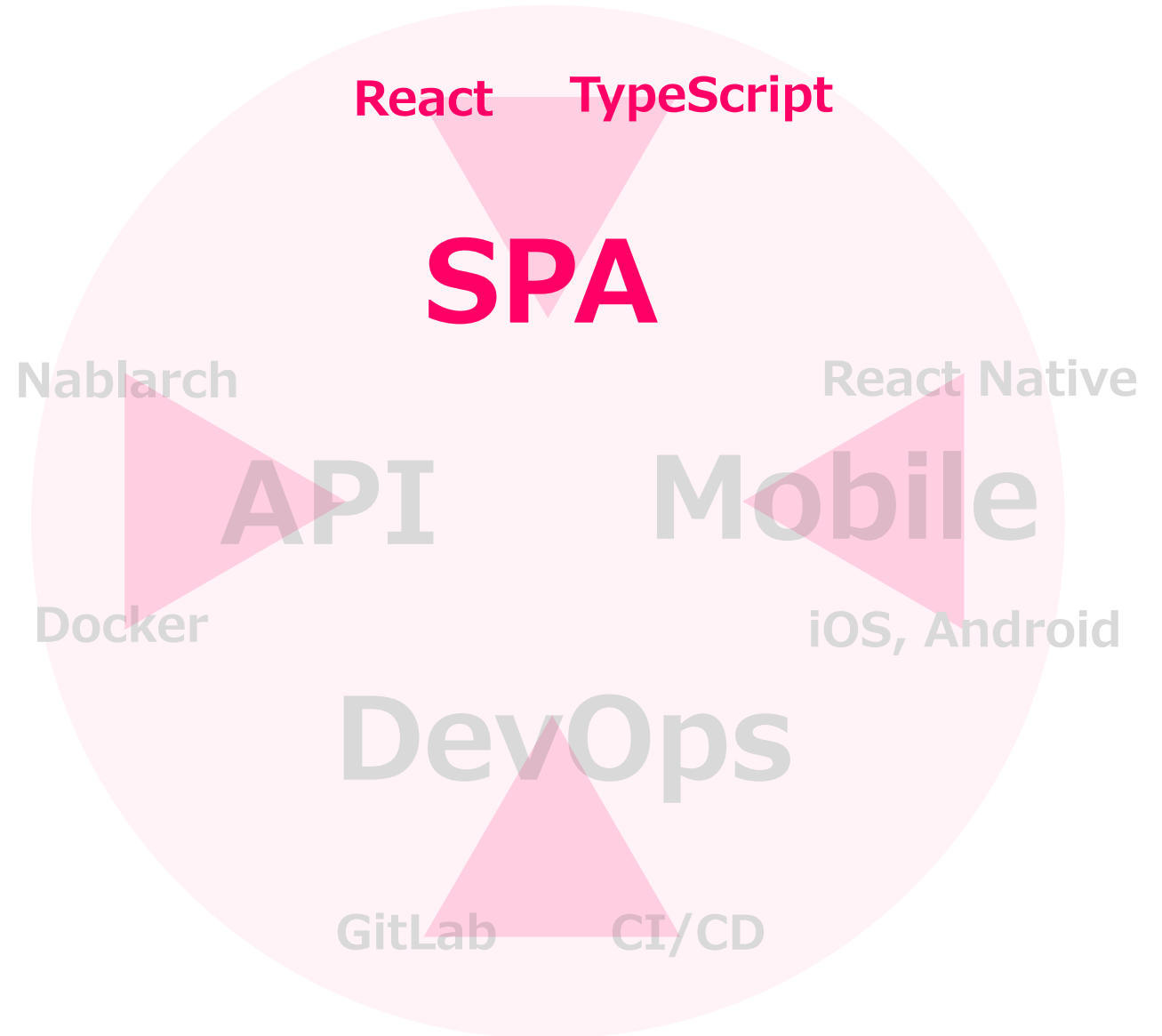
ダウンロード後、任意の場所でzipファイルの解凍をお願いします。



サービス開発エンジニア体験

サービス/プロダクトの開発に欠かせない
アプリ開発とDevOpsを体験してみませんか？

10月 SPAハンズオン
11月 APIハンズオン
12月 モバイルハンズオン
1月 DevOpsハンズオン
2月 腕試しハッカソン



スタッフ紹介

TIS株式会社

テクノロジー & イノベーション本部

テクノロジー & エンジニアセンター

伊藤 清人@会津若松

世古 雅也@会津若松

西日本テクノロジー & イノベーション室

斎藤さん@会津若松



TODO : 斎藤さんの名前と写真

お願い

Zoomで名前（ニックネームも可）を分かるようにしてください。

オフライン＋オンライン開催なのでリアクションは大きな動きをお願いします！

オンラインの人は周りの音が入り込まないようにお願いします。

本で行うハンズオンはFintanで公開している
ハンズオン資料をベースにしています。

<https://fintan.jp/>

今後の改善等に活用したいので

ハンズオン終了後のアンケートにご協力をお願いします。



Reactを使ったSPAの作り方を学ぶハンズオン

ハンズオンのゴール

SPAの作り方を体験するがゴールです。

ReactやTypeScriptの仕様や使い方は細かく説明しないです。（質問はしても大丈夫です！）

SPAの開発に必要な技術要素をできるだけ多く体験できるように構成しています。

そのため、じっくりコーディングするより、ショートカットしてどんどん進めていくハンズオンです。

皆さんが作業しただけにならず、皆さんにSPAの作り方を持ち帰ってもらえるように頑張ります！

React
TypeScript < SPAの作り方 ⇒ 体験

ハンズオンの進め方

スタッフが説明しながら作業→参加者も作業・・・といったかたちでStep by Stepで進行します。
皆さんの作業状況を確認しながら進めますのでリアクションをお願いします。

つまった場合は声をかけてください。画面共有して問題解消にあたります。
質問は随時受け付けます。

ハンズオンのスケジュール（全体180分）

- 05m オープニング
- 05m SPA入門
- 05m ハンズオンの題材
- 05m ハンズオン資料の準備



開発準備（20分）

- 05m プロジェクトの作成
- 05m クライアントコードの生成
- 05m モックサーバーの起動
- 05m APIクライアントの作成



ToDo管理の開発（90分）

- 15m ページ外観の作成
- 15m コンポーネントの分割
- 30m ToDoの一覧表示
- 30m ToDoの登録



ユーザ認証の開発（45分）

- 05m ページ外観の作成
- 10m URLルーティングの設定
- 15m ユーザーコンテキストの作成
- 15m サインアップ～ログアウト
（できるところまで）



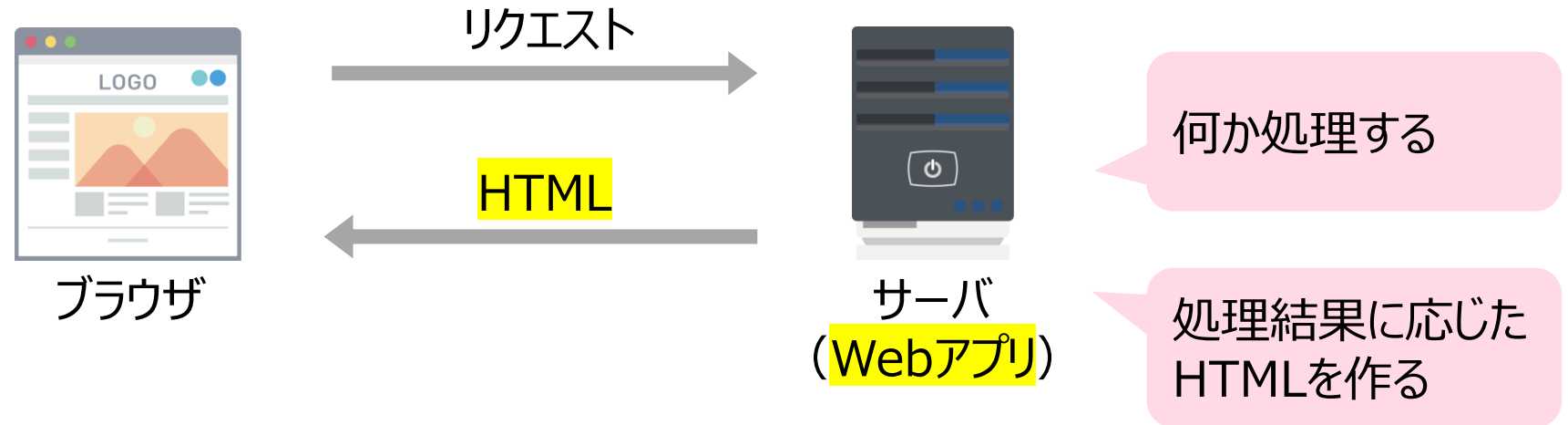
- 05m クロージング

TODO：最後に見直す。休憩を入れる

SPA入門

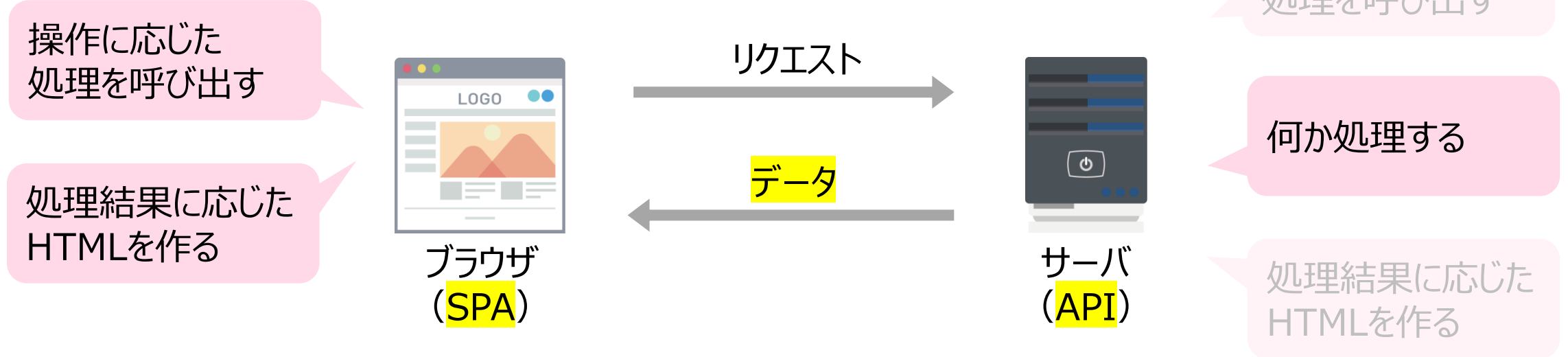
SPA、その前にWebアプリ

リクエストするとHTMLを返します。
Webアプリが画面遷移をコントロールします。



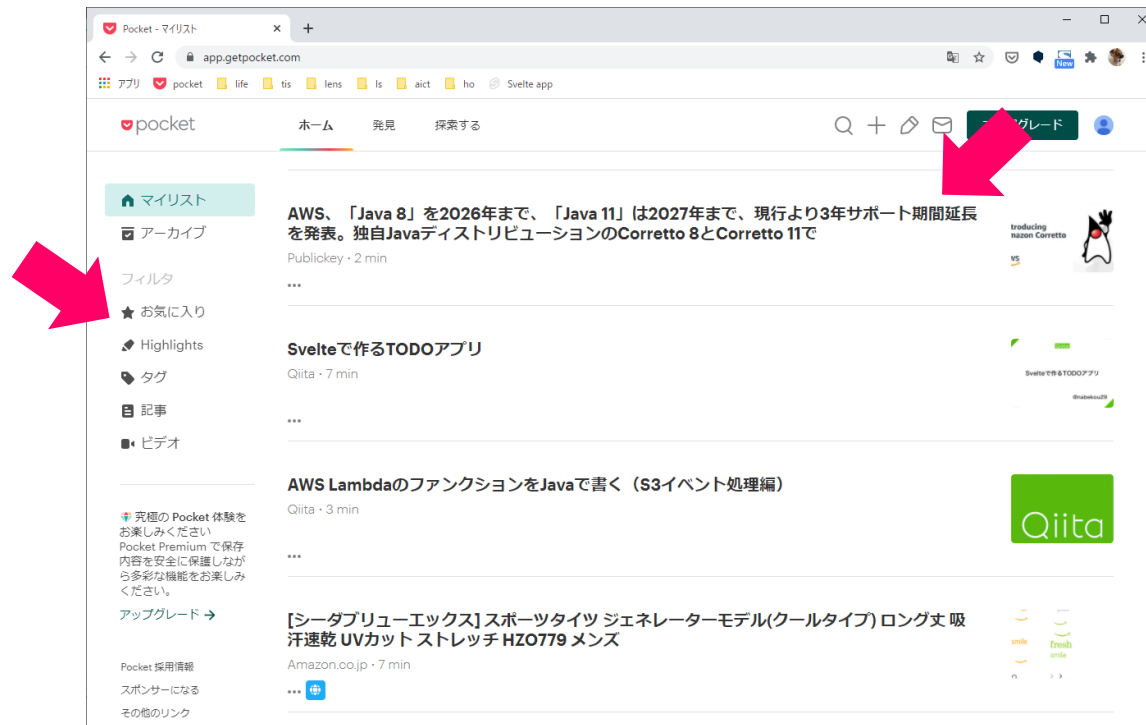
SPAになると

リクエストするとデータが返ってきます。
SPAが画面遷移をコントロールします。



Single Page Application

HTML、JavaScript、CSSを駆使して単一ページでアプリケーションを実現します。



Pocketで左側のメニューを選ぶとコンテンツ部分だけ再描画されます

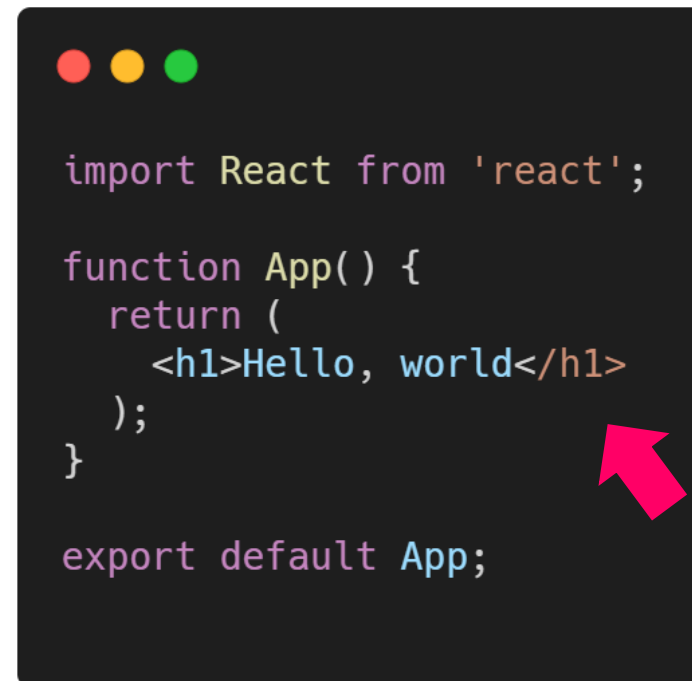
React

SPAを1から作るのは相当大変なのでReactを使います。

ReactはSPAを作るためのフレームワークです。
画面部品(=コンポーネント)、イベントハンドリング、
状態保持、画面遷移(=ルーティング)など、
SPA作成に必要な仕組みを提供してくれます。

日本語の公式サイトもあります。
とても分かりやすいです。

<https://ja.reactjs.org/>



```
import React from 'react';

function App() {
  return (
    <h1>Hello, world</h1>
  );
}

export default App;
```

「Hello, world」と出すReactのコード

ソースコードはcarbonを使用しています。

<https://carbon.now.sh/14>

TypeScript

SPAの大部分はJavaScriptで作ります。

JavaScriptは型がないので苦労します。

動かすまで間違いに気づけないです。

少しでも苦労を和らげたいのでTypeScriptを使い、
JavaScriptに型を導入します。

学習コストに見合うだけの恩恵を受けられます。



```
type Todo = {  
  id: number  
  text: string  
  completed: boolean  
}  
  
export const TodoBoard: React.FC = () => {  
  const [todos] = useState<Todo[]>([  
    { id: 2001, text: '洗い物をする', completed: true },  
    { id: 2002, text: '洗濯物を干す', completed: false },  
    { id: 2003, text: '買い物へ行く', completed: false }  
  ]);  
  
  return (  
    <div className="TodoBoard_content">  
      <TodoForm />  
      <TodoFilter />  
      <TodoList todos={todos}/>  
    </div>  
  );  
};
```

TypeScriptで型を定義しているコード（Todo部分）

Visual Studio Code

TypeScriptの恩恵を受けるためVSCodeを使います。

型に基づいてコード補完や

コードの間違いを教えてください。

```
4
5 type Todo = {
6   id: number,
7   text: string,
8   completed: boolean
9 };
10
11 type Props = {
12   todos: Todo[],
13   toggleTodoCompletion: (id: number) => void
14 };
15
16 export const TodoList: React.FC<Props> = ({todos, toggleTodoCompletion}) => {
17   return (
18     <ul className="TodoList_list">
19       {todos.map(todo =>
20         <TodoItem key={todo.id}
21           id={todo.id}
22           text={todo.text} completed={todo.completed}
23           completed={todo.completed} id={todo.id}
24           toggleTodoCompletion={toggleTodoCompletion} text={todo.text}
25       )}
26     </ul>
27   );
28 }
```



ToDoのプロパティがコード補完されます。number型も分かっています

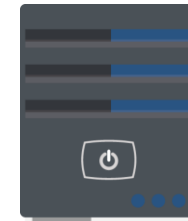
SPAとREST APIの並行開発？

SPAとAPIが独立しているのでうれしい面が多々ありますが、**並行して開発するのが難しく**なります。

APIがないとSPAが動かせない？



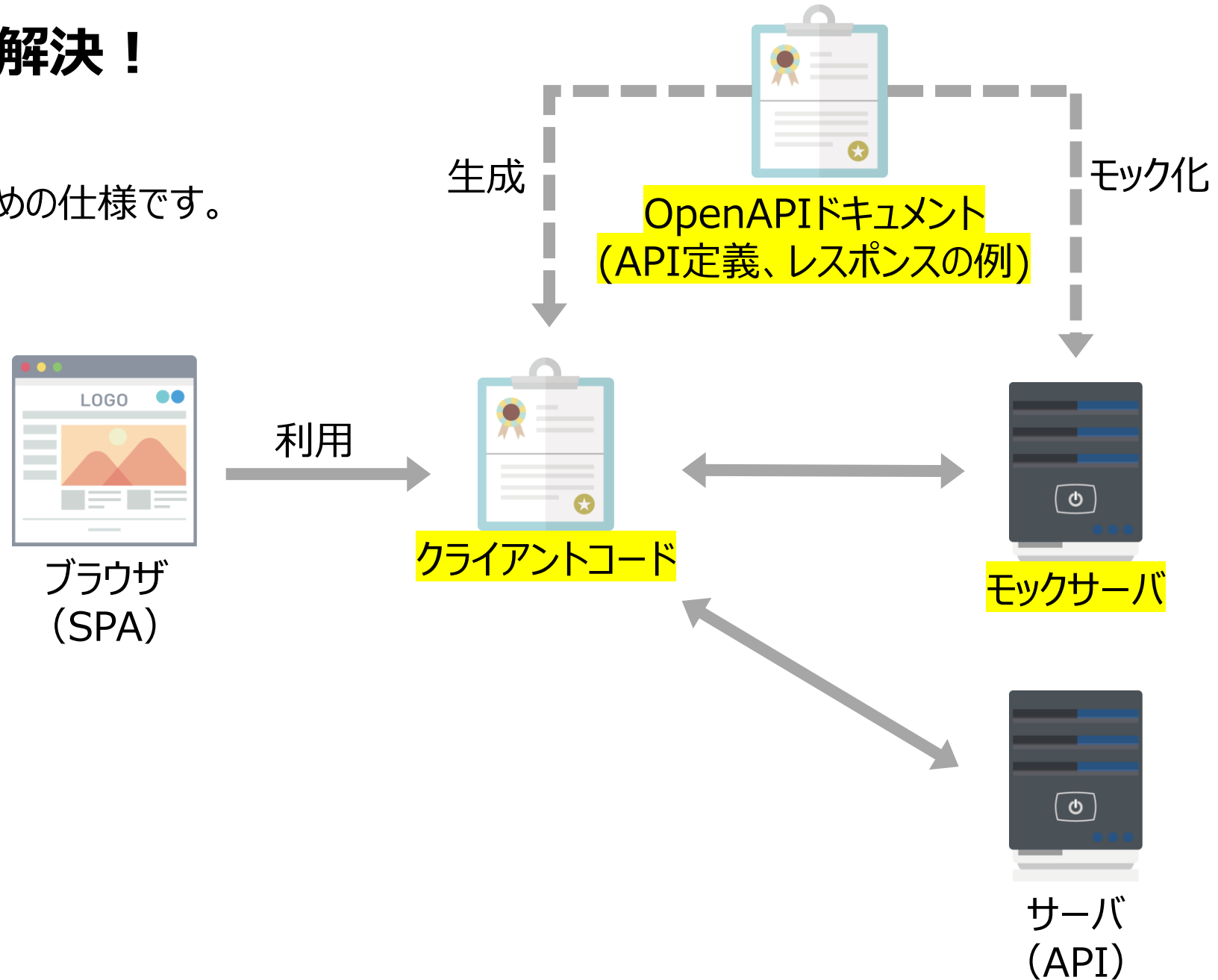
ブラウザ
(SPA)



サーバ
(API)

OpenAPIが解決！

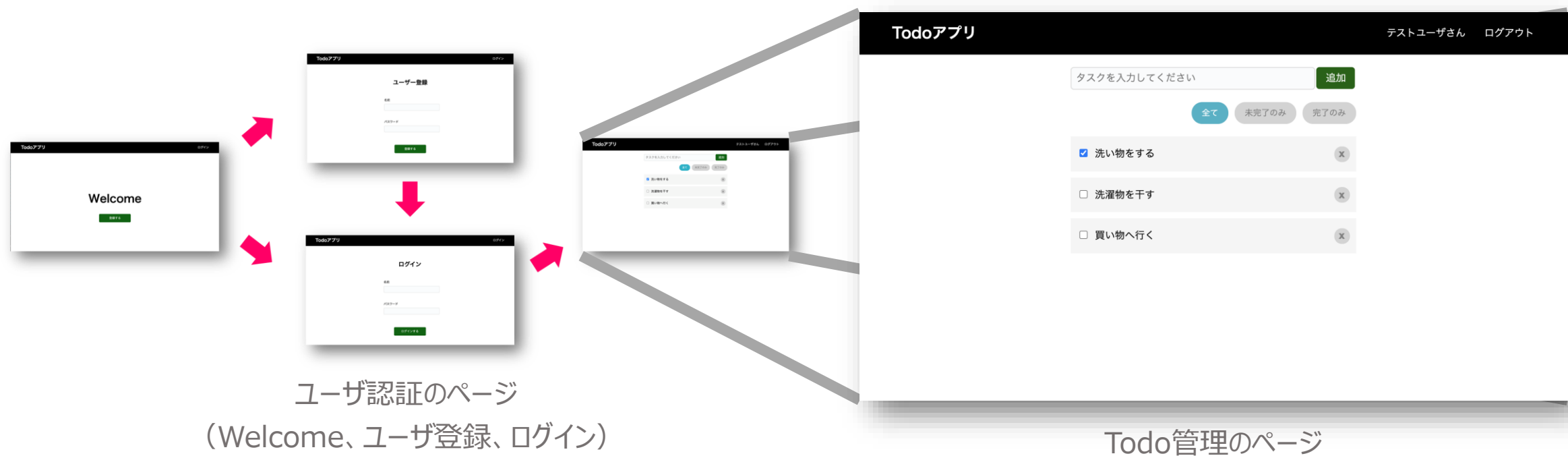
APIを定義するための仕様です。



ハンズオンの題材

ToDoアプリ

ToDoを管理するためのサービスを提供するToDoアプリを作成します。



ハンズオン資料を準備する

このスライドをPDFにしたファイルを提供しています。

PDFファイルを開いて見れる状態にしてください。

[spa-restapi-handson/spa-handson.pdf](#)



開発準備

プロジェクトの作成

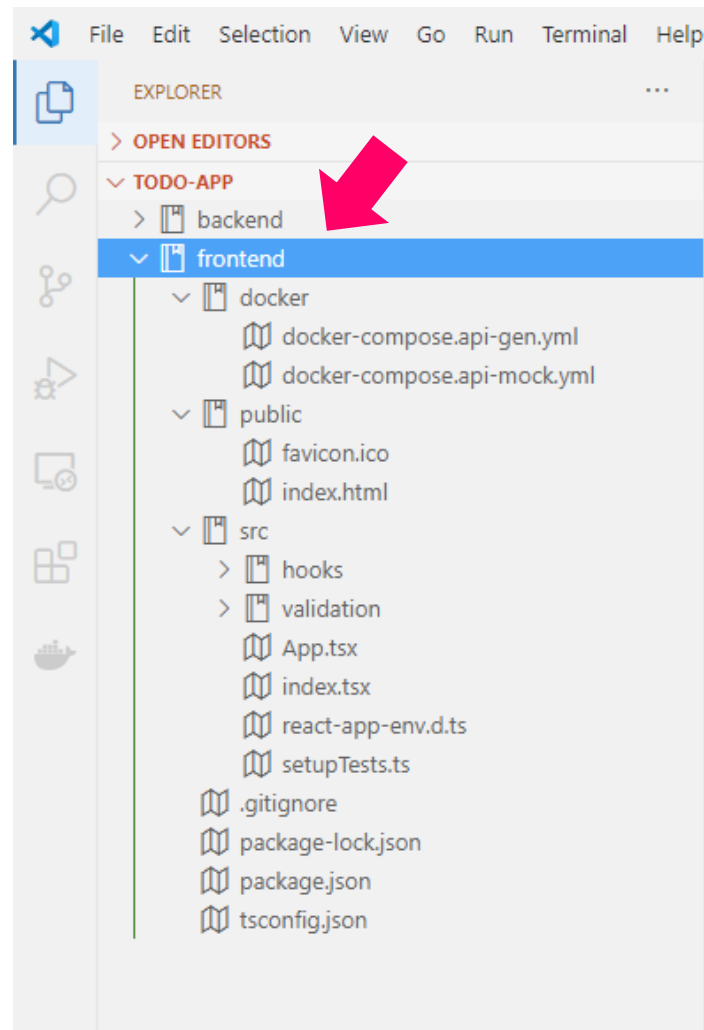
プロジェクトを作成します。

spa-restapi-handson/starter-kitディレクトリを
任意の場所にコピーします。

ディレクトリ名をstarter-kit→todo-appに変更します。

todo-appディレクトリをVSCodeで開きます。

Create React Appというツールでプロジェクトを作成しています。
TypeScript用のテンプレートを使用しています。



クライアントコードの生成

クライアントコードを生成します。

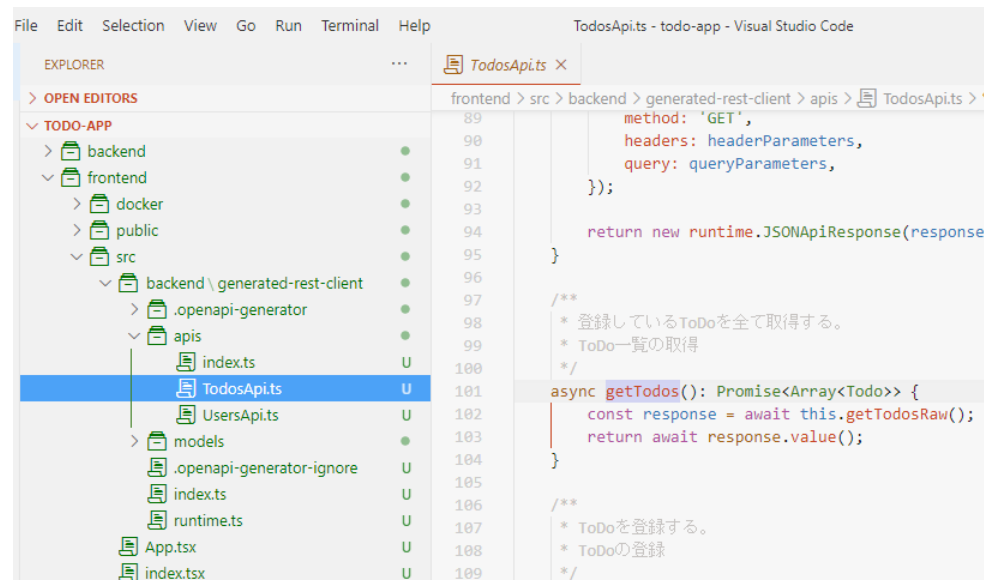
frontendディレクトリで次のコマンドで生成します。

```
$ docker-compose -f docker/docker-compose.api-gen.yml up
```

次のディレクトリに生成されます。

```
src/backend/generated-rest-client/
```

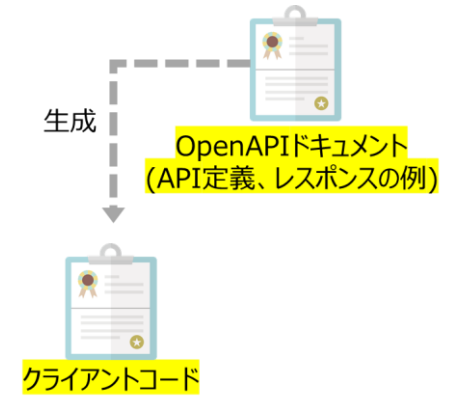
クライアントコードの生成にはOpenAPI Generatorというツールを使用しています。
Dockerコンテナで実行します。



OpenAPIドキュメントとクライアントコードの対応

```
/api/todos:  
get:  
  summary: ToDo一覧の取得  
  description: >  
    登録しているToDoを全て取得する。  
  tags:  
    - todos  
  operationId: getTodos  
  responses:  
    '200':  
      description: OK  
      content:  
        application/json:  
          schema:  
            type: array  
            items:  
              $ref: '#/components/schemas/ToDo'  
          examples:  
            example:  
              value:  
                - id: 2001  
                  text: やること 1
```

tagsでグルーピングします。
コード生成するとグループごとにクラスが作成されます。
operationIdでREST APIを識別するIDを指定します。
コード生成すると関数名に使われます。



```
TodosApi.ts X  
frontend > src > backend > generated-rest-client > apis > TodosApi.ts > 1  
43 *  
44 */  
45 export class TodosApi extends runtime.BaseAPI {  
46  
47   /**  
48    * 登録しているToDoを削除する。  
49    * ToDo  
50    */  
51   async de  
96  
97   /**  
98    * 登録しているToDoを全て取得する。  
99    * ToDo一覧の取得  
100   */  
101   async getTodos(): Promise<Array<ToDo>> {  
102     const response = await this.getTodosRaw();  
103     return await response.value();  
104   }  
105
```

モックサーバが返すレスポンス

```
/api/todos:  
get:  
  summary: ToDo一覧の取得  
  description: >  
    登録しているToDoを全て取得する。  
  tags:  
    - todos  
  operationId: getTodos  
  responses:
```

examplesに実際に返却される例を定義します。
モックサーバが返却するデータになります。

```
    application/json:  
      schema:  
        type: array  
        items:  
          $ref: '#/components/schemas/ToDo'  
      examples:  
        example:  
          value:  
            - id: 2001  
              text: やること 1  
              completed: true  
            - id: 2002  
              text: やること 2  
              completed: false  
      '403':  
        description: Forbidden
```

```
{  
  [  
    {  
      "id": 2001,  
      "text": "やること 1",  
      "completed": true  
    },  
    {  
      "id": 2002,  
      "text": "やること 2",  
      "completed": false  
    }  
  ]  
}
```



モックサーバの起動

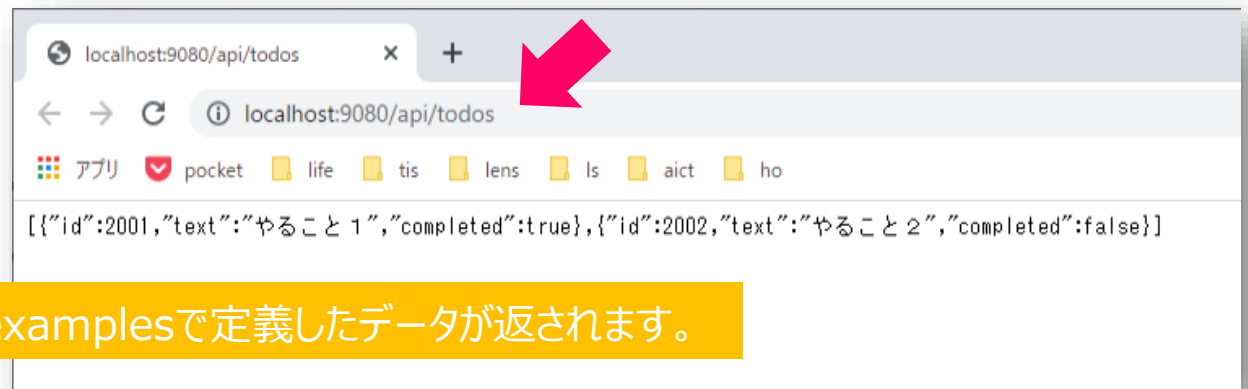
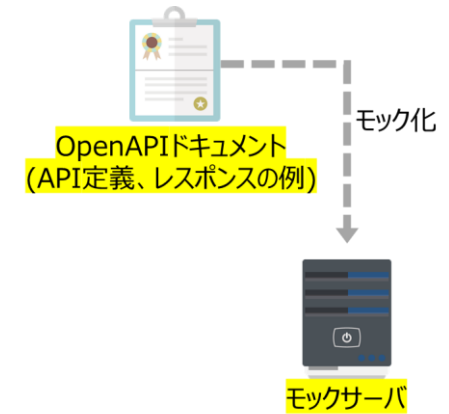
REST APIを呼び出せるようにします。

frontendディレクトリで次のコマンドで起動します。

```
$ docker-compose -f docker/docker-compose.api-mock.yml up
```

ブラウザで次のURLにアクセスします。

```
http://localhost:9080/api/todos
```



APIクライアントの作成

生成したクライアントコードをラッピングしたBackendServiceを作成します。

BackendServiceにより、各機能を作る時のREST APIの呼び出しを実装しやすくし、API呼び出し時の共通処理を埋め込むことが可能になります。

src/backendディレクトリにBackendService.tsファイルを作ります。

1からタイプすると時間がかかります。

ファイルを作ったら次のページのコードをコピーしましょう。

コピーしたら、Alt + Shift + Fを押してフォーマットしましょう。

作業が終わったら全員でリーディングしましょう。

BackendService.ts

```
import {
  Configuration,
  TodosApi,
  Middleware,
  UsersApi
} from './generated-rest-client';

const requestLogger: Middleware = {
  pre: async (context) => {
    console.log(`>> ${context.init.method} ${context.url}`, context.init);
  },
  post: async (context) => {
    console.log(`<< ${context.response.status} ${context.url}`, context.response);
  }
}

const configuration = new Configuration({
  middleware: [requestLogger]
});

const todosApi = new TodosApi(configuration);

const usersApi = new UsersApi(configuration);

const signup = async (userName: string, password: string) => {
  return usersApi.signup({ inlineObject2: { userName, password } });
};

const login = async (userName: string, password: string) => {
  return usersApi.login({ inlineObject3: { userName, password } });
};

const logout = async () => {
  return usersApi.logout();
};
```



```
const getTodos = async () => {
  return todosApi.getTodos();
};

const postTodo = async (text: string) => {
  return todosApi.postTodo({ inlineObject: { text } });
};

const putTodo = async (todoId: number, completed: boolean) => {
  return todosApi.putTodo({ todoId, inlineObject1: { completed } });
};

export const BackendService = {
  signup,
  login,
  logout,
  getTodos,
  postTodo,
  putTodo
};
```

Middlewareと呼ばれる部品を作成して、リクエストやレスポンスに対する共通的な処理を実装できます。開発時にREST APIの呼び出しを確認しやすいように、リクエストとレスポンスをコンソールにログ出力するMiddlewareを作成しています。

ToDo管理の開発

ToDo管理で開発するページ

ToDoの一覧表示、ToDoの登録を作ります。

ToDo状態の更新、ToDo一覧の絞り込みは
ハンズオン時間の都合で省略します。



いよいよ機能開発、その前に

画面イメージからどうやって作ってあげればいいのだろう？

まず何を作るのだろう？

何を考えないといけないのだろう？



The screenshot shows a web application titled "Todoアプリ" (Todo App). The header bar is black with white text. On the left, it says "Todoアプリ". On the right, it says "テストユーザーさん" and "ログアウト". Below the header, there is a form to add a new task. It consists of a text input field with the placeholder text "タスクを入力してください" and a green button labeled "追加". Below the input field, there are three filter buttons: "全て" (All), "未完了のみ" (Only incomplete), and "完了のみ" (Only completed). Below the filters, there is a list of tasks. Each task is represented by a row with a checkbox, the task text, and a delete button (an 'x' in a circle). The tasks are: "洗い物をする" (Do laundry) with a checked checkbox, "洗濯物を干す" (Hang laundry to dry) with an unchecked checkbox, and "買い物へ行く" (Go shopping) with an unchecked checkbox.

Todoアプリ

テストユーザーさん ログアウト

タスクを入力してください 追加

全て 未完了のみ 完了のみ

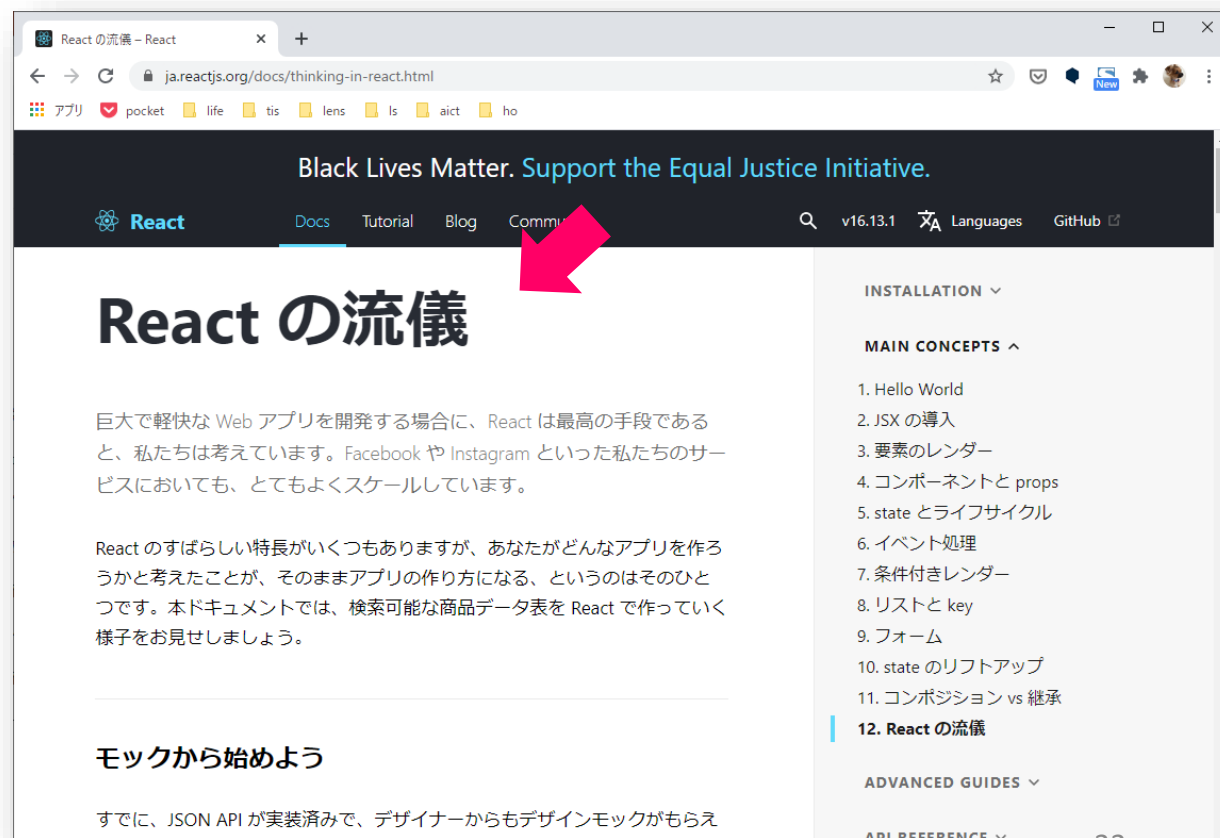
- ☒ 洗い物をする x
- ☐ 洗濯物を干す x
- ☐ 買い物へ行く x

先人の知恵に学ぼう

Reactの流儀

<https://ja.reactjs.org/docs/thinking-in-react.html>

ページの作り方が紹介されています。

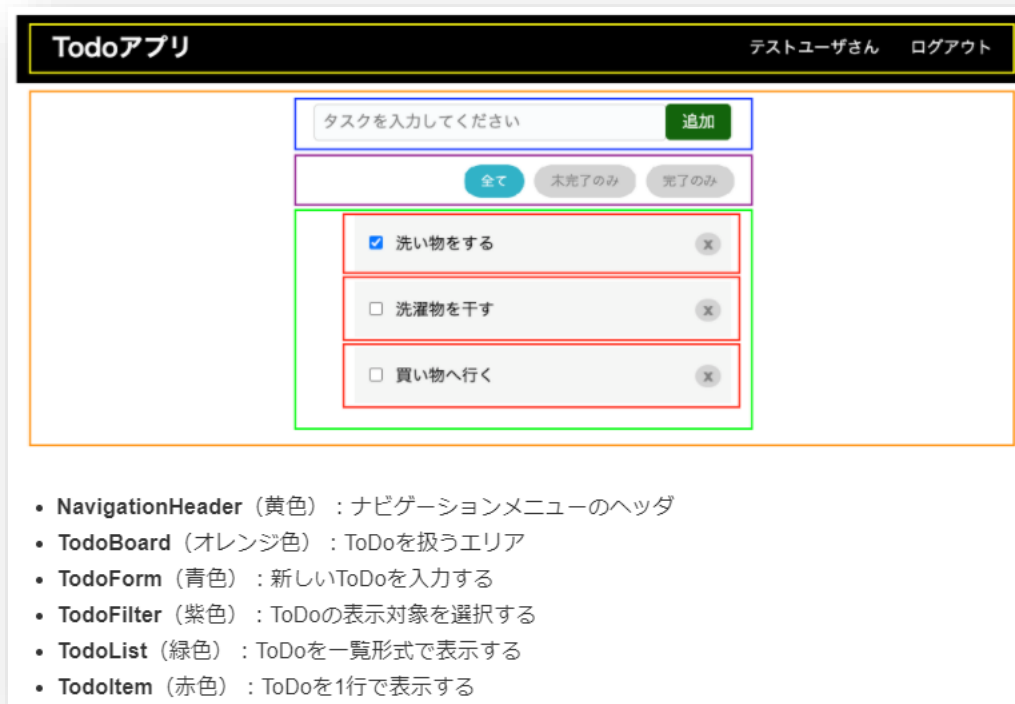


開発ステップ

モックをそのまま表示する



見たままにコンポーネントに分割する



状態(state)を決めて機能を作る

最小限の状態を考える

導出できるものは状態としない

状態をどのコンポーネントで持つか決める

ページ外観の作成

まずはモックをそのまま表示します。

はじめにアプリを起動し、
HTML、CSSの順に反映していきます。

Hello, world

はじめにアプリを起動します。
変更したら検知して反映されます。

HTMLを反映

Todoアプリ

- テストユーザーさん
- ログアウト

タスクを入力してください

追加

全て

未完了のみ

完了のみ

- ☒ 洗い物をする
- ☐ 洗濯物を干す
- ☐ 買い物へ行く

Todoアプリ

テストユーザーさん ログアウト

タスクを入力してください

追加

全て

未完了のみ

完了のみ

☒ 洗い物をする

X

☐ 洗濯物を干す

X

☐ 買い物へ行く

X

CSSを反映

アプリの起動

はじめにアプリを起動します。

frontendディレクトリで次のコマンドで起動します。

```
$ npm install ←初回のみ
```

```
$ npm run start
```

Hello, world

はじめにアプリを起動します。
変更したら検知して反映されます。

```
import React from 'react';

function App() {
  return (
    <h1>Hello, world</h1>
  );
}

export default App;
```


HTMLの反映

次にHTMLを反映します。

srcディレクトリのApp.tsxファイルを開き、
モックのHTMLを反映します。

モックの場所

spa-restapi-handson/todo-app-mock/

Hello, world

はじめにアプリを起動します。
変更したら検知して反映されます。

HTMLを反映

Todoアプリ

- テストユーザーさん
- ログアウト

タスクを入力してください

追加

全て

未完了のみ

完了のみ

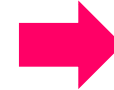
- ☒ 洗い物をする
☐
- ☐ 洗濯物を干す
☐
- ☐ 買い物へ行く
☐

HTMLの反映

モックのbodyタグの内容（header～div）を
Appのreturn文のカッコの中にコピーします。
Appに元々あったh1は消してください。

コピーするとHTMLのままではエラーが出るので
次のページ以降で対応します。

```
<!DOCTYPE html>
<html lang="en">
<head>
  . . .
</head>
<body>
  <header class="PageHeader_header">
    . . .
  </header>
  <div class="TodoBoard_content">
    <div class="TodoForm_content">
      . . .
    </div>
    <div class="TodoFilter_content">
      . . .
    </div>
    <ul class="TodoList_list">
      . . .
    </ul>
  </div>
</body>
</html>
```



```
import React from 'react';

function App() {
  return (
    <h1>Hello, world</h1>
  );
}

export default App;
```

ReactではJSXと呼ばれるJavaScriptの拡張構文を使ってUIを実装します。

JSXの親要素を1つにする

JSXでは親要素を1つにする必要があります。

headerとdivの2つの親要素があります。

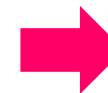
親要素が複数ある場合は

Reactが提供するFragmentコンポーネントで
全体を囲って親要素を1つにします。

```
import React from 'react';

function App() {
  return (
    <header class="PageHeader_header">
      . . .
    </header>
    <div class="ToDoBoard_content">
      . . .
    </div>
  );
}

export default App;
```



```
import React from 'react';

function App() {
  return (
    <React.Fragment>
      <header class="PageHeader_header">
        . . .
      </header>
      <div class="ToDoBoard_content">
        . . .
      </div>
    </React.Fragment>
  );
}

export default App;
```

class属性をclassName属性に修正する

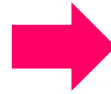
JSXではCSSのクラス指定をclassName属性に指定します。

class属性をclassName属性に一括置き換えします。

```
import React from 'react';

function App() {
  return (
    <React.Fragment>
      <header class="PageHeader_header">
        . . .
      </header>
      <div class="TodoBoard_content">
        . . .
      </div>
    </React.Fragment>
  );
}

export default App;
```



```
import React from 'react';

function App() {
  return (
    <React.Fragment>
      <header className="PageHeader_header">
        . . .
      </header>
      <div className="TodoBoard_content">
        . . .
      </div>
    </React.Fragment>
  );
}

export default App;
```

checked属性をJS式に修正する

JSXではchecked属性にbooleanの値を指定する必要があります。

JSXでは`{}`でJavaScriptの式を書きます。

checked属性に`{true}`を指定します。

```
<input type="checkbox" className="TodoItem_checkbox" checked="checked" />
```



```
<input type="checkbox" className="TodoItem_checkbox" checked={true} />
```

App.tsx

```
import React from 'react';

function App() {
  return (
    <React.Fragment>
      <header className="PageHeader_header">
        <h1 className="PageHeader_title">Todoアプリ</h1>
        <nav>
          <ul className="PageHeader_nav">
            <li>テストユーザさん</li>
            <li>ログアウト</li>
          </ul>
        </nav>
      </header>
      <div className="TodoBoard_content">
        <div className="TodoForm_content">
          <form className="TodoForm_form">
            <div className="TodoForm_input">
              <input type="text" placeholder="タスクを入力してください" />
            </div>
            <div className="TodoForm_button">
              <button type="button">追加</button>
            </div>
          </form>
        </div>
        <div className="TodoFilter_content">
          <button className="TodoFilter_buttonSelected">全て</button>
          <button className="TodoFilter_buttonUnselected">未完了のみ</button>
          <button className="TodoFilter_buttonUnselected">完了のみ</button>
        </div>
      </div>
    </React.Fragment>
  );
}
```

HTML→JSXの変更内容

- 親要素を1つにする
- class属性→className属性にする
- checked属性などboolean値は{JS式}にする

```
<ul className="TodoList_list">
  <li className="TodoItem_item">
    <div className="TodoItem_todo">
      <label>
        <input type="checkbox" className="TodoItem_checkbox" checked={true} />
        <span>洗い物をする</span>
      </label>
    </div>
    <div className="TodoItem_delete">
      <button className="TodoItem_button">x</button>
    </div>
  </li>
  <li className="TodoItem_item">
    <div className="TodoItem_todo">
      <label>
        <input type="checkbox" className="TodoItem_checkbox" />
        <span>洗濯物を干す</span>
      </label>
    </div>
    <div className="TodoItem_delete">
      <button className="TodoItem_button">x</button>
    </div>
  </li>
  <li className="TodoItem_item">
    <div className="TodoItem_todo">
      <label>
        <input type="checkbox" className="TodoItem_checkbox" />
        <span>買い物へ行く</span>
      </label>
    </div>
    <div className="TodoItem_delete">
      <button className="TodoItem_button">x</button>
    </div>
  </li>
</ul>
</div>
</React.Fragment>
);
}

export default App;
```

CSSの反映

次にCSSを反映します。

srcディレクトリにApp.cssファイルを作成し、
モックのCSSを全てコピーします。

モックの場所

spa-restapi-handson/todo-app-mock/

App.tsxでApp.cssをインポートすると
インポートしたCSSが適用されます。

Hello, world

はじめにアプリを起動します。
変更したら検知して反映されます。

HTMLを反映

Todoアプリ

- テストユーザーさん
- ログアウト

タスクを入力してください

追加

全て

未完了のみ

完了のみ

- ☒ 洗い物をする
- ☐ 洗濯物を干す
- ☐ 買い物へ行く

Todoアプリ

テストユーザーさん ログアウト

タスクを入力してください

追加

全て

未完了のみ

完了のみ

- ☒ 洗い物をする

- ☐ 洗濯物を干す

- ☐ 買い物へ行く

CSSを反映

App.css

```
body {  
  margin: 0;  
}  
  
.PageHeader_header {  
  display: flex;  
  justify-content: space-between;  
  align-items: center;  
  padding: 0 5%;  
  border-bottom: solid 1px black;  
  background: black;  
}  
  
  :  
  省略  
  :  
  
.TodoItem_button {  
  font-size: 17px;  
  font-weight: bold;  
  border: none;  
  color: grey;  
  background: lightgrey;  
  border-radius: 100%;  
  width: 25px;  
  height: 25px;  
  line-height: 20px;  
  cursor: pointer;  
  outline: none;  
}
```

App.tsx

```
import React from 'react';  
import './App.css';  
  
function App() {  
  . . .  
}  
  
export default App;
```

インポートするとCSSが適用されます。

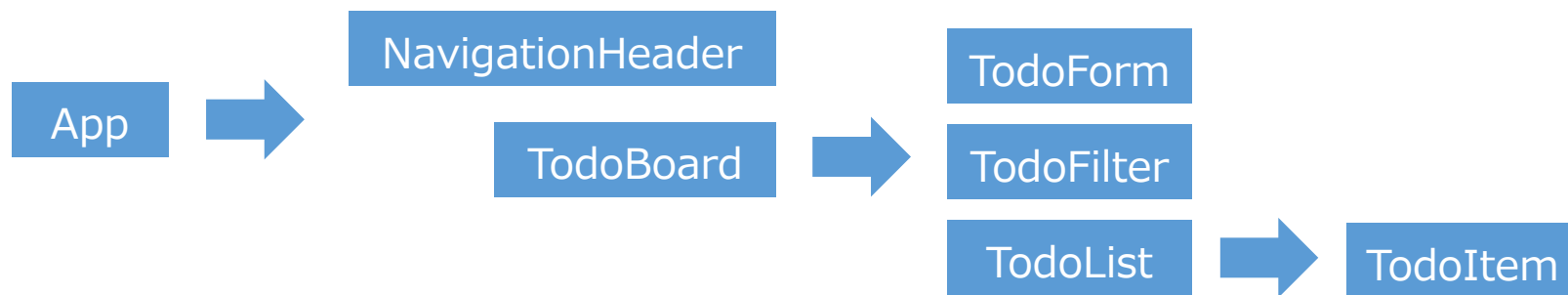
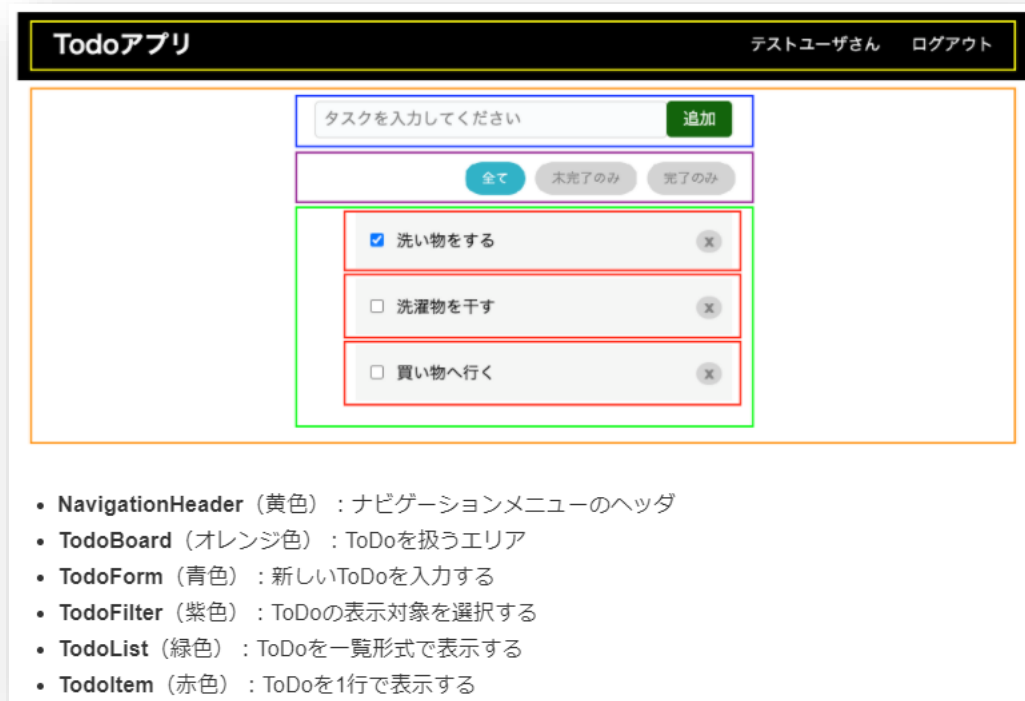


コンポーネントの分割

見たままにコンポーネントに分割します。

分割⇔表示を繰り返しながら徐々に分割します。

時間の都合上、分割作業はショートカットします。



コンポーネントの分割

1からタイプすると時間がかかります。

分割後のファイルがありますので

上書きでペーストしましょう。

spa-restapi-handson/support/ToDo管理のコンポーネント

srcディレクトリに上書きでコピーします。

TodoList.tsx

```
import React from 'react';
import { TodoItem } from './TodoItem';
import './TodoList.css';

export const TodoList: React.FC = () => {
  return (
    <ul className="TodoList_list">
      <TodoItem text="洗い物をする" completed={true} />
      <TodoItem text="洗濯物を干す" completed={false} />
      <TodoItem text="買い物へ行く" completed={false} />
    </ul>
  );
};
```

TodoItem.tsx

```
import React from 'react';
import './TodoItem.css';

type Props = {
  text: string;
  completed: boolean;
}

export const TodoItem: React.FC<Props> = ({ text, completed }) => {
  return (
    <li className="TodoItem_item">
      <div className="TodoItem_todo">
        <label>
          <input type="checkbox" className="TodoItem_checkbox" checked={completed} />
          <span>{text}</span>
        </label>
      </div>
      <div className="TodoItem_delete">
        <button className="TodoItem_button">x</button>
      </div>
    </li>
  );
};
```

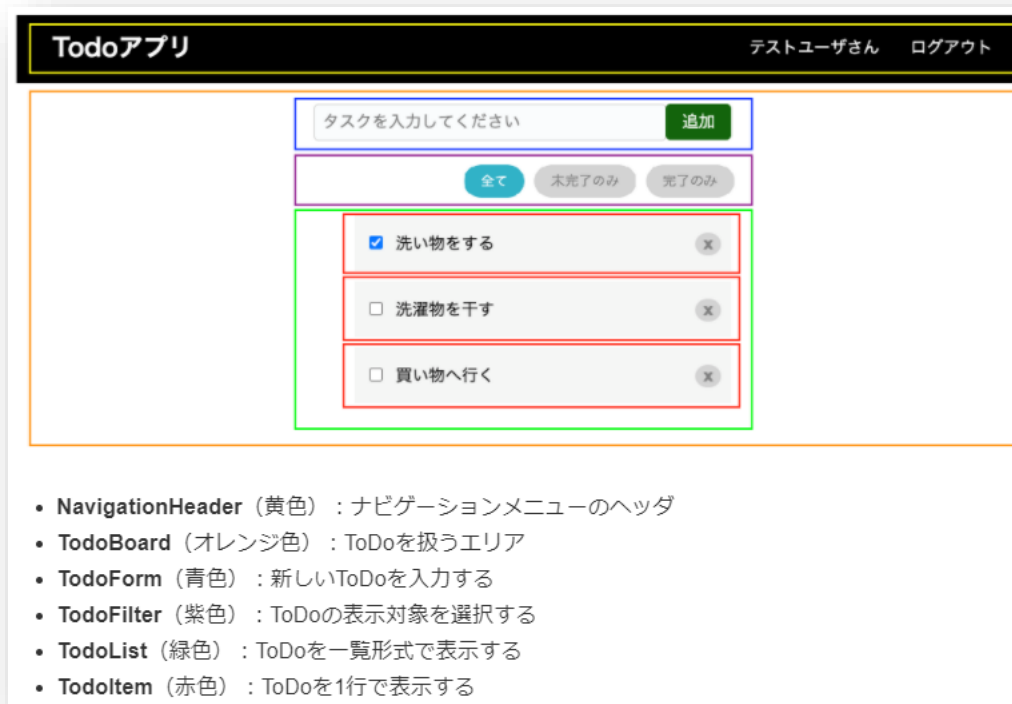
Reactでは、コンポーネントが外から値を受け取るために、プロパティと呼ばれる仕組みを提供しています。

開発ステップ°（再掲）

モックをそのまま表示する



見たままにコンポーネントに分割する



状態(state)を決めて機能を作る

最小限の状態を考える

導出できるものは状態としない

状態をどのコンポーネントで持つか決める

ToDoの一覧表示

ToDoを一覧表示できるように実装します。

ToDoの一覧表示に必要なstate(状態)は？

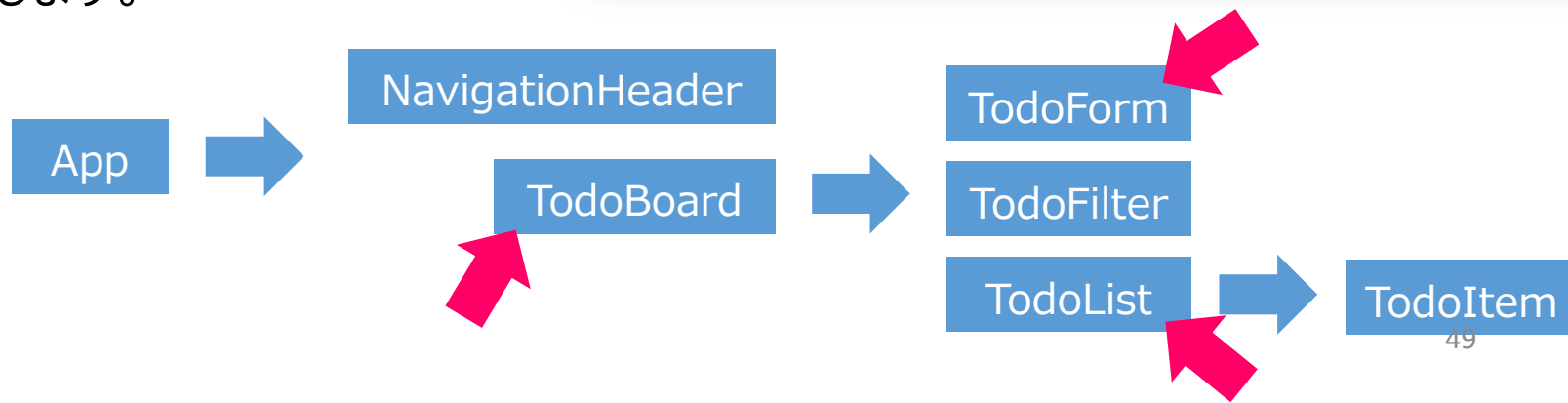
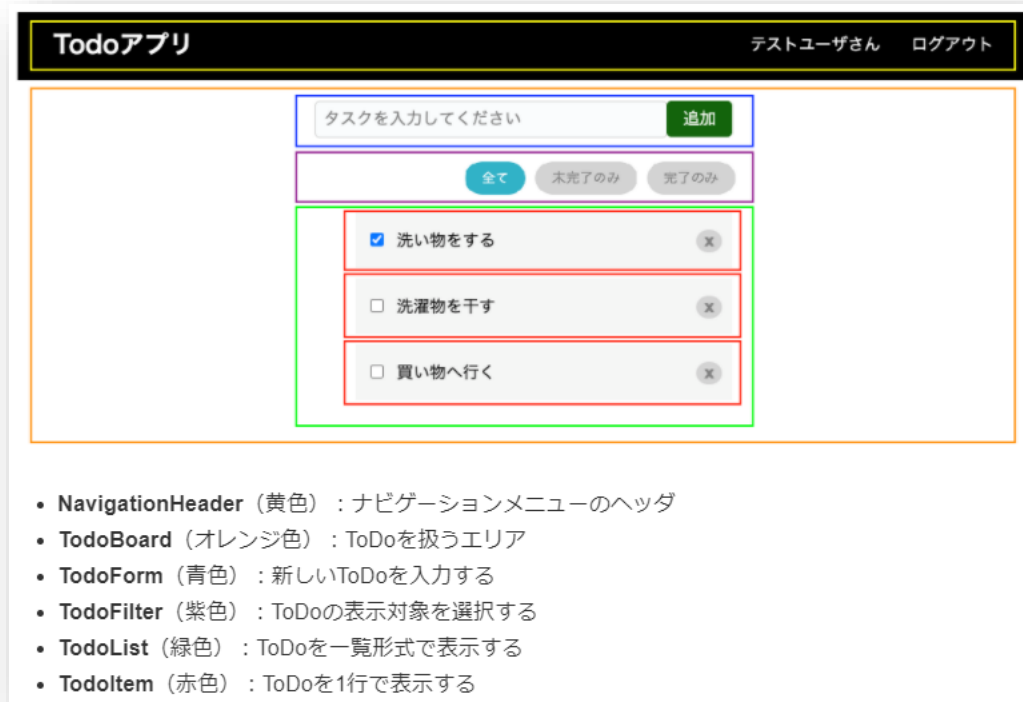
ToDoの一覧は変化していくものなのでstateとします。

このstateを配置するコンポは？

TodoListでは？ TodoFormも一覧に追加する？

複数のコンポで使うので共通の親コンポである

TodoBoardにこのstateを配置します。



ToDoの一覧表示（stateの追加）

TodoBoard.tsx

```
import React, { useState } from "react";
import './TodoBoard.css';
import { TodoFilter } from './TodoFilter';
import { TodoForm } from './TodoForm';
import { TodoList } from './TodoList';

type Todo = {
  text: string;
  completed: boolean;
}

export const TodoBoard: React.FC = () => {

  const [todos] = useState<Todo[]>([
    { text: "洗い物をする", completed: true },
    { text: "洗濯物を干す", completed: false },
    { text: "買い物へ行く", completed: false }
  ]);

  return (
    <div className="TodoBoard_content">
      <TodoForm />
      <TodoFilter />
      <TodoList todos={todos} />
    </div>
  );
};
```

TodoList.tsx

```
import React from 'react';
import { TodoItem } from './TodoItem';
import './TodoList.css';

type Todo = {
  text: string;
  completed: boolean;
}

type Props = {
  todos: Todo[];
}

export const TodoList: React.FC<Props> = ({ todos }) => {
  return (
    <ul className="TodoList_list">
      {todos.map(todo =>
        <TodoItem text={todo.text} completed={todo.completed} />
      )}
    </ul>
  );
};
```

Reactの機能はフックと呼ばれる関数で提供されます。
コンポーネントの状態を実現するstateフックを使用します。

ToDoの一覧表示（stateの更新）

TodoBoard.tsx

```
import React, { useState } from "react";
import './TodoBoard.css';
import { TodoFilter } from './TodoFilter';
import { TodoForm } from './TodoForm';
import { TodoList } from './TodoList';

type Todo = {
  ...
}

export const TodoBoard: React.FC = () => {

  const [todos] = useState<Todo[]>([
    { text: "洗い物をする", completed: true },
    { text: "洗濯物を干す", completed: false },
    { text: "買い物へ行く", completed: false }
  ]);

  return (
    ...
  );
};
```



```
import React, { useEffect, useState } from "react";
import './TodoBoard.css';
import { TodoFilter } from './TodoFilter';
import { TodoForm } from './TodoForm';
import { TodoList } from './TodoList';

type Todo = {
  ...
}

export const TodoBoard: React.FC = () => {

  const [todos, setTodos] = useState<Todo[]>([]);

  useEffect(() => {
    setTodos([
      { text: "洗い物をする", completed: true },
      { text: "洗濯物を干す", completed: false },
      { text: "買い物へ行く", completed: false }
    ]);
  }, []);

  return (
    ...
  );
};
```



```
import React, { useEffect, useState } from "react";
import { BackendService } from '../backend/BackendService';
import './TodoBoard.css';
import { TodoFilter } from './TodoFilter';
import { TodoForm } from './TodoForm';
import { TodoList } from './TodoList';

type Todo = {
  ...
}

export const TodoBoard: React.FC = () => {

  const [todos, setTodos] = useState<Todo[]>([]);

  useEffect(() => {
    BackendService.getTodos()
      .then(response => setTodos(response));
  }, []);

  return (
    ...
  );
};
```

コンポの状態に影響を与えることを副作用と言います。
副作用を実現するeffectフックを使用します。

REST APIを呼び出すように変更します。

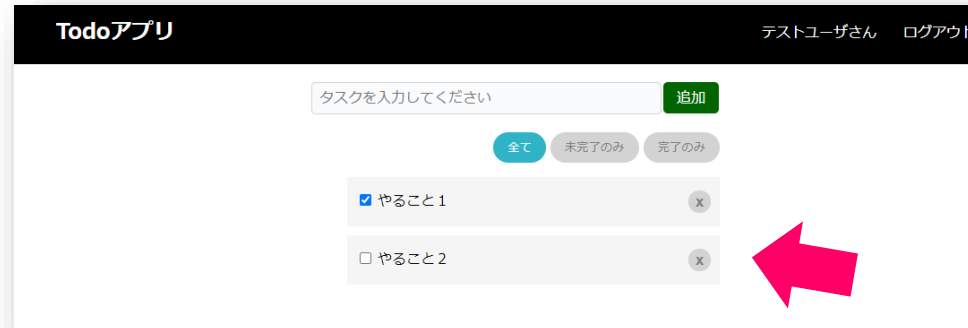
ToDoの一覧表示（動作確認）

モックサーバを起動します。

frontendディレクトリで次のコマンドで起動します。

```
$ docker-compose -f docker/docker-compose.api-mock.yml up
```

OpenAPIドキュメントのレスポンスがToDo一覧に表示されます。



ToDoの登録

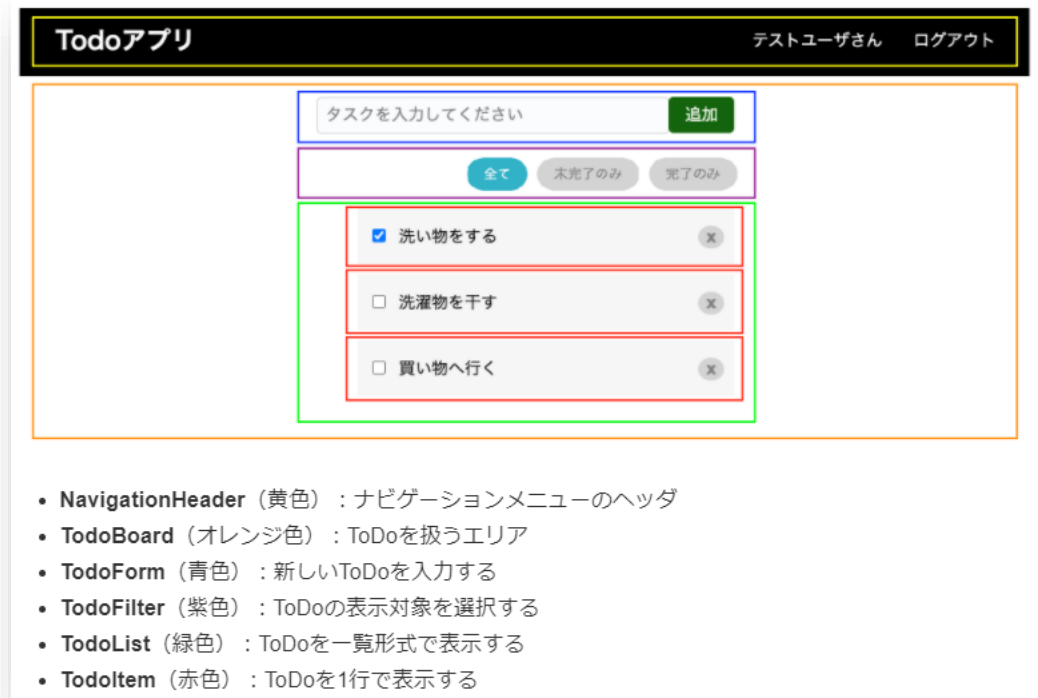
ToDoを登録できるように実装します。
「ToDoの登録」を見ながら作業します。

ToDoの登録に必要なstate(状態)は？

入力中の状態を保持するstateが必要になります。
ToDoの入力内容をstateとします。

このstateを配置するコンポーネントは？

TodoFormでしか使わないためTodoFormに配置します。



ToDoの登録（stateの追加）

TodoForm.tsx

```
import React from 'react';
import { useInput } from '../hooks';
import './TodoForm.css';

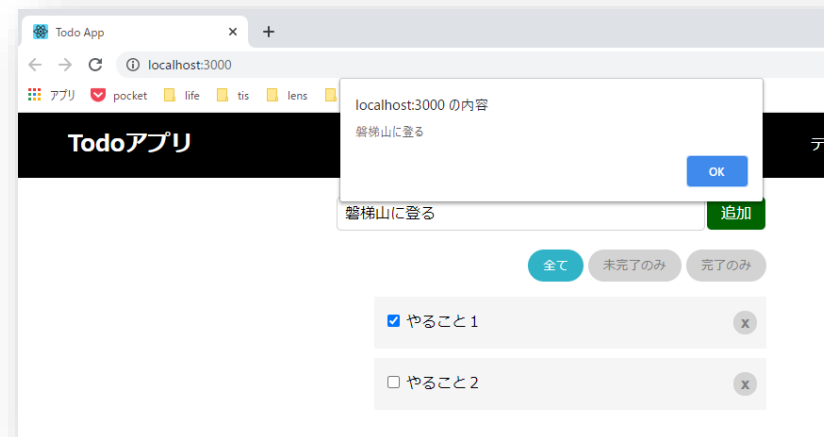
export const TodoForm: React.FC = () => {

  const [text, textAttributes, setText] = useInput('');

  const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    // ここに登録した際の処理を書く予定
    window.alert(text);
  }

  return (
    <div className="TodoForm_content">
      <form onSubmit={handleSubmit} className="TodoForm_form">
        <div className="TodoForm_input">
          <input type="text" {...textAttributes} placeholder="タスクを入力してください" />
        </div>
        <div className="TodoForm_button">
          <button type="submit">追加</button>
        </div>
      </form>
    </div>
  );
};
```

useInputという独自のフックを使ってstateを追加します。
window.alertを入れてstateの追加を確認します。



ToDoの登録（stateの更新）

TodoForm.tsx

```
import React from 'react';
import { BackendService } from '../backend/BackendService';
import { Todo } from '../backend/generated-rest-client';
import { useInput } from '../hooks';
import './TodoForm.css';

interface Props {
  addTodo: (returnedTodo: Todo) => void;
}

export const TodoForm: React.FC<Props> = ({ addTodo }) => {

  const [text, textAttributes, setText] = useInput('');

  const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    if (!text) {
      return;
    }
    BackendService.postTodo(text)
      .then(response => addTodo(response));
    setText('');
  }

  return (
    . . .
  );
};
```

TodoBoard.tsx

```
. . .
export const TodoBoard: React.FC = () => {

  const [todos, setTodos] = useState<Todo[]>([]);

  useEffect(() => {
    . . .
  }, []);

  const addTodo = (returnedTodo: Todo) => {
    setTodos(todos.concat(returnedTodo));
  }

  return (
    <div className="TodoBoard_content">
      <TodoForm addTodo={addTodo} />
      <TodoFilter />
      <TodoList todos={todos} />
    </div>
  );
};
```

BackendServiceを呼び出すように変更します。
TodoBoardが持っているToDo一覧に追加する必要があるため、
TodoBoardにstate更新用のコールバック関数を設け、
TodoFormにプロパティでその関数を渡します。

ToDoの登録（動作確認）

ToDoアプリ

テストコ

タスクを入力してください

追加

全て

未完了のみ

完了のみ

☒ やること1

x

☐ やること2

x

☐ やること3

x

☐ やること3

x

☐ やること3

x

何を入力してもexampleに定義した「やること3」が追加されます。

```
42 post:
43   summary: ToDoの登録
44   tags:
45     - todos
46   description: >
47     ToDoを登録する。
48   operationId: postTodo
49   requestBody:
50     required: true
51     content:
52       application/json:
53         schema:
54           type: object
55           properties:
56             text:
57               type: string
58               description: ToDoのタイトル
59           required:
60             - text
61           additionalProperties: false
62   examples:
63     example:
64       value:
65         text: やること3
66   responses:
67     '200':
68       description: OK
69       content:
70         application/json:
71           schema:
72             $ref: '#/components/schemas/ToDo'
73   examples:
74     example:
75       value:
76         id: 2003
77         text: やること3
78         completed: false
```

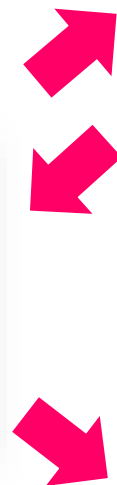
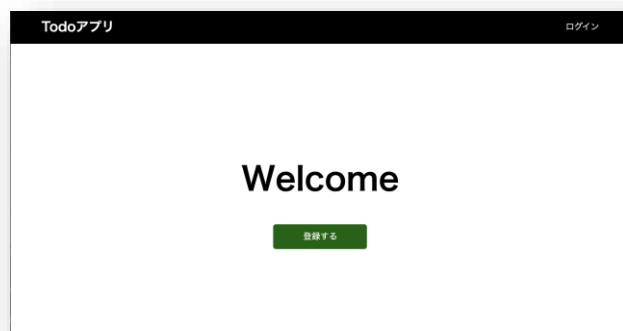
ユーザ認証の開発

ユーザ認証で開発するページ

ページを切り替えるルーティング

ログイン状態に応じたナビゲーションの切り替え

入力値のバリデーション



ページ外観の作成

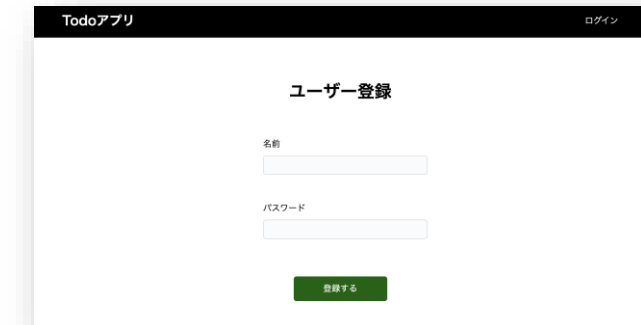
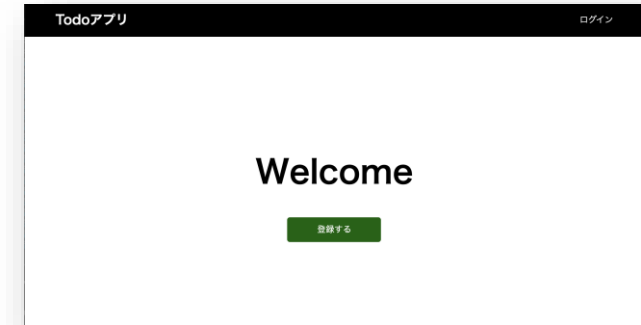
ユーザー認証で追加するページの外観を作成します。

1からタイプすると時間がかかります。

分割後のファイルがありますのでペーストしましょう。

spa-restapi-handson/support/ユーザー認証のコンポーネント

ここはページ追加するだけで、
目新しいものがないので次に進みます。



URLルーティングの設定

それぞれのページに遷移できるようにします。

React用のルーティングライブラリであるReact Routerを導入します。

frontendディレクトリで次のコマンドでインストールします。

```
$ npm install --save react-router-dom @types/react-router-dom
```


URLルーティングの設定（ルーティングの定義）

App.tsx

```
import React from 'react';
import './App.css';
import { NavigationHeader } from './components/NavigationHeader';
import { TodoBoard } from './components/TodoBoard';

function App() {
  return (
    <React.Fragment>
      <NavigationHeader />
      <TodoBoard />
    </React.Fragment>
  );
}
export default App;
```



```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom';
import './App.css';
import { Login } from './components/Login';
import { NavigationHeader } from './components/NavigationHeader';
import { Signup } from './components/Signup';
import { TodoBoard } from './components/TodoBoard';
import { Welcome } from './components/Welcome';

function App() {
  return (
    <BrowserRouter>
      <NavigationHeader />
      <Switch>
        <Route exact path="/board">
          <TodoBoard />
        </Route>
        <Route exact path="/signup">
          <Signup />
        </Route>
        <Route exact path="/login">
          <Login />
        </Route>
        <Route exact path="/">
          <Welcome />
        </Route>
      </Switch>
    </BrowserRouter>
  );
}
export default App;
```

BrowserRouterコンポーネントでReactRouterを有効化します。
SwitchコンポーネントとRouteコンポーネントでルーティングを定義します。
デフォルトで部分一致のため、
exactプロパティで完全一致に変更しています。

URLルーティングの設定（Welcomeページ）

Welcome.tsx

```
import React from "react";
import { Link } from "react-router-dom";
import './Welcome.css';

export const Welcome: React.FC = () => {
  return (
    <div className="Welcome_content">
      <div>
        <h1 className="Welcome_title">Welcome</h1>
        <div className="Welcome_buttonGroup">
          <Link to="/signup">
            <button className="Welcome_button">登録する</button>
          </Link>
        </div>
      </div>
    </div>
  );
};
```

マークアップのボタンやリンク等で遷移させたい場合は
Linkコンポーネントを使用してページ遷移を実装します。

URLルーティングの設定（ユーザ登録ページ、ログインページ）

Signup.tsx

```
import React from "react";
import { useHistory } from "react-router-dom";
import './Signup.css';

export const Signup: React.FC = () => {

  const history = useHistory();

  const signup: React.FormEventHandler<HTMLFormElement> = async (event) => {
    event.preventDefault();
    history.push('/');
  }

  return (
    <div className="Signup_content">
      <div className="Signup_box">
        <div className="Signup_title">
          <h1>ユーザー登録</h1>
        </div>
        <form className="Signup_form" onSubmit={signup}>
          <div className="Signup_item">
            <div className="Signup_label">名前</div>
            <input type="text" />
          </div>
          <div className="Signup_item">
            <div className="Signup_label">パスワード</div>
            <input type="password" />
          </div>
          <div className="Signup_buttonGroup">
            <button className="Signup_button">登録する</button>
          </div>
        </form>
      </div>
    </div>
  );
};
```

イベントハンドリング等、プログラムで遷移させたい場合は
historyフックを使用してページ遷移を実装します。

Login.tsx

```
import React from "react";
import { useHistory } from "react-router-dom";
import './Login.css';

export const Login: React.FC = () => {

  const history = useHistory();

  const login: React.FormEventHandler<HTMLFormElement> = async (event) => {
    event.preventDefault();
    history.push('/board');
  }

  return (
    <div className="Login_content">
      <div className="Login_box">
        <div className="Login_title">
          <h1>ログイン</h1>
        </div>
        <form className="Login_form" onSubmit={login}>
          <div className="Login_item">
            <div className="Login_label">名前</div>
            <input type="text" />
          </div>
          <div className="Login_item">
            <div className="Login_label">パスワード</div>
            <input type="password" />
          </div>
          <div className="Login_buttonGroup">
            <button className="Login_button">ログインする</button>
          </div>
        </form>
      </div>
    </div>
  );
};
```

URLルーティングの設定（ナビゲーションヘッダ）

NavigationHeader.tsx

```
import React from 'react';
import { Link } from 'react-router-dom';
import './NavigationHeader.css';

export const NavigationHeader: React.FC = () => {

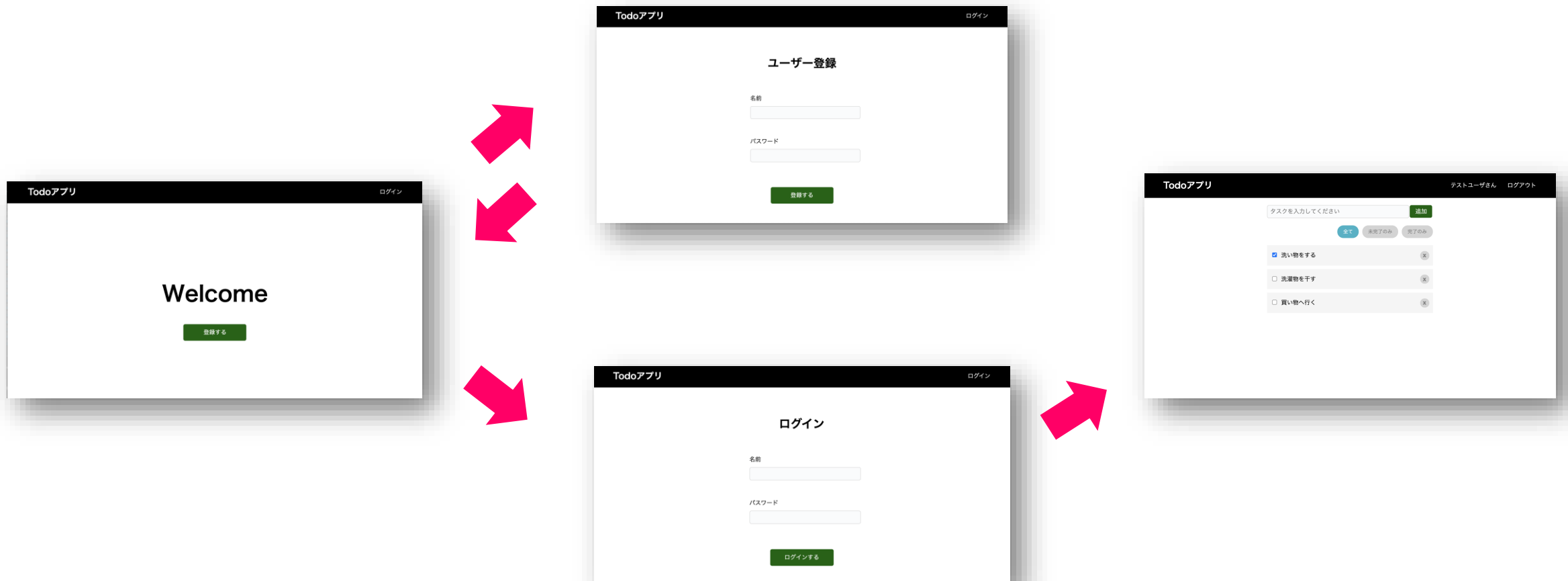
  const logout = async () => {
    window.location.href = '/';
  };

  return (
    <header className="PageHeader_header">
      <h1 className="PageHeader_title">Todoアプリ</h1>
      <nav>
        <ul className="PageHeader_nav">
          <li>
            <Link to="/login">ログイン</Link>
          </li>
          <li>テストユーザーさん</li>
          <li>
            <button type="button" onClick={logout}>ログアウト</button>
          </li>
        </ul>
      </nav>
    </header>
  );
};
```

ログアウト時はページを読み込み直してReactの状態を安全に破棄したいので、ReactRouterではなく
windows.location.hrefを使用します。

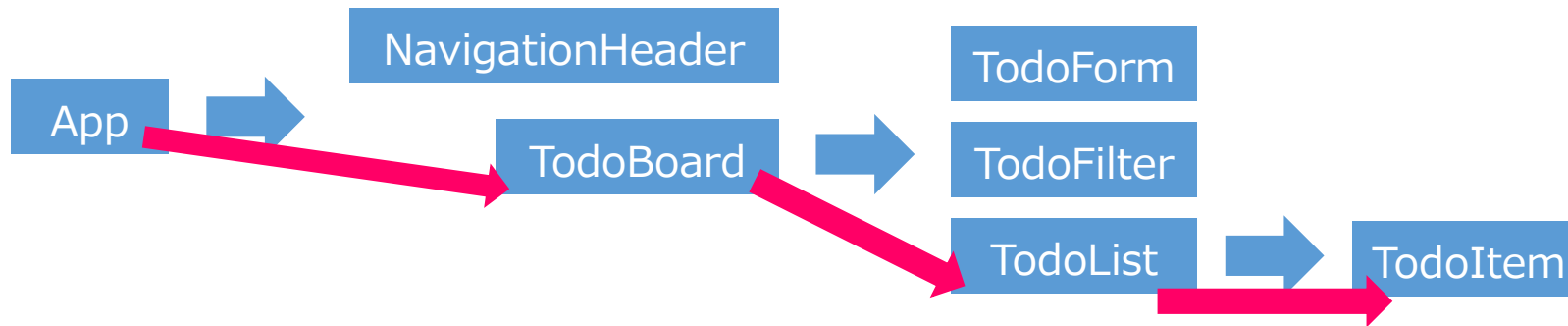
URLルーティングの設定（動作確認）

全てのページに遷移してみましょう。



コンテキスト

ユーザ情報のようにいろんなコンポから参照されるデータがある場合は？
プロパティを使うと渡していくのが大変です。



Reactの**コンテキスト**と呼ばれる仕組みを使うと、
プロパティを使わずに**コンポ間でデータを共有**できます。

ユーザコンテキストの作成

ユーザの認証に関する値をどのコンポからでも使用できるようにユーザコンテキストを作成します。
また、ユーザの認証に関する処理を集約したいため、合わせてユーザコンテキストに実装します。

ユーザコンテキストの内容

- ユーザ名
- ログインしているか？
- ユーザ登録
- ログイン
- ログアウト

UserContext.tsx

1からタイプすると時間がかかります。

実装済みのファイルがありますのでペーストしましょう。

spa-restapi-handson/support/ユーザコンテキスト

srcディレクトリにコピーします。

UserContext.tsx

```
import React, { useContext, useState } from 'react';
import { BackendService } from '../backend/BackendService';

export class AccountConflictError { }

export class AuthenticationFailedError { }

interface ContextValueType {
  signup: (userName: string, password: string) => Promise<void | AccountConflictError>,
  login: (userName: string, password: string) => Promise<void | AuthenticationFailedError>,
  logout: () => Promise<void>,
  userName: string
  isLoggedIn: boolean,
}

export const UserContext = React.createContext<ContextValueType>({} as ContextValueType);

export const useUserContext = () => useContext(UserContext);

export const UserContextProvider: React.FC = ({ children }) => {

  const [userName, setUserName] = useState<string>('');

  const contextValue: ContextValueType = {
    signup: async (userName, password) => { . . . },
    login: async (userName, password) => { . . . },
    logout: async () => { . . . },
    userName: userName,
    isLoggedIn: userName !== ''
  };

  return (
    <UserContext.Provider value={contextValue}>
      {children}
    </UserContext.Provider>
  );
};
```

ユーザコンテキストのインタフェースを定義します。
ユーザ登録時の重複エラー、ログイン時の認証失敗エラー
の際に返すクラスも定義します。

ユーザコンテキストを作成します。

各コンポーネントでユーザコンテキストを使うためのフックを作成します。

各コンポーネントでユーザコンテキストを使えるようにするためにプロバイダを作成します。

UserContext.tsx

```
import React, { useContext, useState } from 'react';
import { BackendService } from '../backend/BackendService';

export class AccountConflictError { }

export class AuthenticationFailedError { }

interface ContextValueType {
  signup: (userName: string, password: string) => Promise<void | AccountConflictError>,
  login: (userName: string, password: string) => Promise<void | AuthenticationFailedError>,
  logout: () => Promise<void>,
  userName: string
  isLoggedIn: boolean,
}

export const UserContext = React.createContext<ContextValueType>({} as ContextValueType);

export const useUserContext = () => useContext(UserContext);

export const UserContextProvider: React.FC = ({ children }) => {

  const [userName, setUserName] = useState<string>('');

  const contextValue: ContextValueType = {
    signup: async (userName, password) => { . . . },
    login: async (userName, password) => { . . . },
    logout: async () => { . . . },
    userName: userName,
    isLoggedIn: userName !== ''
  };

  return (
    <UserContext.Provider value={contextValue}>
      {children}
    </UserContext.Provider>
  );
};
```

ユーザコンテキストのインタフェースを定義します。
ユーザ登録時の重複エラー、ログイン時の認証失敗エラー
の際に返すクラスも定義します。

UserContext.tsx

```
import React, { useContext, useState } from 'react';
import { BackendService } from '../backend/BackendService';

export class AccountConflictError { }

export class AuthenticationFailedError { }

interface ContextValueType {
  signup: (userName: string, password: string) => Promise<void | AccountConflictError>,
  login: (userName: string, password: string) => Promise<void | AuthenticationFailedError>,
  logout: () => Promise<void>,
  userName: string
  isLoggedIn: boolean,
}

export const UserContext = React.createContext<ContextValueType>({} as ContextValueType);

export const useUserContext = () => useContext(UserContext);

export const UserContextProvider: React.FC = ({ children }) => {

  const [userName, setUserName] = useState<string>('');

  const contextValue: ContextValueType = {
    signup: async (userName, password) => { . . . },
    login: async (userName, password) => { . . . },
    logout: async () => { . . . },
    userName: userName,
    isLoggedIn: userName !== ''
  };

  return (
    <UserContext.Provider value={contextValue}>
      {children}
    </UserContext.Provider>
  );
};
```

ユーザコンテキストを作成します。

各コンポーネントでユーザコンテキストを使うためのフックを作成します。

UserContext.tsx

```
import React, { useContext, useState } from 'react';
import { BackendService } from '../backend/BackendService';

export class AccountConflictError { }

export class AuthenticationFailedError { }

interface ContextValueType {
  signup: (userName: string, password: string) => Promise<void | AccountConflictError>,
  login: (userName: string, password: string) => Promise<void | AuthenticationFailedError>,
  logout: () => Promise<void>,
  userName: string
  isLoggedIn: boolean,
}

export const UserContext = React.createContext<ContextValueType>({} as ContextValueType);

export const useUserContext = () => useContext(UserContext);

export const UserContextProvider: React.FC = ({ children }) => {

  const [userName, setUserName] = useState<string>('');

  const contextValue: ContextValueType = {
    signup: async (userName, password) => { ... },
    login: async (userName, password) => { ... },
    logout: async () => { ... },
    userName: userName,
    isLoggedIn: userName !== ''
  };

  return (
    <UserContext.Provider value={contextValue}>
      {children}
    </UserContext.Provider>
  );
};
```

ユーザコンテキストを使えるようにするためにプロバイダを作成します。

stateフックを使ってユーザ名を保持し、ユーザ認証に関わる処理を実装します。

UserContext.tsx

```
...
export const UserContextProvider: React.FC = ({ children }) => {
  const [userName, setUserName] = useState<string>('');

  const contextValue: ContextValueType = {
    signup: async (userName, password) => {
      try {
        await BackendService.signup(userName, password);
      } catch (error) {
        if (error.status === 409) {
          return new AccountConflictError();
        }
        throw error;
      }
    },
    login: async (userName, password) => {
      try {
        await BackendService.login(userName, password);
        setUserName(userName)
      } catch (error) {
        if (error.status === 401) {
          return new AuthenticationFailedError();
        }
        throw error;
      }
    },
    logout: async () => {
      await BackendService.logout();
      setUserName('');
    },
    userName: userName,
    isLoggedIn: userName !== ''
  };

  return (
    ...
  );
};
```

各処理ではBackendServiceを使って実装します。
エラー発生時は作成しておいたエラークラスを返します。

App.tsx

```
import React from 'react';
import { BrowserRouter, Route, Switch } from 'react-router-dom';
import './App.css';
import { Login } from './components/Login';
import { NavigationHeader } from './components/NavigationHeader';
import { Signup } from './components/Signup';
import { TodoBoard } from './components/TodoBoard';
import { Welcome } from './components/Welcome';
import { UserContextProvider } from './contexts/UserContext';

function App() {
  return (
    <UserContextProvider>
      <BrowserRouter>
        <NavigationHeader />
        <Switch>
          <Route exact path="/board">
            <TodoBoard />
          </Route>
          <Route exact path="/signup">
            <Signup />
          </Route>
          <Route exact path="/login">
            <Login />
          </Route>
          <Route exact path="/">
            <Welcome />
          </Route>
        </Switch>
      </BrowserRouter>
    </UserContextProvider>
  );
}
export default App;
```

作成したユーザコンテキストをいえるようにApp.tsxをい更します。
UserContextProviderで全体を囲みます。

NavigationHeader.tsx

```
import React from 'react';
import { Link } from 'react-router-dom';
import { useUserContext } from '../contexts/UserContext';
import './NavigationHeader.css';

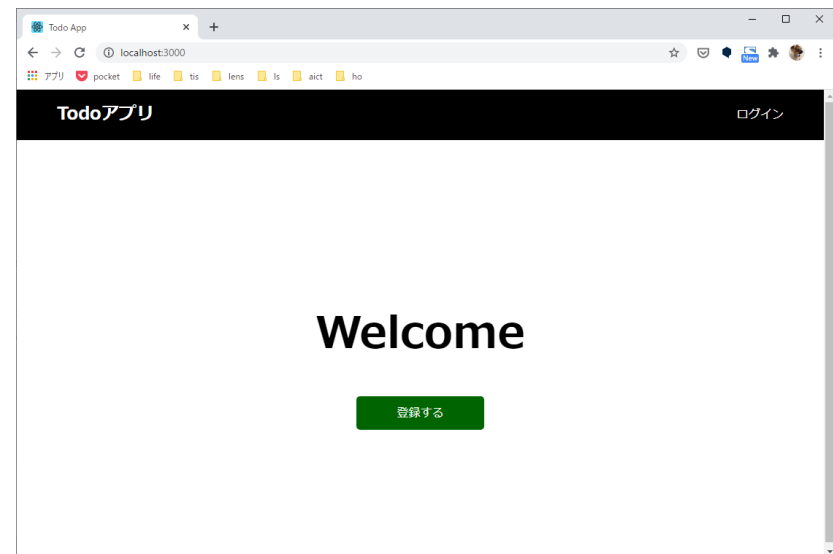
export const NavigationHeader: React.FC = () => {

  const useContext = useUserContext();

  const logout = async () => {
    window.location.href = '/';
  };

  return (
    <header className="PageHeader_header">
      <h1 className="PageHeader_title">Todoアプリ</h1>
      <nav>
        <ul className="PageHeader_nav">
          {userContext.isLoggedIn ? (
            <React.Fragment>
              <li>{userContext.userName}</li>
              <li>
                <button type="button" onClick={logout}>ログアウト</button>
              </li>
            </React.Fragment>
          ) : (
            <li>
              <Link to="/login">ログイン</Link>
            </li>
          )}
        </ul>
      </nav>
    </header>
  );
};
```

ログイン状態に応じたナビゲーションヘッダの表示切替を実装します。
ユーザコンテキストフックを使って実装します。
ナビゲーションヘッダがログインだけになります。



ログイン、サインアップ、ログアウト

ログイン、サインアップ、ログアウトまで一気にできるようにします。

1からタイプすると時間がかかります。

コピペを活用しましょう。

これで作業は最後ですので、

時間の許す限り実装や質問してください。



ログイン

TODO : 作成中

Login.tsx

```
import React, { useState } from "react";
import { useHistory } from 'react-router-dom';
import './Login.css';
import { useInput } from '../hooks/useInput';
import { AuthenticationFailedError, useUserContext } from '../contexts/UserContext';

export const Login: React.FC = () => {
  const [userName, userNameAttributes] = useInput('');
  const [password, passwordAttributes] = useInput('');
  const [formError, setFormError] = useState('');

  const history = useHistory();
  const userContext = useUserContext();

  const login: React.FormEventHandler<HTMLFormElement> = async (event) => {
    event.preventDefault();
    const result = await userContext.login(userName, password);
    if (result instanceof AuthenticationFailedError) {
      setFormError('ログインに失敗しました。名前またはパスワードが正しくありません。');
      return;
    }
    history.push('/board');
  };
};
```

```
return (
  <div className="Login_content">
    <div className="Login_box">
      <div className="Login_title">
        <h1>ログイン</h1>
        <div className="error">{formError}</div>
      </div>
      <form className="Login_form" onSubmit={login}>
        <div className="Login_item">
          <div className="Login_label">名前</div>
          <input type="text" {...userNameAttributes} />
        </div>
        <div className="Login_item">
          <div className="Login_label">パスワード</div>
          <input type="password" {...passwordAttributes} />
        </div>
        <div className="Login_buttonGroup">
          <button className="Login_button">ログインする</button>
        </div>
      </form>
    </div>
  </div>
);
```

ユーザコンテキストを使ってログイン処理を実装します。
認証失敗時のエラーメッセージを状態として保持し、認証失敗時のメッセージ表示に使用します。

ログイン（動作確認）

TODO：作成中

ログイン（バリデーション）

TODO：作成中

Login.tsx

```
import React, { useState } from "react";
import { useHistory } from 'react-router-dom';
import './Login.css';
import { useInput } from '../hooks/useInput';
import { AuthenticationFailedError, useUserContext } from '../contexts/UserContext';
import { stringField, useValidation } from '../validation';

type ValidationFields = {
  userName: string
  password: string
};

export const Login: React.FC = () => {
  ...
  const userContext = useUserContext();

  const { error, handleSubmit } = useValidation<ValidationFields>({
    userName: stringField()
      .required('名前を入力してください'),
    password: stringField()
      .required('パスワードを入力してください')
  });

  const login: React.FormEventHandler<HTMLFormElement> = async (event) => {
    ...
  };
};
```

```
return (
  <div className="Login_content">
    <div className="Login_box">
      <div className="Login_title">
        <h1>ログイン</h1>
        <div className="error">{formError}</div>
      </div>
      <form className="Login_form"
        onSubmit={handleSubmit({ userName, password }, login, () => setFormError(''))}>
        <div className="Login_item">
          <div className="Login_label">名前</div>
          <input type="text" {...userNameAttributes} />
          <div className="error">{error.userName}</div>
        </div>
        <div className="Login_item">
          <div className="Login_label">パスワード</div>
          <input type="password" {...passwordAttributes} />
          <div className="error">{error.password}</div>
        </div>
        <div className="Login_buttonGroup">
          <button className="Login_button">ログインする</button>
        </div>
      </form>
    </div>
  </div>
);
```

独自に作成したバリデーションフックを使って実装します。

クロージング

SPAハンズオンはいかがでしたでしょうか？

TODO : Fintanの紹介

SPAの作り方を体験いただけましたでしょうか？

冒頭で話した通り、Fintanでハンズオン資料として公開予定です。
公開後はどなたでもご使用いただけます。
同僚や友人に紹介いただいたり、社内勉強会でご活用ください。

<https://fintan.jp/>



サービス開発エンジニア体験

サービス/プロダクトの開発に欠かせない
アプリ開発とDevOpsを体験してみませんか？

9月 SPAハンズオン

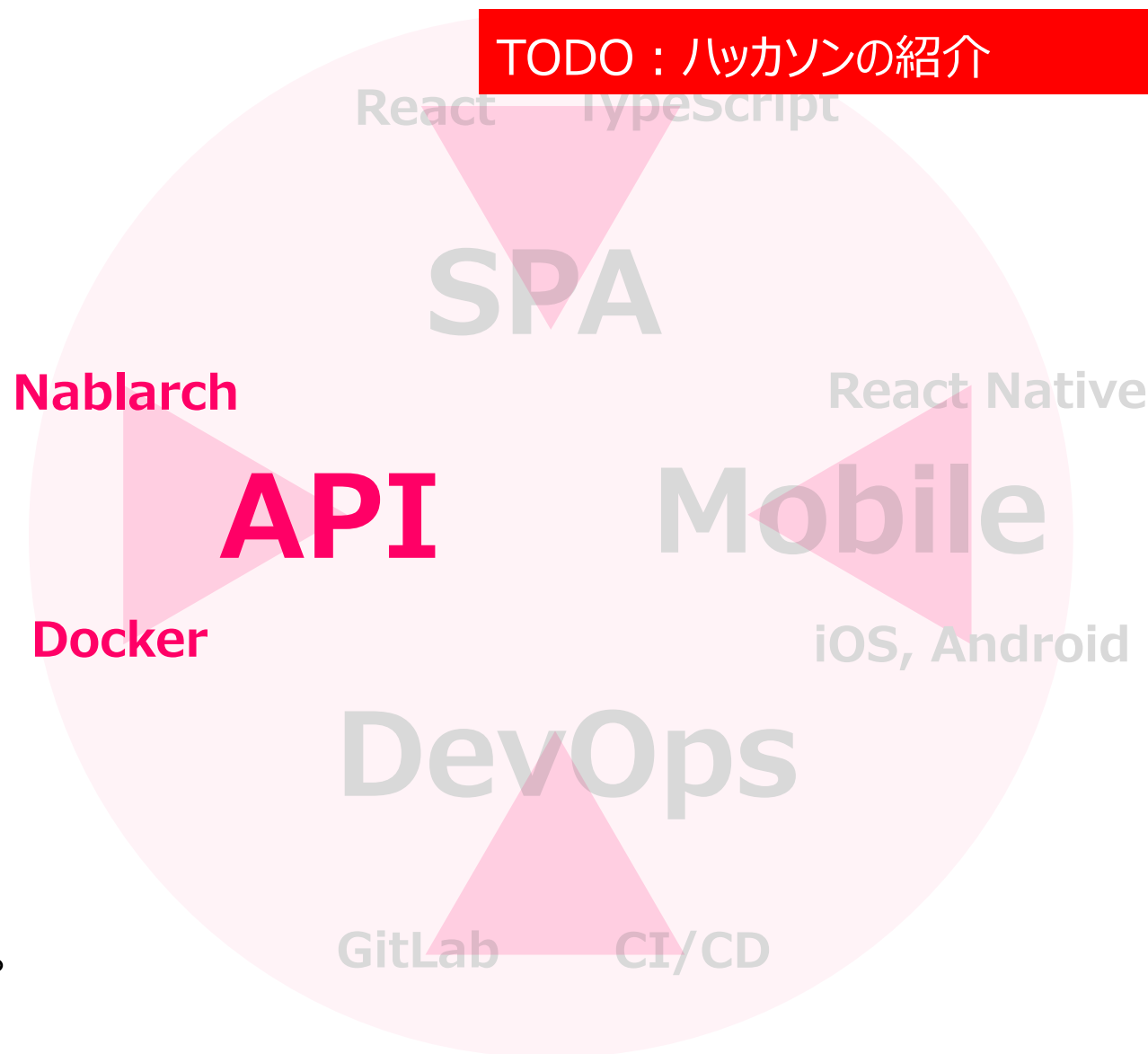
10月 APIハンズオン

11月 モバイルハンズオン

12月 DevOpsハンズオン

次回は10月にSPAのバックエンドとなる
APIの作り方を学ぶハンズオンを開催予定です。
ぜひご参加ください。

TODO : ハッカソンの紹介



サービス開発エンジニア体験

TODO : Aizurageの紹介

サービス/プロダクトの開発に欠かせない
アプリ開発とDevOpsを体験してみませんか？

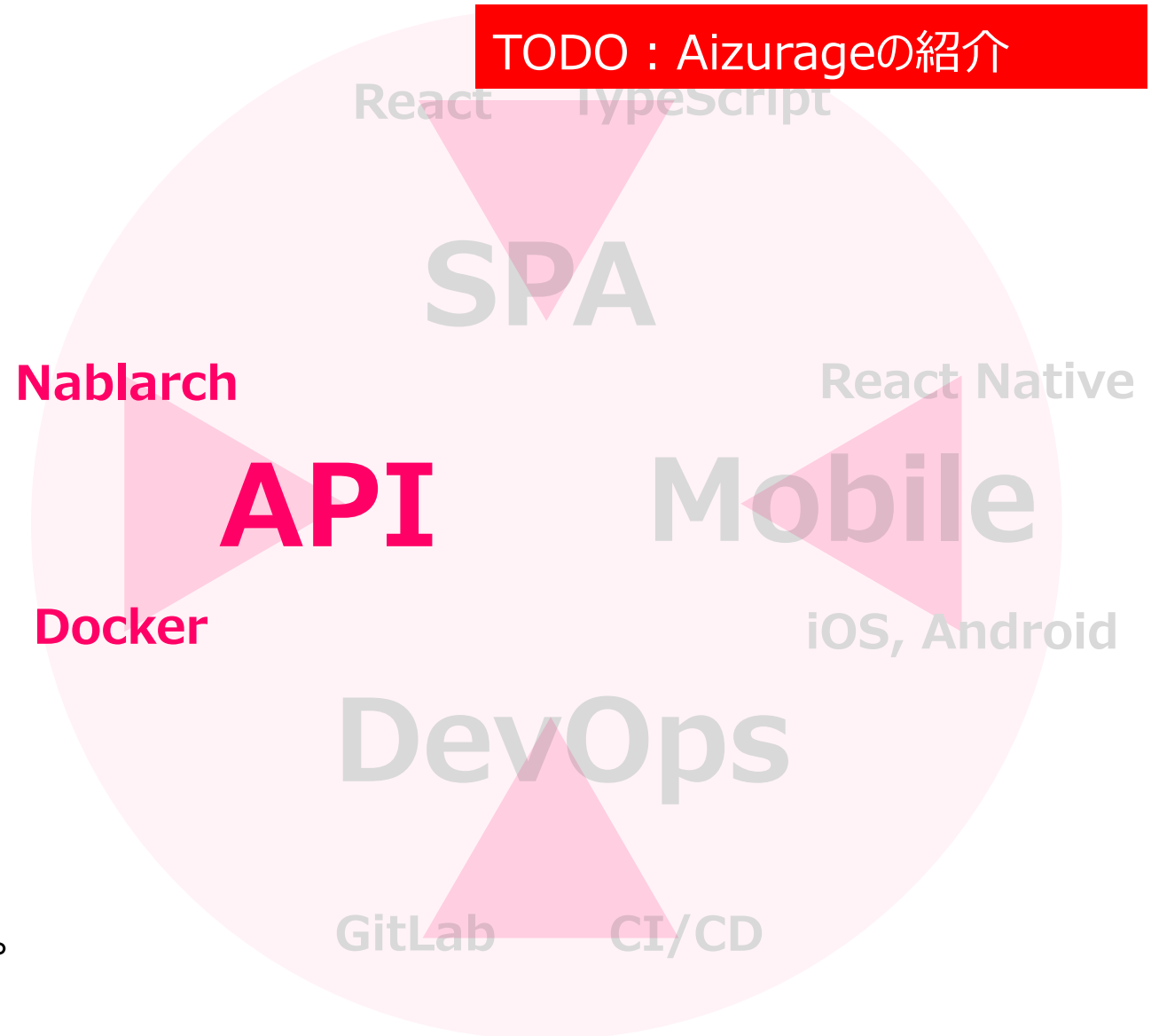
9月 SPAハンズオン

10月 APIハンズオン

11月 モバイルハンズオン

12月 DevOpsハンズオン

次回は10月にSPAのバックエンドとなる
APIの作り方を学ぶハンズオンを開催予定です。
ぜひご参加ください。



サービス開発エンジニア体験

TODO : AiCTの紹介

サービス/プロダクトの開発に欠かせない
アプリ開発とDevOpsを体験してみませんか？

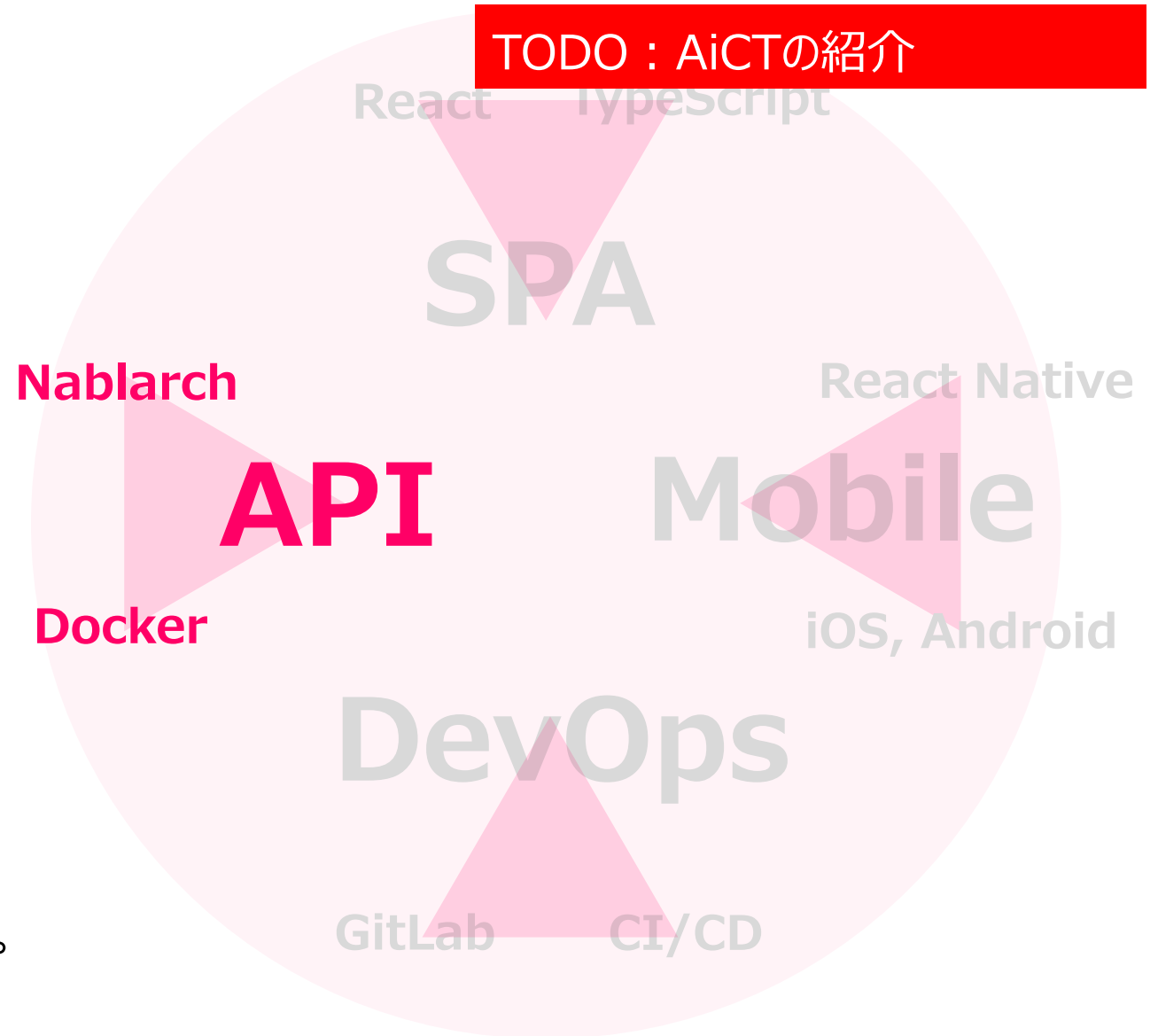
9月 SPAハンズオン

10月 APIハンズオン

11月 モバイルハンズオン

12月 DevOpsハンズオン

次回は10月にSPAのバックエンドとなる
APIの作り方を学ぶハンズオンを開催予定です。
ぜひご参加ください。



サービス開発エンジニア体験

TODO : 会津大との紹介

サービス/プロダクトの開発に欠かせない
アプリ開発とDevOpsを体験してみませんか？

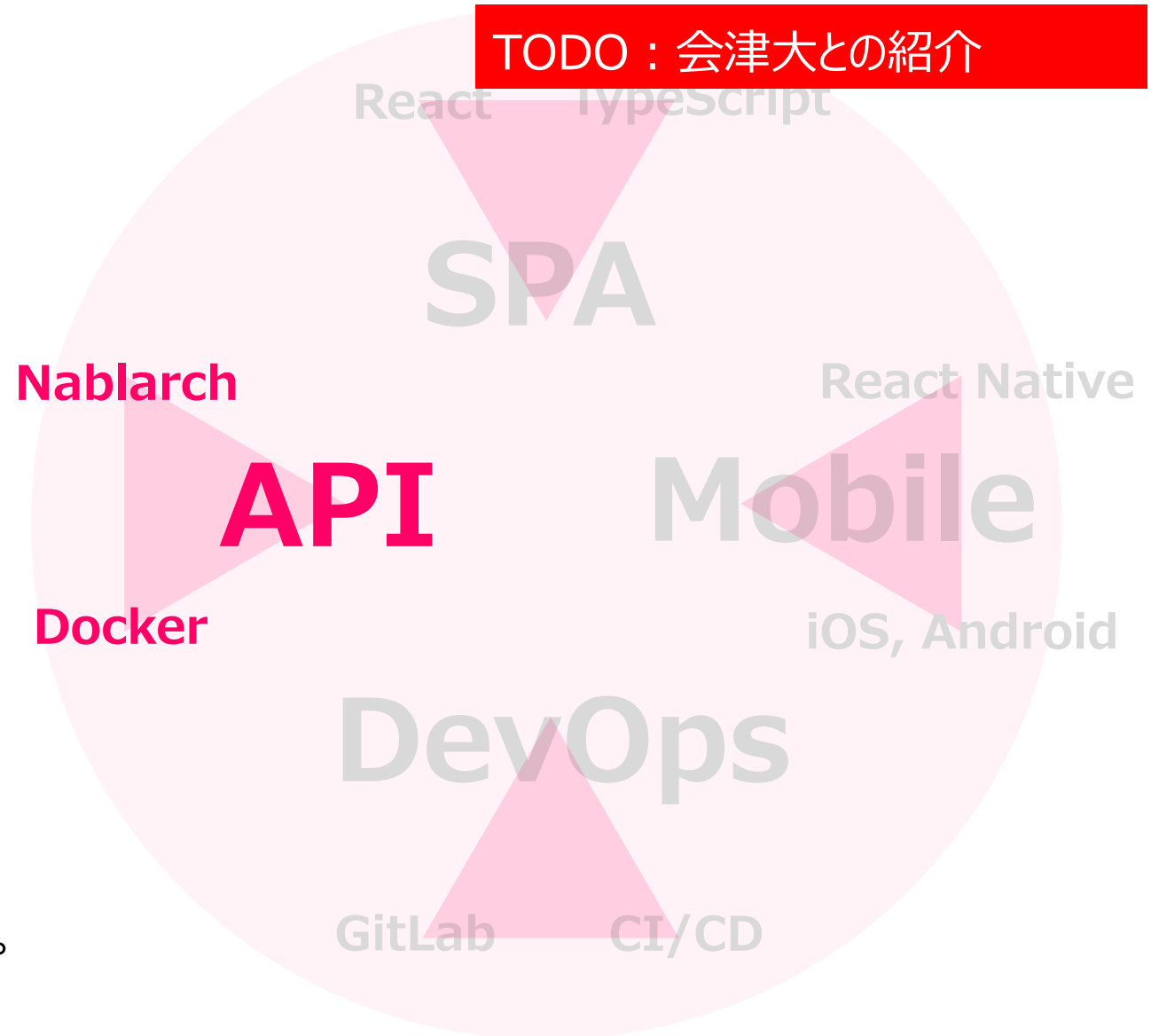
9月 SPAハンズオン

10月 APIハンズオン

11月 モバイルハンズオン

12月 DevOpsハンズオン

次回は10月にSPAのバックエンドとなる
APIの作り方を学ぶハンズオンを開催予定です。
ぜひご参加ください。



We're Hiring!

TODO：採用の紹介

TIS株式会社

テクノロジー＆イノベーション本部（T&I）

地方で働きたいエンジニアを募集してます！

地方（会津若松）でフレームワークやツール等の技術開発を行うエンジニア

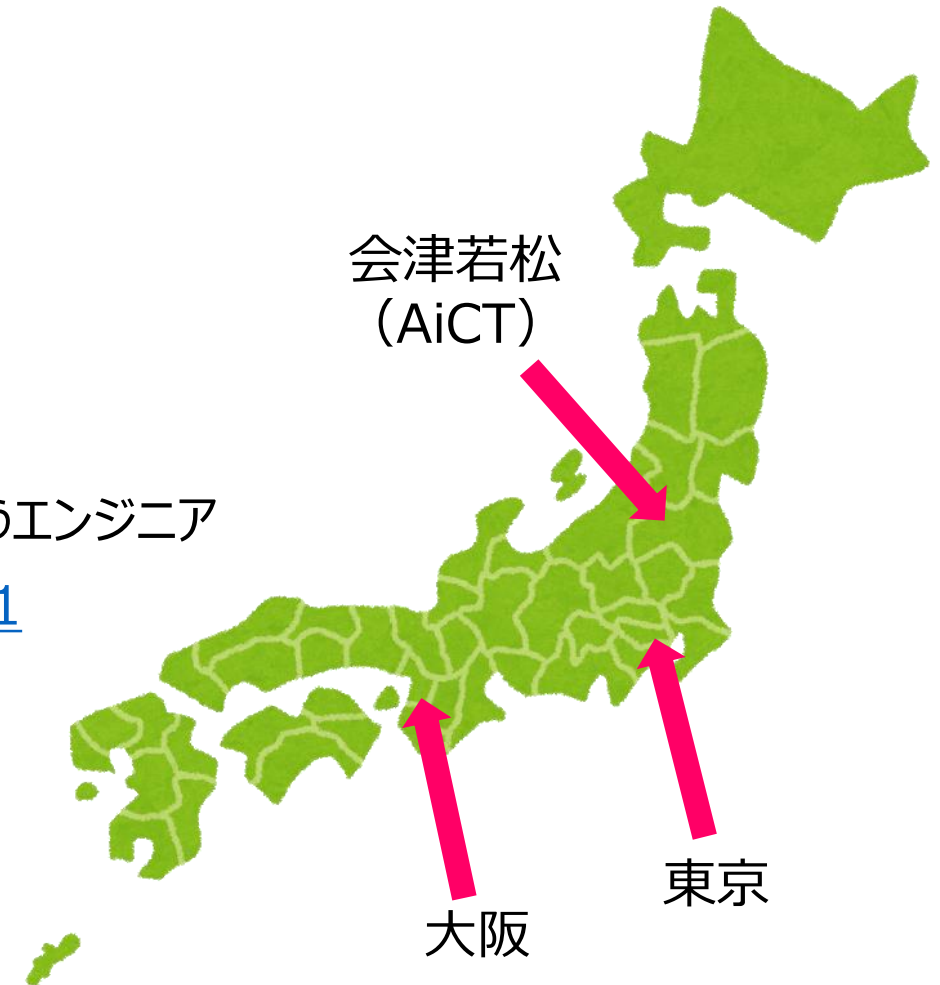
<https://hrmos.co/pages/tissaiyo/jobs/20100400011>

新卒採用も募集しています！

東京、大阪で経験を積んで、そのままでもいいし、

U/Iターンで会津若松でもいいし、技術力で勝負しませんか！

少しでも興味があれば、まずはお話ししましょう。



地図はいらすとやを使用しています。

<https://www.irasutya.com/>

アンケート

今後の改善に活用したいのでアンケートへのご協力をお願いします。

アンケートのURLはZoomのチャットで連絡します。

A scenic landscape at sunset. A paved road curves through a green field towards a distant town and mountains. A black signpost with white Japanese characters '恋人坂' (Koi no Saka) stands on the left. The sky is filled with soft, colorful clouds in shades of blue, orange, and pink. Overlaid on the image is the text 'Thank you Service Dev Engineer TAIKEN' in large, bold letters.

Thank you
Service Dev Engineer TAIKEN