React | Descomplica | Projeto

Projeto

Vamos criar um sistema que o usuário pode salvar “pins” em pastas para organizar esses conteúdos.

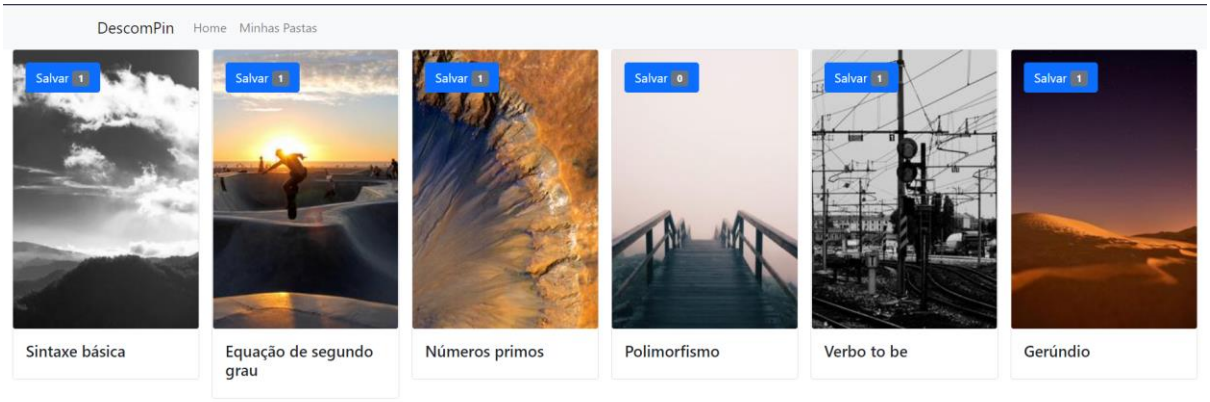
As pastas são uma forma de organizar esses pins e elas são persistidas no local storage para que possa ser acessadas posteriormente.

O projeto consiste em duas páginas:

1. **Home** - possui as principais funcionalidades
2. **Minhas Pastas** - contém lista de pastas e o número de pins salvos dentro de cada uma

Tela inicial

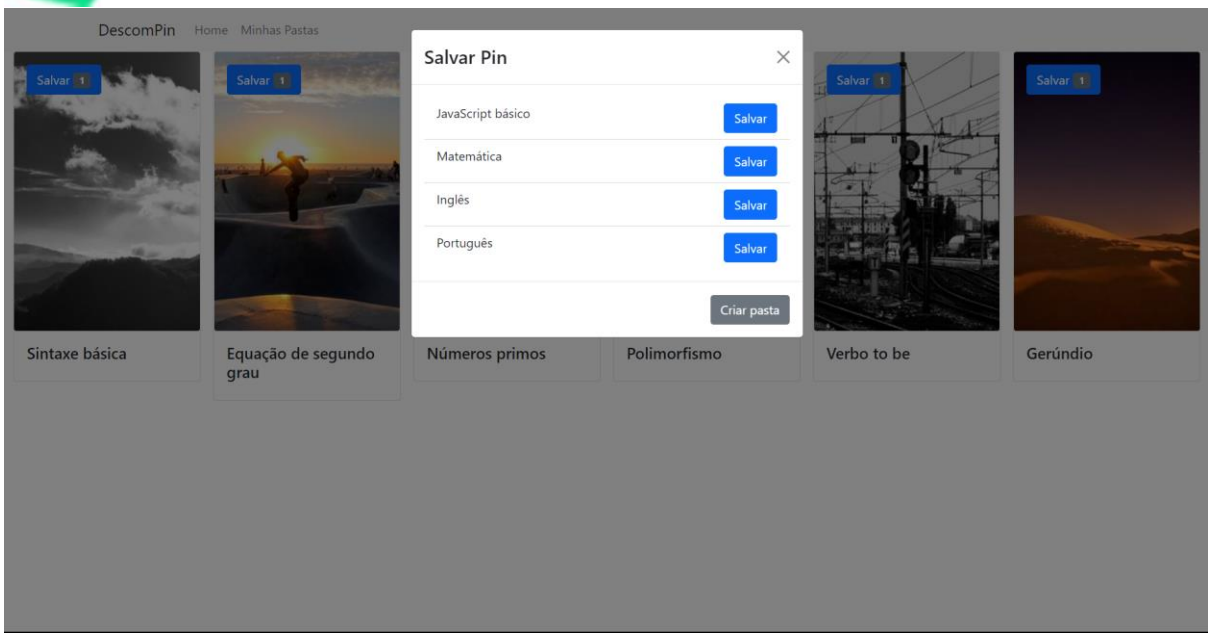
A home possui uma lista de pins. Cada pin possui um botão para salvar em uma determinada pasta. O número ao lado do botão é a quantidade de pastas que esse pin está salvo.



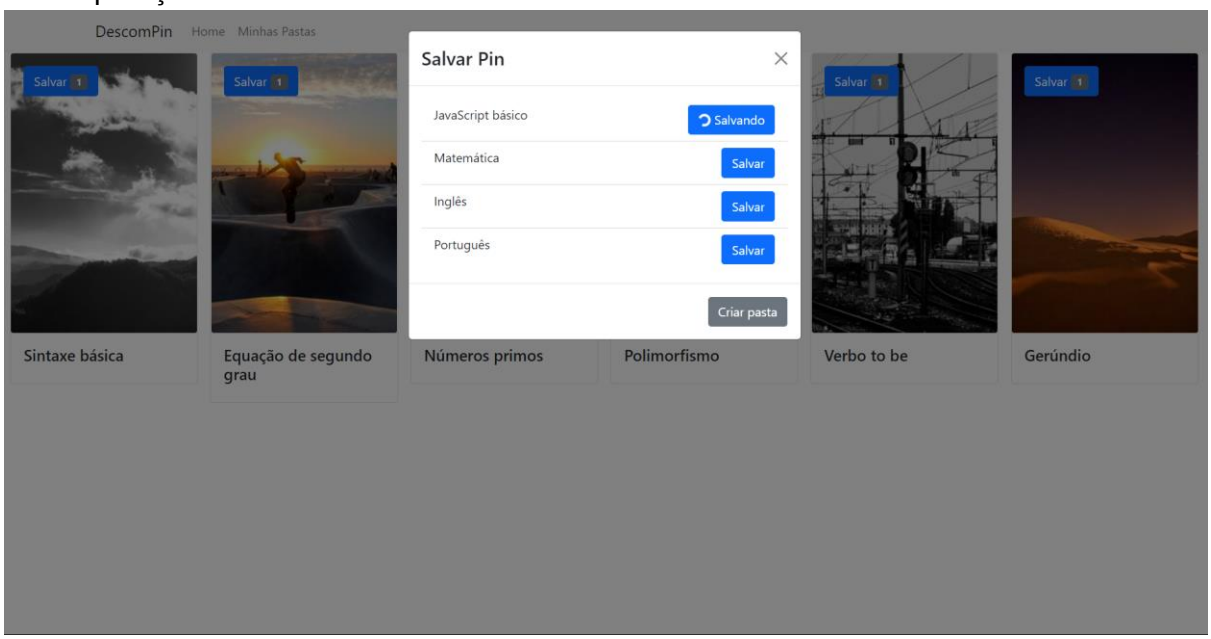
Modal Salvar pin

Após acessar o botão “Salvar” de alguns dos pins, abre o modal para escolher a pasta que deseja salvar ou criar uma pasta.

descomplica

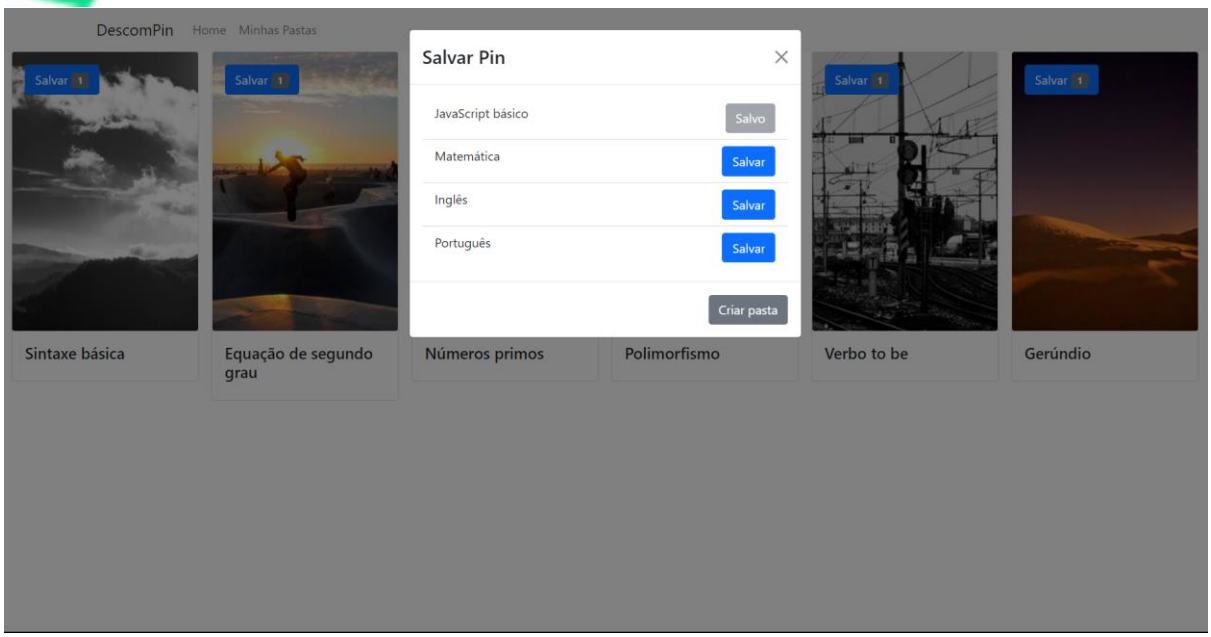


Ao clicar em “Salvar”, o botão fica em estado de “Salvando” com um GIF de loading ao lado até a operação ser concluída.



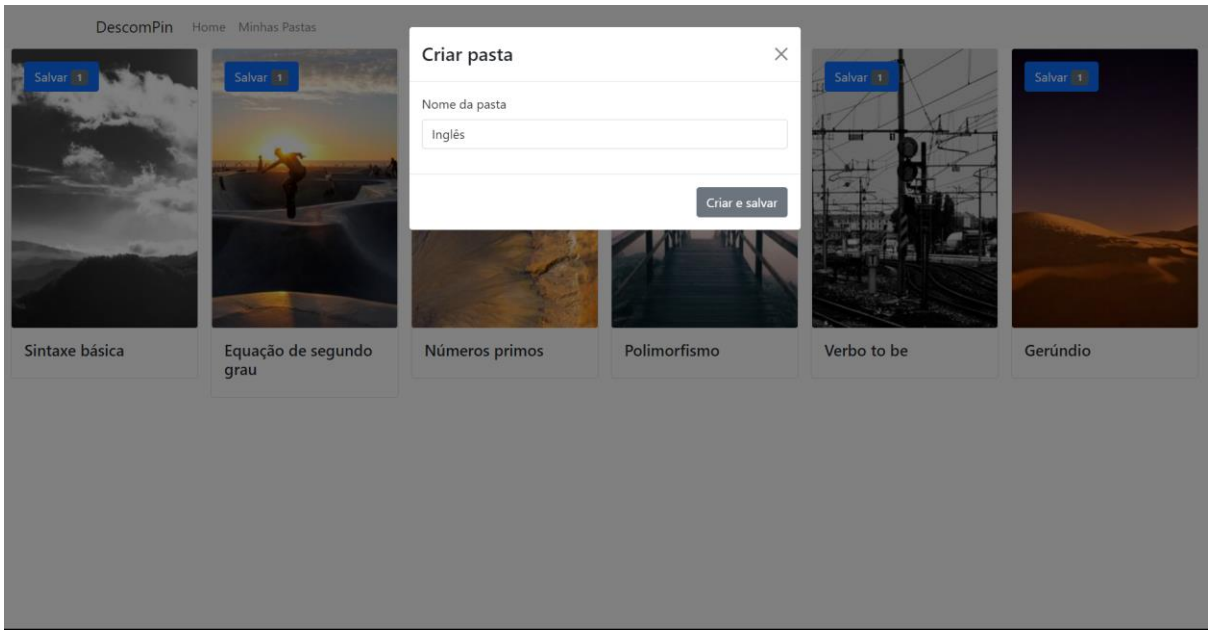
Após salvo, o botão fica como “Salvo” e desabilitado.

descomplica



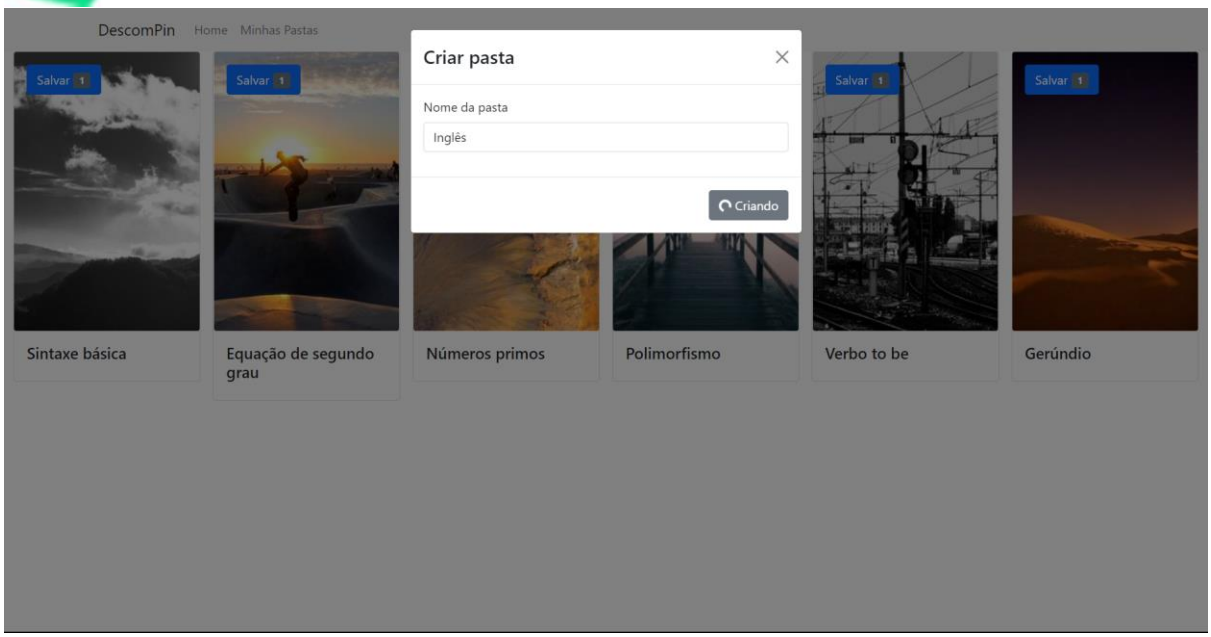
Modal criar pasta

Ao clicar na etapa anterior em “Criar pasta”, um novo modal é aberto onde você digita o nome da pasta que deseja criar e salvar o pin ativo

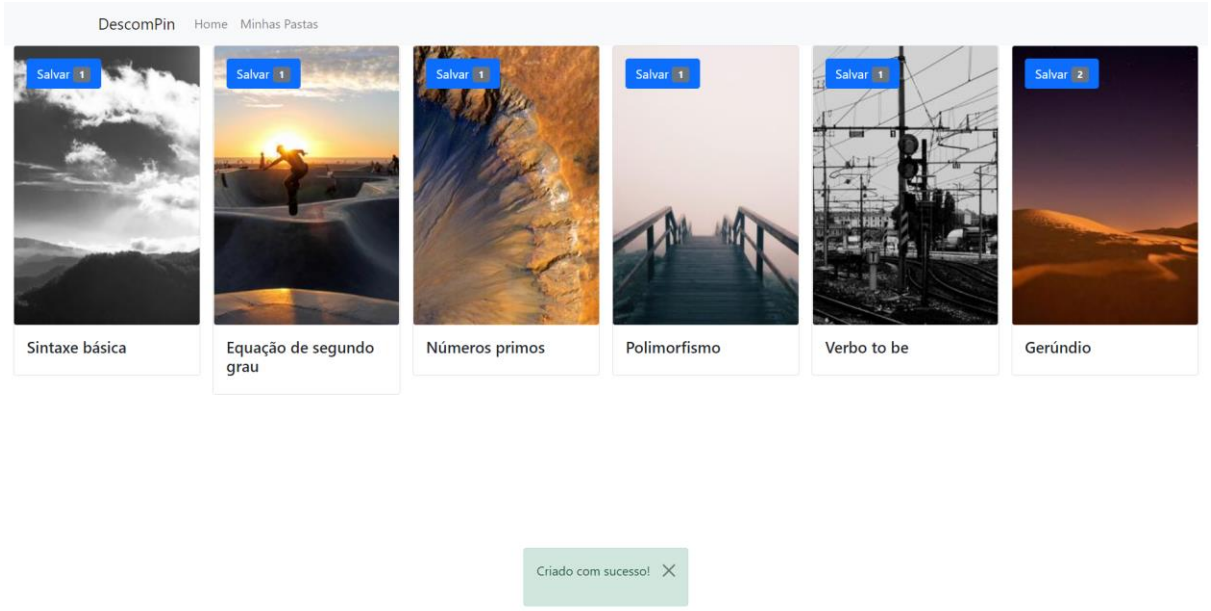


Ao clicar em “Criar e salvar”, o botão entra em estado de “Criando” com gif de loading ao lado, indicando que a operação está em execução.

descomplica



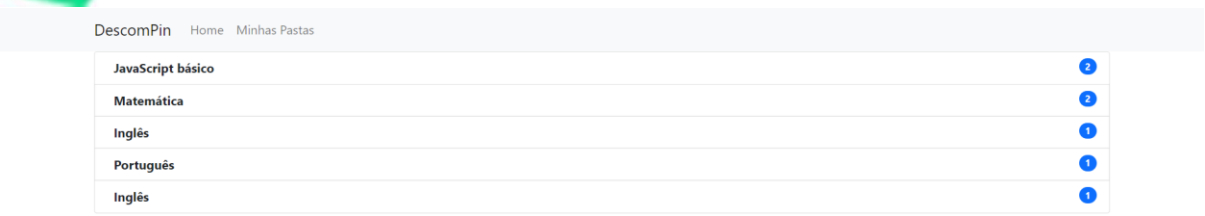
Após a pasta ser criada, o modal é automaticamente fechado e um “Alerta” é exibido na parte central inferior da home indicando que foi criado com sucesso.



Página Minhas Pastas

Nessa página podemos ver todas as pastas criadas e ao lado do seu nome, a quantidade de pins salvo lá dentro.

descomplica



O que vamos fazer (definindo funcionalidades)

Imagina que isso aqui é um protótipo junto com todo o layout pronto fornecido pelo UX do seu time.

Precisamos analisar as funcionalidades, listá-las para assim pensar em como vai ser organizado e desenvolvido.

Passando pelo protótipo fornecido pelo UX, podemos destacar as funcionalidades

- Página Home
    - Listar pins (buscar pins em algum lugar)
        - Botão com número de pastas salvas
            - Abre modal salvar pin
    - Modal salvar pin
        - Lista pastas que já foram salvas e disponíveis
        - Salvar pin em pasta
            - Loading
        - Botão criar pastas (abre modal)
    - Modal criar pasta
        - Digitar nome da pasta
        - Criar pasta com pin dentro
        - Fechar modal
        - Mostrar alert que foi criado com sucesso
- Página minhas pastas
    - Listar minhas pastas
        - Número de pins salvos

descomplica

Como vamos organizar tudo isso?

Quando nos deparamos com uma aplicação completa para ser desenvolvida, pode ficar confuso sobre como começar e como organizar tudo.

No passo anterior, nós destacamos as funcionalidades que o projeto apresenta, agora precisamos quebrar pensando mais na parte técnica do desenvolvimento, pensar em alguns passos importantes para a construção do projeto.

De uma forma ainda um pouco macro, podemos listar as etapas:

1. Definir componentes
2. Definir arquitetura
3. Definir modelagem dos dados
4. Definir modelagem do estado
5. Construir páginas e componentes estáticos
6. Criar serviço para salvar pins e pastas (armazenamento/storage)
7. Conectar componentes com estado da aplicação
8. Criar rotas para as páginas
9. Fazer tudo se comunicar

Vamos focar nos itens do 1 ao 4, que são mais de planejamento

Definir componentes

Nós vamos utilizar o *react-bootstrap* para interface do nosso projeto.

A ideia é olhar o protótipo e definir quais componentes vamos criar e quais do bootstrap se encaixa

Pode ser que um componente nosso carregue 1, 2, 3 ou mais componentes do bootstrap

Componentes bootstrap que vamos utilizar

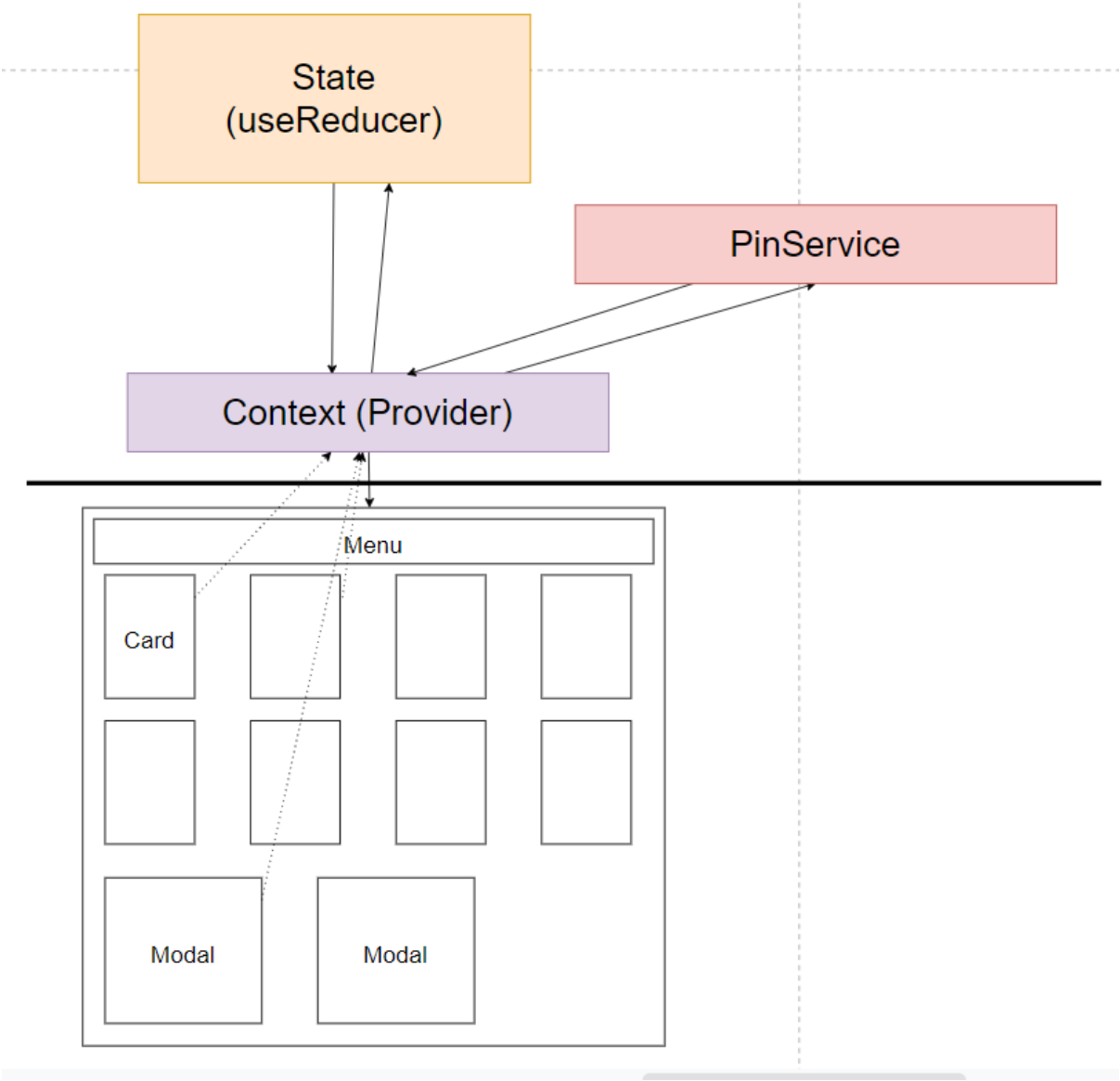
Vamos utilizar o React Bootstrap para construir nossa interface. Com base no protótipo do projeto, podemos identificar quais componentes do React Bootstrap vamos precisar:

- Card
- Button
- Spinner
- Modal
- List Group
- Badge
- Form (e Input)
- NavBar
- Alert

descomplica

Arquitetura

Podemos separar a arquitetura em duas camadas principais: Componentes e Estado.



A parte de baixo são os componentes em si, o JSX. A parte de cima é onde gerenciamos e propagamos as informações do estado global.

Para gerenciar o estado global da aplicação vamos utilizar o hook *useReducer***.** Para distribuir as informações do estado global vamos utilizar Context API.

Vamos criar o serviço PinService que também fica na camada de cima e seu valor também vai ser distribuído via Context API.

O Context API junto com o *useReducer* providenciam a forma como os componentes vão requisitar os métodos do PinService. Ou seja, em nenhum momento nossos componentes usam o PinService diretamente

descomplica

Modelagem de dados

Precisamos modelar dados pois vamos persistir no LocalStorage as pastas criadas e os pins salvos dentro delas. Assim, caso o usuário faça o refresh da página, os dados não serão perdidos.
Se vamos armazenar em algum lugar, temos que definir a sua estrutura para que seja fácil de trabalhar, de acessar e de salvar.

No projeto temos dois recursos principais:

- Pin
- Pasta

Pin

```
{
  id: string,
  title: string,
  description: string,
  url: string,
  image: string
}
```

Pasta

```
{
  id: string,
  name: string,
  pins: [pinId]
}
```

Modelagem do estado

A modelagem de estado é uma das principais partes na hora de desenvolver em React. Tudo é baseado no estado para que os componentes reflitam ele.

Sem estado do componente, tudo fica estático, nada se mexe, nada é reativo

Você precisa analisar minuciosamente para identificar os gatilhos que fazem os componentes reagirem e quais informações na tela se alteram.

Exemplo de algumas coisas que ficam no estado:

- Se o modal está aberto
- Qual modal está aberto
- Se o botão está em loading
- Quais pins devem renderizar na tela
- Quais pastas devem aparecer no modal

Outro ponto importante é saber identificar o que fica no **estado global compartilhado** e o que é restrito apenas ao **estado de um componente**.

Vamos utilizar *useReducer* para trabalhar o estado e Context API para propagar os valores para os componentes e fazerem se comunicar

React extra

No decorrer do projeto, exploramos dois recursos da biblioteca que não foram abordados nas aulas de fundamentos React, que são:

1. Portals
2. Hook useReducer

Vamos agora explorar esses dois tópicos em detalhes.

React Portals

Como você viu, sempre que criamos um componente React e o utilizamos, ele renderiza sempre exatamente onde você está chamando ele. Isso parece óbvio, e é, mas, pode não ser o desejado.

Você pode precisar chamar um componente em um lugar mas deseja que ele renderize em outro. Isso acontece principalmente quando falamos de modais.

Muitas vezes precisamos ativar um modal para que ele apareça na tela, mas o local onde precisamos fazer isso não é onde ele precisa ser renderizado. O próprio modal do react-boostrap implementa Portals por baixo dos panos.

Portals é simples no React, basicamente você cria um componente mas já define onde ele vai ser renderizado, que pode não ser no lugar onde está sendo chamado. *Portals* é de fato como se fosse um portal para um outro lugar.

Para criar um componente com *Portal*, temos que utilizar a função *createPortal* do *ReactDOM*. Veja que a função é do ReactDOM e não do React, por justamente se tratar de renderização diretamente.

descomplica

```
const MeuComponente = (props) => {
  return (
    ReactDOM.createPortal(
      <div>
        <h4>{props.label}</h4>
      </div>,
      document.body
    )
  )
}
```

Observe que a criação do componente é bastante similar, a diferença é que invocamos a função *createPortal* que aceita dois argumentos, o primeiro é o componente em si e o segundo é o local onde vai ser renderizado.

Agora podemos utilizar o *MeuComponente* em qualquer lugar, mas ele vai ser renderizado no final do *body* e não onde está sendo chamado de fato.

```
const App = () => (
  <div classname="app">
    <header>
      <MeuComponente label="Título portal" />
    </header>
    <section>
      <p>Texto</p>
    </section>
  </div>
)
```

Hook useReducer

Nós aprendemos até agora a utilizar o hook *useState* para deixar as coisas reativas no React.

Porém, nós vamos gerenciar o estado global da aplicação, e o *useState* serve para trabalhar com unidades do estado, agora precisamos trabalhar com um bloco de estado.

Em geral, você vai utilizar muito mais o *useState* do que o *useReducer*.

Em aplicações maiores, você vai ver sendo utilizado bibliotecas inteiras para gerenciar o estado, como Redux e Mobx. Mas nesse caso, o *useReducer* vai ser o suficiente

Mesmo que você utilize algo como Redux, o uso de *useReducer* não é descartado, muito menos o do Context API.

Redux + useReducer + Context podem ajudar juntos e separados

Você aprendendo *useReducer* vai já ter uma noção de quando for trabalhar com Redux pois alguns conceitos são parecidos, com as *actions* e *reducer*.

O que é useReducer?

Igualmente *useState* serve para trabalhar o estado do componente

O *useReducer* é utilizado para gerenciar um bloco de estado ao mesmo tempo, quando as informações estão relacionadas e precisam ser atualizadas ao mesmo tempo, já o *useState* lida com uma informação por vez.

É até possível utilizar o *useState* e passar um objeto para ele lidar com isso e vai funcionar. Porém, o *useReducer* tem funcionalidades mais sofisticadas para isso.

Quando não utilizar?

Em casos simples o *useReducer* traz sofisticação mas também traz complexidade, então evite em cenários simples.

Prática

Vamos voltar a um código que fizemos na aula de hooks para refatorar o *useState* para *useReducer*.

descomplica

```
import { useState } from 'react'

const Assento = (props) => {
  const [ disabled, setDisabled ] = useState(false);

  const handleClick = () => {
   setDisabled(true);
  }

  return (
    <button
      disabled={disabled}
      type="button"
      onClick={handleClick}
      className="assento"
      >
        {
          disabled
            ? <span>-</span>
            : <strong>{props.posicao}</strong>
        }
      </button>
  )
}
```

Agora vamos trocar o *useState* pelo *useReducer*

```
import { useReducer } from 'react'; // Importando useReducer

const Assento = (props) => {
  // Definindo estado inicial
  const initialState = { disabled: false };

  //Invocando useReducer
  const [ state, dispatch ] = useReducer(reducer, initialState);

  //...
}
```

Como falamos, o *useReducer* serve para tratar estados mais complexos, com mais dados. Assim, o seu estado inicial sempre vai ser um objeto com a estrutura desejada. Lembrando que o *useState* também recebe um estado inicial, seja uma string vazia, um booleano, etc.

A função *reducer* pode ser definida fora do componente

descomplica

Como a estrutura do estado inicial do *useReducer* pode ser grande, por convenção a gente cria um constante só para listar o estado.

O próximo passo é invocar o *useReducer* que aceita dois argumentos:

1. Função que modifica o estado
2. Estado inicial

Por enquanto ainda não definimos a função, faremos logo.

O *useReducer*, assim como *useState*, também retorna um array onde aplicamos desestruturação. O primeiro elemento do array é o estado (inicialmente seu valor é igual ao estado inicial *initialState*), e o segundo elemento do array é a função que você invoca toda vez que deseja alterar o estado.

```
import { useReducer } from 'react';

// Função reducer que recebe dois argumentos
function reducer (state, action) {
  return state;
}

const Assento = (props) => {
  const initialState = { disabled: false };

  const [ state, dispatch ] = useReducer(reducer, initialState);

  //...
}
```

A função modificadora de estado *reducer* recebe dois parâmetros:

1. O estado atual
2. Um objeto que detalha o que precisa ser modificado no estado

O objetivo da função *reducer* é sempre retornar o estado, seja ele o mesmo ou modificado. Por isso, criamos uma função simples que já retorna o próprio *state*. Logo vamos voltar nessa função para adicionar alguma lógica.

Como já deu para perceber, não existe mais o state *disabled* que usávamos antes. Agora essa informação fica dentro do objeto *state*. Por isso, vamos refatorar o JSX para utilizar do *state* e não mais o *disabled* diretamente.

descomplica

```
const Assento = (props) => {
  const initialState = { disabled: false };

  const [ state, dispatch ] = useReducer(reducer, initialState);

  const handleClick = () => {
    // Comentando setDisabled pois não existe mais
    // setDisabled(true);
  }

  return (
    <button
      disabled={state.disabled} // Alterando para state.disabled
      type="button"
      onClick={handleClick}
      className="assento"
      >
        {
          disabled
            ? <span>-</span>
            : <strong>{props.posicao}</strong>
        }
      </button>
  )
}
```

Veja que no código, comentamos a chamada ao *setDisabled* pois não existe mais, vamos substituir pela função *dispatch* criada ao invocar *useReducer*. Também alteramos a propriedade *disabled* do *button* para usar o objeto *state*.

Agora vamos refatorar a função *handleClick* para chamar o *dispatch* gerado pelo *useReducer*. Como já sabemos, o *dispatch* serve para modificar o estado gerado pelo *useReducer*.

Quando você invoca *dispatch*, a função que criamos chamada *reducer* vai ser invocada, pois é justamente ela que modifica o estado.

Modifique a função *handleClick* do componente para:

```
const handleClick = () => {
  dispatch({
    type: 'habilita status botao'
  });
}
```

Veja que estamos passando um objeto para o *dispatch*. Como propriedades e valores desse objeto você pode passar o que quiser, mas como convenção, sempre passamos duas propriedades:

1. **type** - descreve a ação/mudança que deseja realizar
2. **payload** - qualquer dado/informação que precise para modificar o estado

No exemplo acima só precisamos do **type** pois só com a descrição já é possível modificar o estado.

Agora vamos modificar a função *reducer* que até agora ela só retorna o próprio estado, ou seja, ela não modifica nada até então.

```
function reducer (state, action) {
  if (action.type === 'habilita status botao') {
    return {
      disabled: true
    }
  }
  return state;
}
```

Como falamos, a função *reducer* recebe o *state* (estado atual) e a *action*. A *action* é o objeto que passamos ao invocar o *dispatch*.

Na implementação acima, nós sabemos o que precisamos modificar devido a *action*. Por isso verificamos pelo *type*, caso seja de um tipo específico desejado, fazemos a mudança requerida. No caso, só retornamos um novo estado com *disabled* definido para *true*.

Se você invocar *dispatch* passando um *type* que não existe, ele só vai ignorar o *if* e vai retornar o próprio *state* atual.

A grande vantagem do *useReducer* é que podemos implementar diversos modificadores de estado e também combinar dados. Vamos implementar agora um toggle de status do botão. Toggle (alternar) basicamente servem para alternar um valor booleano, caso seja *true* ele vira *false* e vice-versa.

descomplica

```
function reducer (state, action) {
  if (action.type === 'habilita status botao') {
    return {
      disabled: true
    }
  }

  if (action.type === 'alterna status botao') {
    return {
      disabled: !state.disabled
    }
  }

  return state;
}
```

Acima nós adicionamos mais uma condicional para tratar o novo *type*. Caso caia nessa nova condicional, ele retorna um novo *state* definindo o valor de *disabled* para a negação do *disabled* atual (se for *true* vira *false* e se for *false* vira *true).*

Agora, onde precisar, você pode invocar o *dispatch* passando esse novo *type.* Com isso, nosso botão é ativado e desativado quando você clica nele.

```
dispatch({
  type: 'alterna status botao'
});
```

O caso mais comum de você ver uma função *reducer* é utilizando *switch case* ao invés de *if*

descomplica

```
function reducer (state, action) {
  switch (action.type) {
    case 'habilita status botao':
      return {
        disabled: true
      };

    case 'alterna status botao':
      return {
        disabled: !state.disabled
      }
  }
  return state;
}
```

Revisando o código inteiro

descomplica

```
import { useReducer } from 'react';

function reducer (state, action) {
  switch (action.type) {
    case 'habilita status botao':
      return {
        disabled: true
      };

    case 'alterna status botao':
      return {
        disabled: !state.disabled
      }
  }
  return state;
}

const Assento = (props) => {
  const initialState = { disabled: false };

  const [ state, dispatch ] = useReducer(reducer, initialState);

  const handleClick = () => {
    dispatch({
      type: 'alterna status botao'
    });
  }

  return (
    <button
      disabled={state.disabled}
      type="button"
      onClick={handleClick}
      className="assento"
      >
        {
          disabled
            ? <span>-</span>
            : <strong>{props.posicao}</strong>
        }
      </button>
  )
}
```

State com múltiplas informações

Praticamente em todos os casos que você user *useReducer*, vai ter mais de uma informação no estado. Até porque, se precisar carregar uma única informação nele, a melhor escolha seria usar o *useState*.

Vamos simular que junto do *disabled* do botão, queremos por um título no estado que vai servir para renderizar o título do componente.

```
// Movi o initialState para fora do componente pois não precisa estar lá.
Junto foi adicionado a propriedade titulo.
const initialState = {
  titulo: 'Meu Componente',
  disabled: false
};

const Assento = (props) => {
  const [ state, dispatch ] = useReducer(reducer, initialState);

  const handleClick = () => {
    dispatch({
      type: 'alterna status botao'
    });
  }

  return (
    <div classname="container">
      {/* Renderizando o titulo do state */}
      <h2>{state.titulo}</h2>
      <button
        disabled={state.disabled}
        type="button"
        onClick={handleClick}
        className="assento"
        >
          {
            disabled
              ? <span>-</span>
              : <strong>{props.posicao}</strong>
          }
      </button>
    </div>
  )
}
```

Se você não refatorar a função *reducer*, vai perceber que quando clicar no botão do assento, o título vai sumir. Isso acontece porque nossa função *reducer* está retornando um novo estado apenas com a informação do *disabled*. Por isso, precisamos refatorar para a função sempre retornar os dados que estavam no estado antes, modificando apenas o que precisa.

descomplica

```
function reducer (state, action) {
  switch (action.type) {
    case 'habilita status botao':
      return {
        ...state, // merge das informações
        disabled: true
      };

    case 'alterna status botao':
      return {
        ...state, // merge das informações
        disabled: !state.disabled
      }
  }
  return state;
}
```

Agora você pode criar uma *action* para modificar o título, exemplo:

```
function reducer (state, action) {
  switch (action.type) {
    case 'habilita status botao':
      return {
        ...state,
        disabled: true
      };

    case 'alterna status botao':
      return {
        ...state,
        disabled: !state.disabled
      }

    // Nova action para alterar título
    case 'altera titulo':
      return {
        ...state,
        titulo: action.payload
      }
  }
  return state;
}
```

Nesse novo *case* de *action* que criamos, também utilizamos a propriedade *payload*. Ou seja, toda vez que você ficar um *dispatch* passando o *type 'alterar titulo'*, você também deve passar o *payload* com o novo título, ficando assim:

```
dispatch({
  type: 'alterna titulo',
  payload: 'Descomplica Título'
});
```