Refactoring React

Patterns and practices

Who am I

Inga Pflaumer

Software engineer at Next Practice

VP of Muses Code JS

Organiser of Sydney Unity Game Dev meetup

Corgi owner

Workshop

- https://github.com/Rukia3d/IntroToJS-Marvel
- clone project
- branches:
  - `master` - initial state
  - `fixed` - final state

The Project

- Marvel Universe API
- The team selection part of a fighting game
- Created with TDD
- Needs Refactoring

The Project

- Created with create-react-app
- Runs on localhost:3000
- You will need node.js
- if you didn’t do npm install - do it now

The Project

- Open browser
- Open terminal and project folder in the terminal (2 windows)
- **npm run start** to start the project
- **npm run test** to run tests

Use case

- User searches for their favourite characters
- Adds them to the team
- Original characters can be highlighted
- Characters can be deleted
- Search panel can be hidden
- When the team is formed - game starts

The Project (API)

- originally had API requests
- replaced by captain.json for this workshop
- will read captain.json for any search
- API calls and vars are commented out
- Images were replaced by local images
- you will need your own API key if you want to try it

The Project

- **App.js** Entry point. Creates Characters and Search
- **Search.js** Search form
- **Item.js** Shows the search items
- **Characters.js** Displays NoCharacters or ShowCharacters
- **ShowCharacters.js** Displays and paginates Characters

Refactoring

- Do you have tests?
- If no - WRITE TESTS FIRST
- If yes - check the coverage
- If it’s not good enough - write more tests

App.test.js

- No characters selected
- Enters “Captain”. Search results are in
- Adds Captain America. No pages. Pic + Details are in.
- Adds 5 more Captains. Pagination test.
- Deletion test.
- Collapse Search test.

App.test.js

- What’s missing?
- No toggle test.
- Decision: Ignore it (bad decision!)
- You need to know what you decided to ignore.
- And how it can affect the result.

General Refactoring

- Separate classes
- Rename variables (use let and const!)
- Make functions more readable
- Add PropTypes
- etc…

React Refactoring

- Anonymous functions
- When do we re-render?

Anonymous functions

- Will create a copy for every render.
- Not a problem on a small project.
- What if you’re making a Stadium seat booking system? With a function generated for every seat on every render?

Anonymous functions

- The easiest way to fix anonymous functions is to pass data via events.
- Be specific. Only transfer data you need, not whole objects.

Use Pure Component

Controllable re-rendering - React.PureComponent

PureComponent changes the life-cycle method `shouldComponentUpdate` and adds some logic to automatically check whether a re-render is required for the component. This allows a PureComponent to call method `render` only if it detects changes in `state` or `props`, hence, one can change the state in many components without having to write extra checks like:

https://60devs.com/pure-component-in-react.html

Item.js

Controllable re-rendering - React.PureComponent

```
class Item extends React.PureComponent {
```

General takeaway

- Change variable names
- Split logic into smaller functions
- Use destructuring to make props readable
- Use Prop.Types

General takeaway

- Check your anonymous functions. Use events.
- Give references instead of objects where possible.
- Re-render only when you need to.

React Patterns

- Nested components
- Render props
- High order components
- Hooks

- Problem: We have collapsed component on the Search, it’s not strictly related to Search functionality and later we want it on ShowCharacters.

- Solution: **nested components**

Nested components

- No dependancy between the parent and the child
- Parent doesn’t control the data passed to the child

Nested components

- Collapser component doesn’t care what’s in Search
- Can be separated
- Search can be rendered inside
- Can use the same approach in ShowCharacter

Nested components

- Step 1: Separate Search and Collapser inside of the Search
- Step 2: Move Collapser into a separate component
- Step 3: Add Collapser to App and make Search it’s child
- Step 4: Apply the same approach to ShowCharacter

Search.js

- Step 1: Separate Search and Collapser inside of the Search

Search.js

```
  render() {
    return (
      <div className={this.state.collapse? "open": "closed"} data-testid="collapse-block">
        <button type="button"
          className="btn btn-danger"
          onClick={this.collapseBlock}
          data-testid="collapse-button">
        </button>
        <div className="row">
          <div className="col-lg-12">
          <form className="bs-component" onSubmit={this.search}>…
          {this.state.loading…
        </div>
      </div>
    );
  }
}
```

Collapser.js

- Step 2: Move Collapser into a separate component

Collapser.js

```
class Collapser extends Component {

  constructor(props){
    super(props);
    this.state = {
      collapse: true,
    }
    this.collapseBlock = this.collapseBlock.bind(this);
  }

  collapseBlock(){
    this.setState((oldState) => ({collapse: !oldState.collapse}));
  }

  render(){
    return(
      <div className={this.state.collapse? "open": "closed"} data-testid="collapse-block">
        <button type="button"
          className="btn btn-danger"
          onClick={this.collapseBlock}
          data-testid="collapse-button">
        </button>
      </div>
    )
  }
}

export default Collapser;
```

Search.js

- Step 2b: Remove the collapser state and collapser code from Search.js

Search.js

```
  render() {
    return (
        <div className="row">
          <div className="col-lg-12">
          <form className="bs-component" onSubmit={this.search}>
          {this.state.loading
        </div>
    );
  }
}

export default Search;
```

App.js

- Step 3: Add Collapser to App and make Search it’s child

App.js

```
  render() {
    return (
      <div className="App">
        <Nav/>
        <div className="container">
          <Characters chars={this.state.characters} removeCharacter={this.removeCharacter}/>
          <Collapser>
            <Search add={this.addCharacter} />
          </Collapser>
        </div>
      </div>
    );
  }
}
```

App.js

- Step 3b: Render Search as a child of Collapser

Collapser.js

```
  render(){
    return(
      <div className={this.state.collapse? "open": "closed"} data-testid="collapse-block">
        <button type="button"
          className="btn btn-danger"
          onClick={this.collapseBlock}
          data-testid="collapse-button">
        </button>
        {this.props.children}
      </div>
    )
  }
}

export default Collapser;
```

Homework

- Step 4: Apply the same approach to ShowCharacter

- Problem: Collapser needs to directly control the Search to show summary of the hidden component

- Solution: **render props**

Render props

- Parent directly controls the child
- Parent passes the data to the child

Render props

- Example: react router

```
// wrapping/composing
// You can spread routeProps to make them available to your rendered Component
const FadingRoute = ({ component: Component, ...rest }) => (
  <Route {...rest} render={routeProps => (
    <FadeIn>
      <Component {...routeProps}/>
    </FadeIn>
  )}/>
)

<FadingRoute path="/cool" component={Something}/>
```

Render props

- Step1: Change the App.js so Collapser can render Search
- Step2: Change Collapser so it renders the Search
- Step3: Add collapsed as props from Collapser to Search
- Step4: Render Search based on the collapsed prop

Render props

- Step1: Change the App.js so Collapser can render Search

App.js

```
  render() {
    return (
      <div className="App">
        <Nav/>
        <div className="container">
          <Characters chars={this.state.characters} removeCharacter={this.removeCharacter}/>
          <Collapser render={() => (<Search add={this.addCharacter} />)}/>
        </div>
      </div>
    );
  }
}

export default App;
```

Render props

- Step2: Change Collapser so it renders the Search

Collapser.js

```
  render(){
    return(
      <div className={this.state.collapse? "closed": "open"} data-testid="collapse-block">
        <button type="button"
          className="btn btn-danger"
          onClick={this.collapseBlock}
          data-testid="collapse-button">
        </button>
        {this.props.render()}
      </div>
    )
  }
}

export default Collapser;
```

Render props

- Step3: Add collapsed as props from Collapser to Search

App.js

```
render() {
  return (
    <div className="App">
      <Nav/>
      <div className="container">
        <Characters chars={this.state.characters} removeCharacter={this.removeCharacter}/>
        <Collapser render={(collapsed) => (<Search add={this.addCharacter} collapsed={collapsed} />)}/>
      </div>
    </div>
  );
}
}

export default App;
```

Collapser.js

```
  render(){
    return(
      <div className={this.state.collapse? "closed": "open"} data-testid="collapse-block">
        <button type="button"
          className="btn btn-danger"
          onClick={this.collapseBlock}
          data-testid="collapse-button">
        </button>
        {this.props.render(this.state.collapse)}
      </div>
    )
  }
}

export default Collapser;
```

Render props

- Step4: Render Search based on the collapsed prop

Search.js

```
    renderSearch(){
      return (
        <div className="row">
          <div className="col-lg-12">
          <form className="bs-component" onSubmit={this.search}>…
          {this.state.loading…
        </div>
      )
    }

    render() {
      return (
        this.props.collapsed
        ? <div className="row"> <h3>Here's the summary</h3></div>
        : this.renderSearch()
      );
    }
  }

  export default Search;
```

- Problem: Show a placeholder while the image loads
- Solution: **higher order component**

Higher-order component

- Additional functionality for the component

Higher-order component

- Step 1: Separate IMG component
- Step 2: Wrap image in the function. This function will render the original component with additional functionality
- Step 3: Add placeholder functionality
- Step 4: Replace the image with the function

Higher-order component

- Step 1: Separate IMG component

ShowCharacters.js

```
const Img = ({name, src}) => (
  <img
    style={{maxWidth: "18rem"}}
    data-testid="picture"
    alt={name}
    src={src}
  />
)

const Character = (props) => (
  <div className="card text-white bg-primary mb-3"
      style={{maxWidth: "20rem"}}
      data-testid="character">
    <div className="card-body">
      <h4 className="card-title" data-testid="name">{props.character.name}</h4>
      <Img
        name={props.character.name}
        src={props.character.thumbnail.path+"."+props.character.thumbnail.extension}
      />
      <p className="card-text" data-testid="descr">{props.character.description}</p>
      <button⋯
      >⋯
    </div>
  </div>
);
```

Higher-order component

- Step 2: Wrap image in the function. This function will render the original component with additional functionality

ShowCharacters.js

```
function withPlaceholder(Image) {
   return class extends Component{
     render(){
       return <Image {...this.props} />
     }
   }
}

const Img = ({name, src}) => (
  <img
    style={{maxWidth: "18rem"}}
    data-testid="picture"
    alt={name}
    src={src}
  />
)

const ImgWithPlaceholder = withPlaceholder(Img);
```

Higher-order component

- Step 3: Add placeholder functionality

```react
import React, { Component } from 'react';
import logo from './logo.svg';
const groupSize = 3;

function withPlaceholder(ImageComp) {
   return class extends Component{
     constructor(props){
       super(props);
       this.state = {
         loaded: false,
       }
     }

     componentDidMount(){
       const img = new Image();
       img.src = this.props.src;
       img.onload = () => {
         this.setState({
           loaded: true,
         })
       }
     }

     render(){
       const { src, ...other} = this.props;
       let url = this.state.loaded ? src : logo;
       return <ImageComp src={url} {...other} />
     }
   }
}
```

Higher-order component

- Step 4: Replace the image with the function

ShowCharacters.js

```
const Character = (props) => (
  <div className="card text-white bg-primary mb-3"
      style={{maxWidth: "20rem"}}
      data-testid="character">
    <div className="card-body">
      <h4 className="card-title" data-testid="name">{props.character.name}</h4>
      <ImgWithPlaceholder
        name={props.character.name}
        src={props.character.thumbnail.path+"."+props.character.thumbnail.extension}
      />
      <p className="card-text" data-testid="descr">{props.character.description}</p>
      <button⋯
      >⋯
    </div>
  </div>
);
```

- Problem: We want to model different behaviours in one component.
    - Render props lead to increased nesting
    - Higher-order components are not obvious to read
- Solution: **Hooks because we can**

React hooks

- They are hipstery!

React hooks

- Step 1: Decide what hooks you need for what component. Img in this case
- Step 2: Return Img render, clean up the withPlaceholder ref.
- Step 3: Use useState to set the url
- Step 4: Use useEffect for the functionality of componentDidMount
- Step 5: Make it reusable

React hooks

- Step 1: Decide what hooks you need for what component. Img in this case

ShowCharacters.js

```
import React, { Component, useState, useEffect } from 'react';
import logo from './logo.svg';
const groupSize = 3;
```

React hooks

- Step 2: Return Img render, clean up the withPlaceholder ref.

ShowCharacters.js

```
const Character = (props) => (
  <div className="card text-white bg-primary mb-3"
      style={{maxWidth: "20rem"}}
      data-testid="character">
    <div className="card-body">
      <h4 className="card-title" data-testid="name">{props.character.name}</h4>
      <Img
        name={props.character.name}
        src={props.character.thumbnail.path+"."+props.character.thumbnail.extension}
      />
      <p className="card-text" data-testid="descr">{props.character.description}</p>
      <button⋯
      >⋯
    </div>
```

React hooks

- Step 3: Use useState to set the url

ShowCharacters.js

```
import React, { Component, useState, useEffect } from 'react';
import logo from './logo.svg';
const groupSize = 3;

const Img = ({name, src}) => {
  const [url, setUrl] = useState(logo);

  return (
    <img
      style={{maxWidth: "18rem"}}
      data-testid="picture"
      alt={name}
      src={url}
    />
  )
}

const Character = (props) => (
  <div className="card text-white bg-primary mb-3"
      style={{maxWidth: "20rem"}}
      data-testid="character">
    <div className="card-body">
      <h4 className="card-title" data-testid="name">{props.character.name}</h4>
      <Img
```

React hooks

- Step 4: Use useEffect for the functionality of componentDidMount

ShowCharacters.js

```
const Img = ({name, src}) => {
  const [url, setUrl] = useState(logo);

  useEffect(() => {
    const img = new Image();
    img.src = src;
    img.onload = () => setUrl(src);
  });

  return (
    <img
      style={{maxWidth: "18rem"}}
      data-testid="picture"
      alt={name}
      src={url}
    />
  )
}
```

useEffect()

- There's a GOTCHA and we will come back to it in a sec!

React hooks

- Step 5: Make it reusable
    - We want to use a hook to render a placeholder or the correct loaded data ANYWHERE
    - Let’s make our own hook!

usePlaceholder.js

- Make a new file with a hook name and move functionality there

usePlaceholder.js

Search.js | Collapser.js | App.js | App.css | ShowCharacters.js | usePlaceholder.js | bc

```
import { useState, useEffect } from 'react';
import logo from './logo.svg';

const usePlaceholder = (src) => {
  const [url, setUrl] = useState(logo);

  useEffect(() => {
    const img = new Image();
    img.src = src;
    img.onload = () => setUrl(src);
  });

  return url;
}

export default usePlaceholder;
```

usePlaceholder.js

- Make Img use the new hook.

ShowCharacters.js

Search.js | Collapser.js | App.js | App.css | ShowCharacters.js | usePlaceholder.js

```
import React, { Component } from 'react';
import usePlaceholder from './usePlaceholder';
const groupSize = 3;

const Img = ({name, src}) => {
  const url = usePlaceholder(src);

  return (
    <img
      style={{maxWidth: "18rem"}}
      data-testid="picture"
      alt={name}
      src={url}
    />
  )
}
```

usePlaceholder.js

- You can refactor it further, remove Img and use hook inside of Img in Character

usePlaceholder.js

- GOTCHA!

```
const usePlaceholder = (src) => {
  const [url, setUrl] = useState(logo);

  useEffect(() => {
    const img = new Image();
    img.src = src;
    img.onload = () => setUrl(src);
    console.log("I AM USEEFFECT");
  });

  return url;
}
```

- Find characters and check your console to see how many time it re-renders.

useEffect.js

- GOTCHA!

- UseEffect executes twice
  - When component mounts
  - When state changes (call setUrl)

useEffect.js

- GOTCHA!

  - To remedy this useEffect takes an optional second argument - “only re-render if this changes”

useEffect.js

- GOTCHA!

```
const usePlaceholder = (src) => {
  const [url, setUrl] = useState(logo);

  useEffect(() => {
    const img = new Image();
    img.src = src;
    img.onload = () => setUrl(src);
    console.log("I AM USEEFFECT");
  }, [setUrl, src]);

  return url;
}
```

Some helpful reading

Refactoring

Improving the Design of Existing Code

By: Martin Fowler

Thank you!

Find me here:

- Twitter: @IngaPflaumer
- Medium: @IngaPflaumer
- LinkedIn: Inga Pflaumer