Loop the Loop: Iterating over data structures

Next week we are considering the following question:

Session 1:

“How might we iterate over data structures and display them in the DOM?”

Session Dependencies

Prior to starting the session, you should clone the latest version of the ongoing project:

- Open the project in VS code, add command in terminal :

```
git clone -b Session-1-solution https://github.com/Amytrainer/React-Goal-Tracking-Application.git
```

- Install the dependencies, npm install
- You can then start the application, npm run start

From next week forward, we are incrementally going to be working on our goal tracking application

By the end next week, you should:

- Know how to iterate over lists and display your data in the DOM
- Understand how JSX facilitates displaying lists

# Why do we need to iterate over data structures?

Typically, front-end applications receive data from the back-end in a list format. Typically, in the context of Web Development, this list will take the form of a JSON array:

```
[
  {
    "name": "Joe",
    "location": "Brighton"
  },
  {
    "name": "Martin",
    "location": "Bournemouth"
  }
]
```

An example JSON object

You should be familiar with JSON. It's simply a way to represent data. As you can see, it is very similar to JavaScript. In fact, it's easy to process JSON:

```
const obj = JSON.parse(
  '[{"name": "Joe","location": "Brighton"},{"name":
"Martin","location": "Bournemouth"}]'
);
console.log(obj);

/*Prints to the console:
  [ { name: 'Joe', location: 'Brighton' },
  { name: 'Martin', location: 'Bournemouth' } ]
 */
```

An example of converting a JSON object to an array of JavaScript object literals

# Displaying iterative data in React

Often you will have to iterate over some data structures, like the one above, to construct your React views, this is very easy to achieve in JSX.

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) => (
  <li key={number.toString()}>{number}</li>
));
```

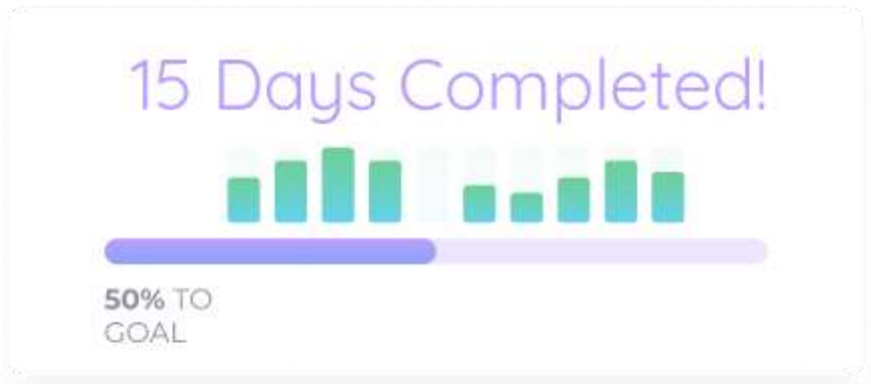
We can simply use an array map function (see above) to render array structured data into our views

(opens new window).

Notice how we are embedding pure JavaScript into our views.

# A Real World Example

Let's revisit what our completed DaysCompleted component should look like:



To add further context, each histogram bar represents a percentage score from the last 10 days. As you can imagine, this information will be receive from an external data source. However, we don't really know what that is going to look like yet. Until this is the case, we will have to mock our data.

Commonly, several components will need to reflect changes in data. Remember, data in a React application flows one-way, from parent to child. As such, it is common practice to lift the state to a top-level component (e.g. App.js).

Task 1

# Task 1 Mocking Data 🚀

Let's create a data structure that contains check-in scores out of 20. Within App.js, create the following data structure:

```
const checkins = [
  {
    date: "Wed Jan 29 2020 07:17:11 GMT+0000 (Greenwich Mean Time)",
    score: 20,
  },
  {
    date: "Wed Jan 28 2020 07:17:11 GMT+0000 (Greenwich Mean Time)",
    score: 15,
  },
  { date: "Wed Jan 27 2020 07:17:11 GMT+0000 (Greenwich Mean Time)", score: 8 },
  { date: "Wed Jan 26 2020 07:17:11 GMT+0000 (Greenwich Mean Time)", score: 2 },
  {
    date: "Wed Jan 25 2020 07:17:11 GMT+0000 (Greenwich Mean Time)",
    score: 20,
  },
  {
    date: "Wed Jan 23 2020 07:17:11 GMT+0000 (Greenwich Mean Time)",
    score: 12,
  },
  {
    date: "Wed Jan 22 2020 07:17:11 GMT+0000 (Greenwich Mean Time)",
    score: 19,
  },
  {
    date: "Wed Jan 21 2020 07:17:11 GMT+0000 (Greenwich Mean Time)",
    score: 10,
  },
  {
    date: "Wed Jan 20 2020 07:17:11 GMT+0000 (Greenwich Mean Time)",
    score: 15,
  },
  { date: "Wed Jan 19 2020 07:17:11 GMT+0000 (Greenwich Mean Time)",
score: 6 },
  {
    date: "Wed Jan 18 2020 07:17:11 GMT+0000 (Greenwich Mean Time)",
    score: 20,
  },
  {
    date: "Wed Jan 17 2020 07:17:11 GMT+0000 (Greenwich Mean Time)",
    score: 20,
  },
  {
    date: "Wed Jan 16 2020 07:17:11 GMT+0000 (Greenwich Mean Time)",
    score: 20,
  },
  {
    date: "Wed Jan 15 2020 07:17:11 GMT+0000 (Greenwich Mean Time)",
    score: 20,
  },
];
```

Task 2

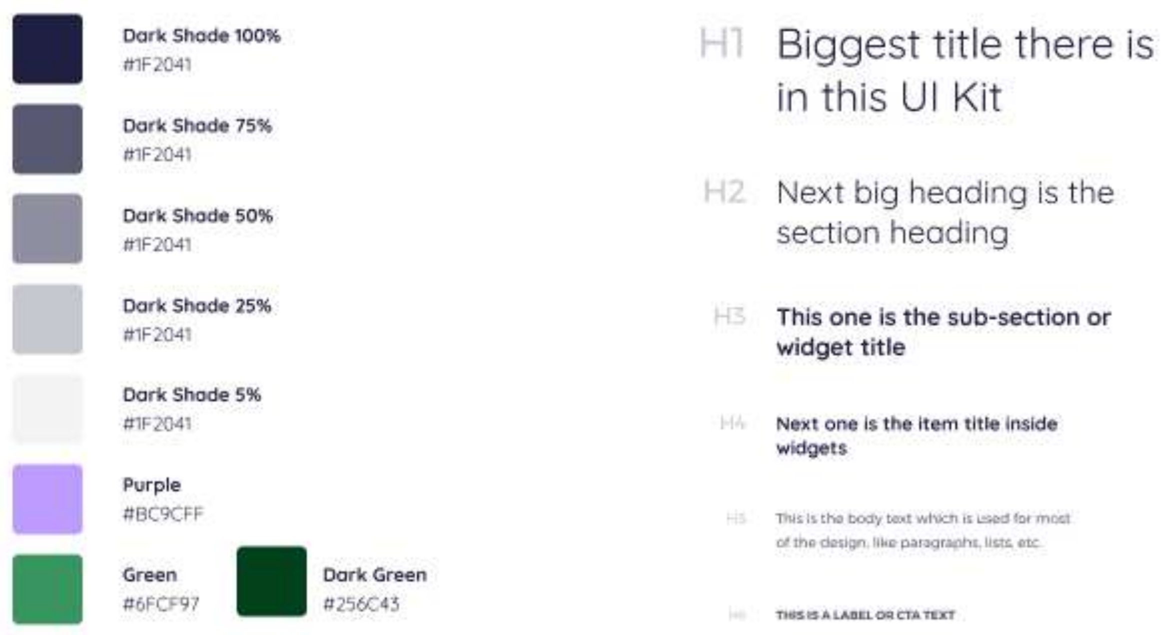
# Task 2 Completing DaysCompleted component 🚀

- Complete the histogram component for you. Data should feed into DaysCompleted first.
- Finally, create the progress bar. To do this you will need to create and use a new component called ProgressBar
- This part of our application should meet the following user stories:
- As a user, I should be able to view how many days have been completed in text and as a percentage, so I know how close to my goal I am
- As a user, I should be able to view a histogram of, up to, my last 10 check-ins, so I can track my progress as a daily percentage out of 20

Session 2:

The core question: **"How can we ensure a consistent look and feel across our applications?"**

There are, of course, several ways to address the above question. To start with, a simple style guide can go a long way in presenting a solution.


Dark Shade 100%
#1F2041
Dark Shade 75%
#1F2041
Dark Shade 50%
#1F2041
Dark Shade 25%
#1F2041
Dark Shade 5%
#1F2041
Purple
#BC9CFF
Green
#6FCF97
Dark Green
#256C43
H1 Biggest title there is in this UI Kit
H2 Next big heading is the section heading
H3 This one is the sub-section or widget title
H4 Next one is the item title inside widgets
H5 This is the body text which is used for most of the design, like paragraphs, lists, etc.
H6 THIS IS A LABEL OR CTA TEXT

The above style guide,lays the foundation colour schemes and font sizings. Crucially, it is in a language developers can quickly understand.

# A Style Guide to A Living Themes

We need a way of converting our style guide, above, into some sort of theme. We can, of course, use CSS, however, it somewhat lacks sophistication. Pre-processors, such as Syntactically Awesome Style Sheets (SASS), addressed some of the limitations of CSS by extending its capabilities allowing control structures and variables to be used.

SASS solved a lot of problems and still should be considered a resonable solution for simple sites that require little to no JavaScript functionality. However, in this course, we are working with single-page web applications and we must assume ourselves and any developer on the project will be well versed in JavaScript.

As such, while we could use SAS, JavaScript is far more powerful and can achieve everything SASS can and more. In realising this idea, recently we have seen a rise in CSS-in-JS libraries. Combining CSS with our JS is an odd concept, to begin with. However, unlike SASS, there is no need to learn a different syntax - you already know JavaScript. In summary, CSS-in-JS presents a very shallow learning curve and the upside of a powerful extension to traditional CSS.

# CSS-in-JS

You have already used CSS-in-JS. For instance, consider setting the inline style attribute equal to some object:

```
const innerBar = {
    background:
    ...
    height: `${percentage}%`,
    ...
};

return (
    <div style={barStyle}>
        <div style={innerBar}></div>
    </div>
);
```

Creating inline styles using JavaScript objects, is one way of working using a CSS-in-JS pattern, and a perfectly reasonable one. However, many third-party libraries provide more sophisticated solutions (see https://github.com/MicheleBertoli/css-in-js).

While, as mentioned above, there are many styling solutions. https://styled-components.com/

(opens new window), for me, presented itself as a simple yet powerful theming solution for React Applications. :::

# Task 1 - Understand the Why 🚀

I always like to know the philosophical position of any third-party library that I use. To access this, I consider what sort of problem they are trying to solve? Moreover, I ask myself the question; will my application, moving forward run into this problem?

For this first task, spend 5 minutes reading the motivations behind (opens new window) styled-components.

# Working with styled-components

Let's get going, the first thing we need to do is install the styled-components node package. This is a little more involved than I would like as, to get the most out of the styled-components library, we need to install and configure the babel preset.

# Task 2 - Installing styled-components 🚀

From command line use npm to install the following dependencies:

```
npm install --save-dev babel-plugin-styled-components
npm install --save styled-components
```

Next, we need to tell babel to actually use the babel-plugin-styled-components preset component. In the root of your project, create a .babelrc file and add the following snippet of JSON:

```
{
        "plugins": ["babel-plugin-styled-components"]
}
```

Remember, we used create-react-app to scaffold our application. As such, under the hood, all of our configuration is taken care of. However this presents an issue as the default config will ignore our .babel file. We can "eject" the application, this will move all of the configuration files directly into our project, however, this seems quite extreme. Instead, we are going to use customize-cra(opens new window) to edit our config without ejecting.

First, install customize-cra and its dependency react-app-rewired.

```
npm install --save-dev react-app-rewired customize-cra
```

Next, create a config-overrides.js file in your root directory and add the following code snippet

```
const { useBabelRc, override } = require("customize-cra");

module.exports = override(useBabelRc());
```

Update your package.json file to run react-app-rewired as opposed to react-scripts. The scripts section of your package.json file should read as follows:

```
"scripts": {
 "start": "react-app-rewired start",
 "build": "react-app-rewired build",
 "test": "react-app-rewired  test",
 "eject": "react-scripts eject"
 }
```

Finally, you will need to restart your web application for these changes to take effect. Phew,I appreciate this process was a little painful. However, we only need to do it once.

# Styled-Components - the basics

Styled components componetise your styles. In their words, "it removes the mapping between components and styles (opens new window)". Like React in general, this is going to be an odd concept begin with.

Let's consider a concrete example, our DaysCompleted.js. Below, I have refactored it using styles-components.

```
...
import styled from "styled-components";



function DaysCompleted(props) {
  const { days, checkins } = props;

  const DaysCompleteHeading = styled.h2`
    text-align: center;
    color: #bc9cff;
  `;

  const RootDiv = styled.div`
    display: grid;
    grid-template-columns: 0.8fr;
    grid-template-rows: 55px 80px 20px auto;
    justify-content: center;
  `;

  const GoalHeading = styled.h4`
    color: #1f2041;
  `;

  return (
    <Tile>
      <RootDiv>
      <DaysCompleteHeading> {days} Days Completed! </DaysCompleteHeading>
      <Histogram barCount={7} bars={checkins.map(c => c.score * 5)} />
      <ProgressBar />
      <GoalHeading>
        <strong>50%</strong> TO GOAL!
      </GoalHeading>
    </RootDiv>
  </Tile>
 );
}
...
```

This is very odd, what is happening above?

We've componentized our styles. styled-components creates a new component for us that includes styling information. Consider, in the above example, our DaysCompleteHeading :

```
const DaysCompleteHeading = styled.h2`
  text-align: center;
  color: #bc9cff;
`;
```

Let's try and break this down. styled.h2 is in-fact a function provided by styled-components. There is a different function for every type of html element. We are then utilising tagged template literals (a recent addition to JavaScript) to write actual CSS code to style the h2 tag. Notice how everything in the "``" is just normal css. Also, note how we are writing CSS, this is not a JavaScript object literal.

# Task 3 - Styling our application 🚀

- Style the days complete component using styled components
- Use styled-components to replace the inline styles in <Histogram> and <ProgressBar>.
- Finally, consider <Tile >, use the documentation to work out how we might export a styled component, replacing our existing wrapped component. Hint, you will need to pass in a prop to the styled component (opens new window). Further, in our inline styles, currently box-shadow is camel cased to boxShadow to allow it to be used as a JavaScript object key. Remember, styled components use standard css not the camel cased alternative.

# Global Styles

So far, I hope that I have demonstrated that styled-components offer a powerful component level styling solution. However, you may well be thinking where do my global styles go? styled-components provides us with a createGlobalStyle module to solve this problem.

Let's add some global styles to our application.

# Task 4 - Creating Global Styles

First, we need to consider where our global styles should live. There are no strict opinions on how we should structure our react applications. It is very much down to you as a developer.

I like to create a src/config folder and place my global styles there.

As such, create the file and folder src/config/GlobalStyles.js. We can now start creating some global styles:

```
import { createGlobalStyle } from "styled-components";

const GlobalStyles = createGlobalStyle`

body {
  font-family: Quicksand;
}

h1 {
  font-size: 42px;
}

h2 {
  font-size: 32px;
}

h3 {
  font-size: 32px;
}

h4 {
  font-size: 19px;
}
```

```
`;

export default GlobalStyles;
```

Notice how GlobalStyles contains the current styling rules found in, src/App.css. Now we need to use these new styles, replacing src/App.css:

At the top of src/App.js, import your GlobalStyles:

```
import GlobalStyles from "./config/GlobalStyles";
```

Next, include the GlobalStyles component to the top of your App components return statement:

```
function App() {
  return (
    <div>
      <GlobalStyles />
      <DaysCompleted days={15} checkins={checkins}>
        {" "}
      </DaysCompleted>
     </ThemeProvider>
    </div>
  );
}
```

You can now remove the App.css at the top of src/App.js, as we have replaced it with a styled component.

# Themeing

Let's revisit our style guide from earlier:

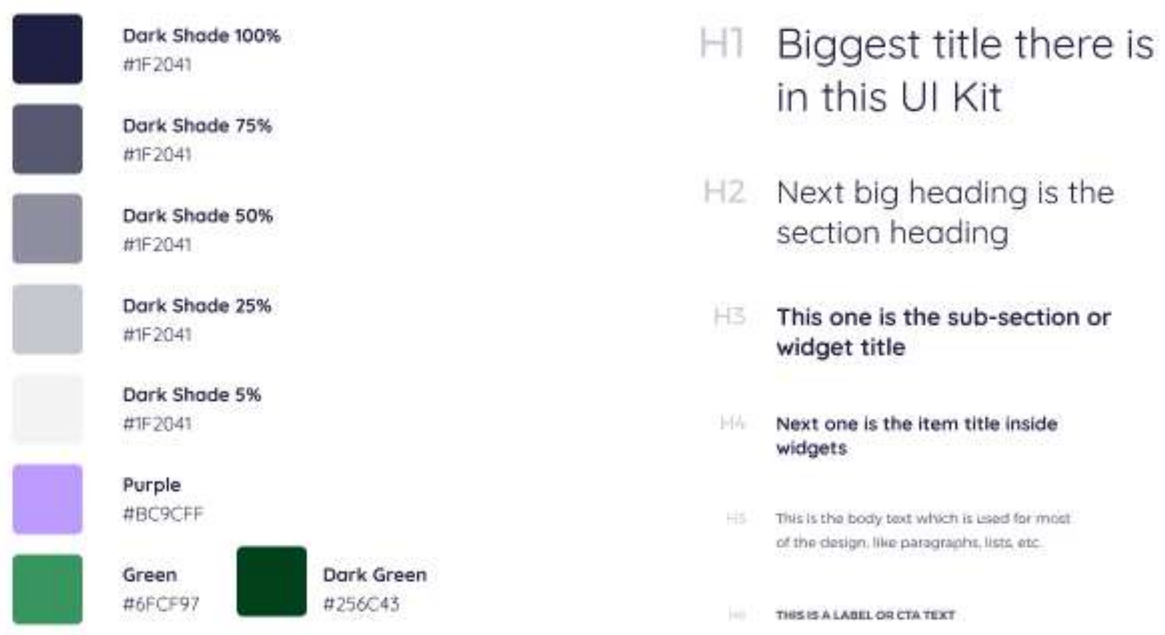

Dark Shade 100%
#1F2041
Dark Shade 75%
#1F2041
Dark Shade 50%
#1F2041
Dark Shade 25%
#1F2041
Dark Shade 5%
#1F2041
Purple
#BC9CFF
Green
#6FCF97
Dark Green
#256C43
H1 Biggest title there is in this UI Kit
H2 Next big heading is the section heading
H3 This one is the sub-section or widget title
H4 Next one is the item title inside widgets
H5 This is the body text which is used for most of the design, like paragraphs, lists, etc.
H6 THIS IS A LABEL OR CTA TEXT

Figure 1, the style guide for our tracking application (opens new window)

One of the key features that really sold me on styled-components is it allows us to represent our style guide as a theme. The theme can be constructed as a simple object.

# Task 5 - Themes 🚀

Create src/config/theme.js and add the following code:

```
const theme = {
  colors: {
    purple: "#BC9CFF",
  },
  typography: {
    fontFamily: "Quicksand",
    h1: {
      fontSize: "42px",
    },
  },
};
export default theme;
```

Above, notice how I have begun to describe our style guide, using a simple JavaScript object literal - a data structure you already know. The const theme is intentionally lower case as it is not a component, this is just a naming convention I try to follow.

styled-components provides us with a theme provider that will the theme object accessible within the styling rules.

In src\App.js import the theme and the styled-components ThemeProvider.

```
import theme from "./config/theme.js";
import { ThemeProvider } from "styled-components";
```

Next, we need to wrap our entire application in the ThemeProvider that will take our theme as a prop.

```
function App() {
  return (
    <div>
      <ThemeProvider theme={theme}>
        <GlobalStyles />
        <DaysCompleted days={15} checkins={checkins}>
          {" "}
        </DaysCompleted>
      </ThemeProvider>
    </div>
  );
}
```

That's it! The theme now globally accessible and will be passed in as a prop to all my styled components. Let's use it, in src/DaysCompleted.js, update DaysCompleteHeading to:

```
const DaysCompleteHeading = styled.h2`
  text-align: center;
  color: ${(props) => props.theme.colors.purple};
`;
```

In src/config/GlobalStyles make the following updates:

```
`
body {
  font-family:  ${(props) => props.theme.typography.fontFamily}
}

h1 {
  font-size:  ${(props) => props.theme.typography.h1.fontSize}
}
`;
```

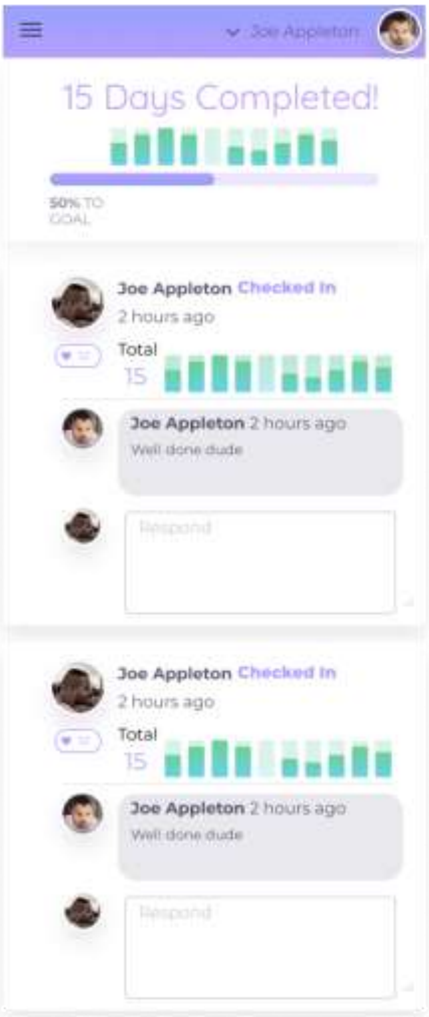
Above, you can see that our theme now gets injected into any function that we use to set a value of our css. We can then return the theme value, or, if we want return any value based on the theme.

# Task 6 - Expanding on our Theme 🚀

Think about how we might expand on our theme so it better represents the style guide above. You can get as in-depth as you like here. For inspiration, you can consider this, huge, theme object found in the material-ui library (opens new window). Your theme will be a fraction of this size. However, you should always use inspiration.

# Task 9 - Completing the Dash (if you want a challenge)

Complete the main_dash_view:



Joe Appleton

15 Days Completed!

50% TO GOAL

Joe Appleton Checked In
2 hours ago
Total
15

Joe Appleton 2 hours ago
Well done dude

Respond

Joe Appleton Checked In
2 hours ago
Total
15

Joe Appleton 2 hours ago
Well done dude

Respond