Matej Kovač Follow
Software Engineer at FlowAndForm. Relaxes writing Python code after JS hardship. @moodydev
May 15 · 7 min read

Create a custom calendar in React

> *The process from start to finish.*

When in need of a component, more often than not, we as developers will just gonna try to find if there is something out there that fits our needs. Rarely will there be a perfect fit, but we just “need something to work for a demo” and we will figure out how to make it better during the process.

And we all know how this ends, features stacking, more bloatware just to make it work and then one day you find yourself running in circles around simple tasks just because some option is missing in the component and your client really wants it.

One more problem with using component libraries is the design. By using these libraries, sites start to look generic. While we as developers may not see that because we just care about functionality, our designers won’t be too happy about it.

Creating custom components helps you create better user experiences, have more design and branding choices and you get a better

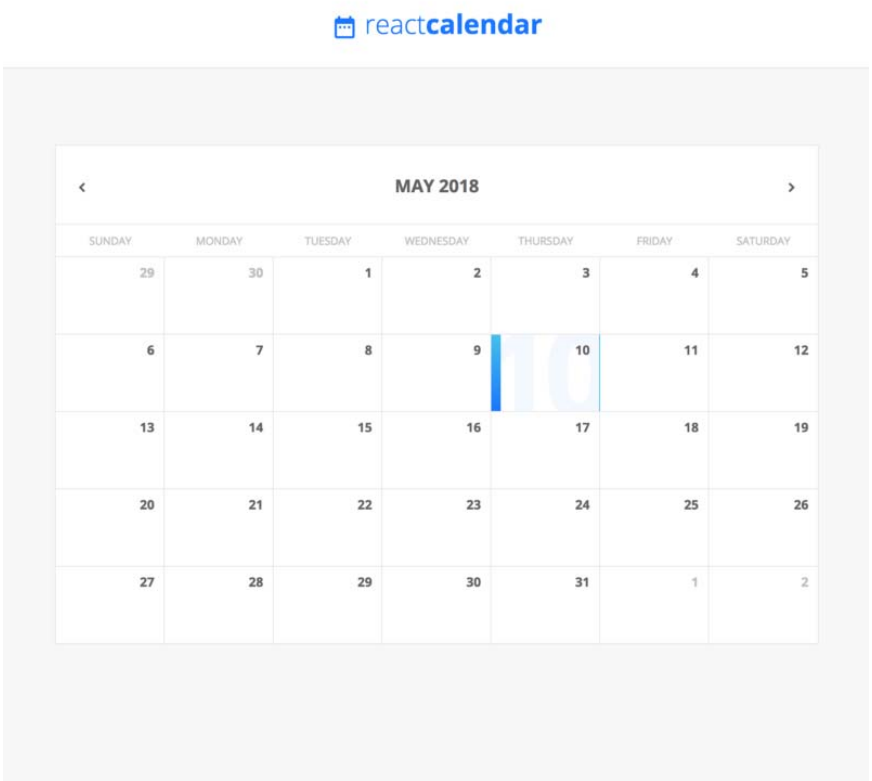
understanding of framework you use.
While you may argue creating custom components takes more time, I strongly disagree. Yes, it does take more time in the beginning, but further down the road, it saves time. You will better understand technologies you use, you will know more inner functionalities and whatever changes in component (either design or functionalities), you will have no problems implementing them (instead of using wrapper components, going to another library, etc.)

So let's start with building our own React calendar component.

What are we going to build?

We are going to build simple calendar component in which you will be able to select a date from an active month. While this is pretty simple, you can build more functionalities from there, depending on your specific needs.

I presume you have some experience with React and ES6.



Setup

We will use create-react-app for our setup

```
npx create-react-app react-calendar
cd react-calendar
```

Before starting your react app, let's first install package we will use for date handling. We will use date-fns, it's smaller than moment.js, it has really nice documentation and it's immutable.
If you are not familiar with it, go ahead and read more about it.

```
npm install date-fns
```

Once installation finished, go ahead and run you app.

```
npm start
```

Calendar Component Layout

While not important in this example, let's just create components folder and create our Component there.

```
mkdir components
cd components
touch Calendar.jsx
```

Now, let's create basic react component skeleton. We will use smart component, so that calendar component knows how to handle what month is currently displayed. Let's also import dateFns for later use.

```
import React from "react";
import dateFns from "date-fns";

class Calendar extends React.Component {
  render() {
    return (
      <div>
        <div>Header</div>
        <div>Days</div>
        <div>Cells</div>
      </div>
```

Before jumping into functionality, we need to import this component in our app and add it to render.

```
import React from "react";

import Calendar from "./components/Calendar";

import "./App.css";

class App extends React.Component {
  render() {
    return (
      <div className="App">
        <header>
          <div id="logo">
            <span className="icon">date_range</span>
            <span>
              react<b>calendar</b>
            </span>
          </div>
        </header>
```

Calendar UI

You may see we have some new CSS classes here, but since topic of this article is React, I will skip creating CSS, so go ahead and just copy contents below in your App.css

```
/* FONT IMPORTS */

@import url(https://fonts.googleapis.com/css?family=Open+
@import url(https://fonts.googleapis.com/icon?family=Mate

.icon {
  font-family: 'Material Icons', serif;
  font-style: normal;
  display: inline-block;
  vertical-align: middle;
  line-height: 1;
  text-transform: none;
  letter-spacing: normal;
  word-wrap: normal;
  white-space: nowrap;
  direction: ltr;

  -webkit-font-smoothing: antialiased;
  text-rendering: optimizeLegibility;
  -moz-osx-font-smoothing: grayscale;
  font-feature-settings: 'liga';
}


/* VARIABLES */

:root {
  --main-color: #1a8fff;
  --text-color: #777;
  --text-color-light: #ccc;
  --border-color: #eee;
  --bg-color: #f9f9f9;
  --neutral-color: #fff;
}


/* GENERAL */

* {
  box-sizing: border-box;
}
```

Calendar Functionality

We already decided our component is going to be smart and keep its state on what month is currently displayed. In our state we will need `currentMonth` and for simplicity sake, we will also keep track of currently `selectedDate`

As default we will use today's date so add to your component:

```
state = {
    currentMonth: new Date(),
    selectedDate: new Date()
};
```

In render, you can see we will have three different blocks, so lets just create empty functions for rendering.

```
renderHeader() {}

renderDays() {}

renderCells() {}
```

Thinking of functionality, we will need functions to deal with cell click to change a date and functionality to show previous or next month. While showing previous or next month can be dealt with single function by passing some value, we will add separate functions. In this case, all of those functions need to be binded, so we will use arrow functions to avoid binding in a constructor (which we already skipped using)

```
onDateClick = day => {}

nextMonth = () => {}
```

```
prevMonth = () => {}
```

All we now need is just call those render methods and wrap our div in `calendar` class. Use code below as guidance, as we will only use git file to show the final result.

```
import React from "react";
import dateFns from "date-fns";

class Calendar extends React.Component {
  state = {
    currentMonth: new Date(),
    selectedDate: new Date()
  };

  renderHeader() {}

  renderDays() {}

  renderCells() {}

  onDateClick = day => {};

  nextMonth = () => {};

  prevMonth = () => {};

  render() {
```

Rendering Calendar Header

Our header should display three things: icon for switching to the previous month, formatted date showing current month and year, and another icon for switching to next month. As mentioned previously, our icons should also handle `onClick` events to change a month.

```
renderHeader() {
```

```
    const dateFormat = "MMMM YYYY";

    return (
      <div className="header row flex-middle">
        <div className="col col-start">
          <div className="icon" onClick={this.prevMonth}>
            chevron_left
          </div>
        </div>
        <div className="col col-center">
          <span>
            {dateFns.format(this.state.currentMonth,
dateFormat)}
          </span>
        </div>
        <div className="col col-end" onClick={this.nextMonth}>
          <div className="icon">chevron_right</div>
        </div>
      </div>
    );
  }
```

Finally some output. We can see icons, but nothing happens when we click on them, so let's change that. Our functions need to take the current month from state, and either add or subtract a month from it.

```
  nextMonth = () => {
    this.setState({
      currentMonth: dateFns.addMonths(this.state.currentMonth,
1)
    });
  };

  prevMonth = () => {
    this.setState({
      currentMonth: dateFns.subMonths(this.state.currentMonth,
1)
    });
  };
```

Simple right? So what is next?

Rendering Weekday Names

To render day names in a week, we will need to figure out the start of the week, go through seven days and display their names.

date-fns offers us simple way to get startOfWeek so we will use as our starting point and just add days to it and format output.

```
renderDays() {
  const dateFormat = "dddd";
  const days = [];

  let startDate =
dateFns.startOfWeek(this.state.currentMonth);

  for (let i = 0; i < 7; i++) {
    days.push(
      <div className="col col-center" key={i}>
        {dateFns.format(dateFns.addDays(startDate, i),
dateFormat)}
      </div>
    );
  }

  return <div className="days row">{days}</div>;
}
```

So, we simply go through the loop from 0 to 6 and add an integer to our starting date, in this case, the start of the week (any week and it will work with internationalization)

Rendering Days in Month

We obviously need startOfMonth `monthStart` and endOfMonth `monthEnd` to know what dates to show. It would be also nice to show days from the previous month that complete our starting week, referred as `startDate` and days from next month that end last week of our currently selected month, referred as `endDate`

So in our `renderCells` function, we add:

```
const { currentMonth, selectedDate } = this.state;
const monthStart = dateFns.startOfMonth(currentMonth);
const monthEnd = dateFns.endOfMonth(monthStart);
const startDate = dateFns.startOfWeek(monthStart);
const endDate = dateFns.endOfWeek(monthEnd);
```

We will need to loop from `startDate` to `endDate` to show all the dates. Additionally, we will also check if current `day` in the loop is before `monthStart` or after `monthEnd` and disable clicks on it, which will be shown with different style.

To show selected date, we will also need to ask if current `day` in the loop isSameDay as `selectedDate` in our state. You don't need to worry about hours minutes or seconds, because `isSameDay` takes care of that.

Each day will also need an `onClick` event to pick clicked date as `selectedDate`

```
const dateFormat = "D";
const rows = [];

let days = [];
let day = startDate;
let formattedDate = "";

while (day <= endDate) {
  for (let i = 0; i < 7; i++) {
    formattedDate = dateFns.format(day, dateFormat);
    const cloneDay = day;
    days.push(
      <div
        className={`col cell ${
          !dateFns.isSameMonth(day, monthStart)
            ? "disabled"
            : dateFns.isSameDay(day, selectedDate) ?
"selected" : ""
        }`}
        key={day}
        onClick={() =>
this.onDateClick(dateFns.parse(cloneDay))}
      >
        <span className="number">{formattedDate}</span>
        <span className="bg">{formattedDate}</span>
      </div>
    );
    day = dateFns.addDays(day, 1);
  }
  rows.push(
    <div className="row" key={day}>
      {days}
    </div>
  );
  days = [];
}
```

While this double ternary operator may confuse you, it was laziness to change it that kept it in the code. But another side-effect of it is if you for example picked “April 30” and then go to May, that date will only show as disabled, which is fine for our purpose.

Alternatively, we could first ask if the date is selected and add `selected` class, if not, then ask if it’s disabled which would resolve the issue above.

I am also using `const cloneDay` in our loop because otherwise our `onClick` event will always take `endDate` as clicked value, as that’s the value when our loop ends because we defined `day` in the outer scope and its value is changed.

To render days, at the end od `renderCells()` add:

```
return <div className="body">{rows}</div>;
```

One last thing remains; selecting a date.
Our function already has `onClick` event, now we just need to define it. We will store it in our components state, but preferably, you will manage that in outer component that renders `Calendar`

```
onDateClick = day => {
  this.setState({
    selectedDate: day
  });
};
```

This was easy so far, so let’s continue with the hard part now…

The Hard Part

Okay, that was just to scare you off…

We are done!

As promised, this is the final code:

```
import React from "react";
import dateFns from "date-fns";

class Calendar extends React.Component {
  state = {
    currentMonth: new Date(),
    selectedDate: new Date()
  };

  renderHeader() {
    const dateFormat = "MMMM YYYY";

    return (
      <div className="header row flex-middle">
        <div className="col col-start">
          <div className="icon" onClick={this.prevMonth}>
            chevron_left
          </div>
        </div>
        <div className="col col-center">
          <span>{dateFns.format(this.state.currentMonth, (
        </div>
        <div className="col col-end" onClick={this.nextMo
          <div className="icon">chevron_right</div>
        </div>
      </div>
    );
  }

  renderDays() {
    const dateFormat = "dddd";
    const days = [];

    let startDate = dateFns.startOfWeek(this.state.curren

    for (let i = 0; i < 7; i++) {
      days.push(
        <div className="col col-center" key={i}>
          {dateFns.format(dateFns.addDays(startDate, i), (
        </div>
      );
    }
```

Final Words

This is just the basic calendar and I am sure you can think of many new functionalities to implement. Like selecting multiple dates, showing date range, adding appointments to the calendar, etc. All of which is easy to build on existing code.

Hopefully, this shows you there is nothing complicated with creating your own components. It will make you a better developer, it enhances you user's experience and makes your client happier with the overall result.
And… Your designer will love you because you can implement his/her crazy ideas and designs (true story)