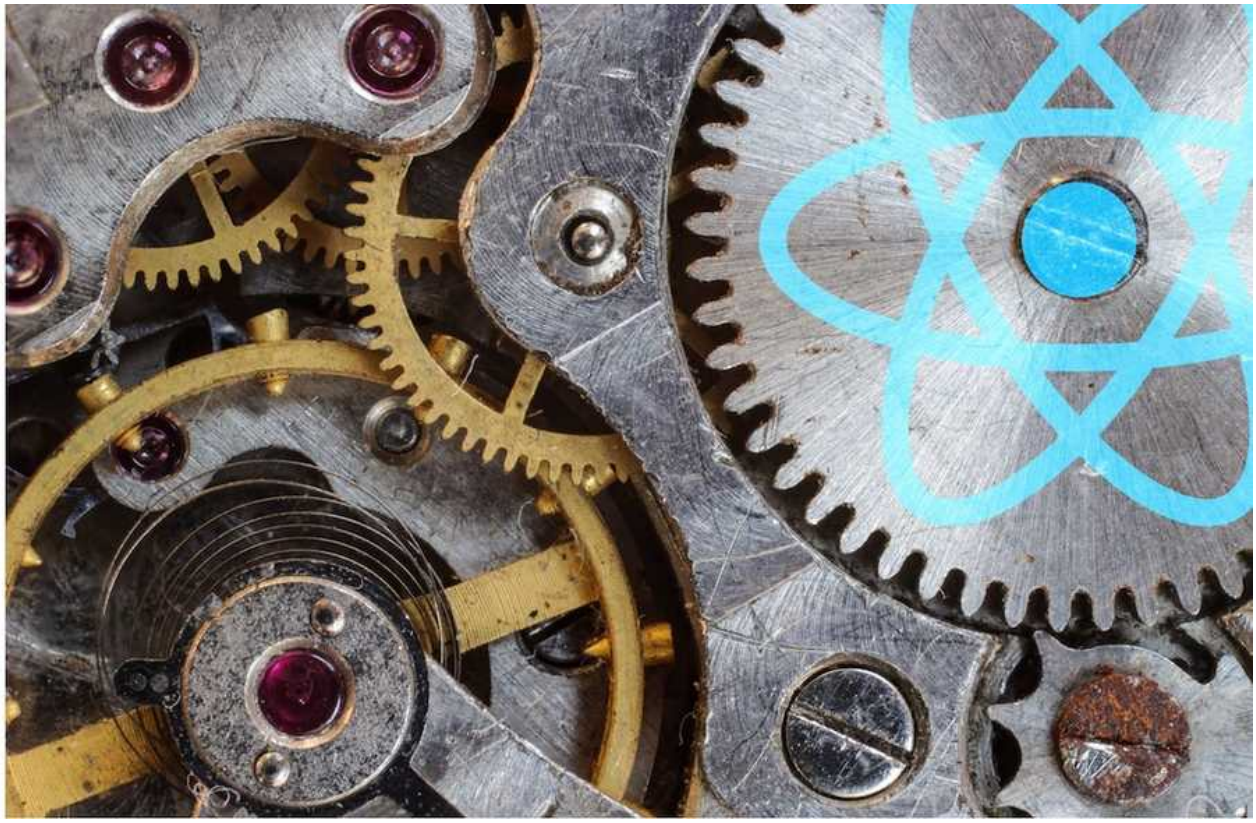
React Libraries in 2019

JANUARY 11, 2018 BY ROBIN WIERUCH - EDIT THIS POST



React has been around for a while and since then a well-rounded, also perceived as overwhelming, ecosystem has evolved around the component driven library. It's not always simple to pick a library that solves your problem in the React world, but once you get to know all the options, its ecosystem becomes very powerful. That's why I want to give every year a new overview of the most popular libraries used to complement a React application.

React only enables you to build component driven user interfaces. It comes with a couple of built-in solutions though, for instance local state management and syntactic events to make your application interactive and your interactions happen, but after all you are only dealing with components. It is often said that plain React is sufficient when building applications in JS. In the Road to learn React it is showcased that only React suffices to build an application. But in the end, when implementing a larger application, you need a few more libraries to have a sophisticated web application with React at its core.

Developers coming from frameworks such as Angular or Ember often have a hard time to figure out all the building blocks they will need to build a sophisticated web application with React. Coming from a framework, you are used to have all necessary functionalities at your disposal. However, React is only a component library. Thus you would need to figure out all the other other libraries that are needed to complement React. Nevertheless I think it is one of the crucial advantages of React in staying flexible when choosing your libraries to complement your React application. I made this experience myself when I came from Angular to React. It might help you to understand the reasons behind a move from a framework to a library.

The following article will give you an opinionated approach to select from these libraries to build a sophisticated React application. Nevertheless, it is up to you to exchange them with your own preferred libraries. After all, the article attempts to give beginners in the React ecosystem just an opinionated overview.

TABLE OF CONTENTS

REACT'S BOILERPLATE DECISION

Developers still struggle on making a decision on how to setup their React project when joining the React community. There are thousands of boilerplate projects to choose from and every boilerplate project attempts to fulfil different needs. They vary in a range of minimalistic to almost bloated projects.

The status quo in the community is by starting your project with create-react-app. It comes with a zero-configuration setup and gives you a minimalistic up and running React application out of the box. You can always decide to lay open the tool set by using its eject functionality. Afterward, you can alter the underlying tool set.

In the end, there will never be the perfect boilerplate project. You will always have to add your own tooling. That's why it makes sense, when having a solid understanding of React itself, to start of with a minimal React boilerplate project. You will be able to understand the underlying mechanics, you will do it on your own without copying a project, and you can add your own tooling to it. When choosing a bloated React boilerplate project in the first place, you will only be overwhelmed when you want to change something in the tool set.

An alternative in the ecosystem, similar to create-react-app, is Next.js. It is a zero-configuration React boilerplate as well, but for server-side rendered React. Another alternative is called Gatsby.js which is an excellent choice for static websites with React.

If you are tempted to choose a custom boilerplate for your own doings, try to narrow down your requirements. The boilerplate should be minimal and not trying to solve everything. It should be specific for your problem. For instance, the gatsby-firebase-authentication boilerplate "only" gives you a full authentication flow with Firebase in a Gatsby.js application. But everything else is bare bones

Recommendations:

- create-react-app
- Gatsby.js for static websites in React
- Next.js for server-side rendered React

UTILITY LIBRARIES FOR REACT

JavaScript ES6 and beyond gives you plenty of built-in functionalities dealing with arrays, objects, numbers, objects and strings. One of the most used JavaScript built-in functionalities in React is the built-in map() Array. Why? Because you always have to render a list of items in a component. Since JSX is a mixture of HTML and JavaScript, you can use JavaScript to map over your items and return JSX.

```
const List = ({ list }) =>
  <div>
    {list.map(item => <div key={item.id}>{item.title}</div>)}
  </div>
```

However, you might come to a point where you have to choose a utility library that gives you more elaborate functionalities. You might even want to be more flexible when chaining these utility functions or even compose them dynamically into each other. That's the point in time where you would introduce a utility library. My personal recommendations are two libraries.

The first recommendation is Lodash. It is the most widespread utility library in JavaScript. I guess there are people who know more about Lodash than about the native JavaScript functionalities, because people often learn libraries before learning a programming language, but also because JavaScript introduced new functionalities in its recent versions. Nevertheless, Lodash comes with a powerful set of functions to access, manipulate and compose.

The second recommendation is Ramda. When you lean towards functional programming (FP) in JavaScript, there is no way around this utility library. Even though Lodash comes with its own functional programming derivate (Lodash FP), I would always recommend using Ramda when dealing with FP in JavaScript. It gives you a powerful set of functionalities to be productive.

So when introducing a utility library to your React core, you could make the decision between Lodash and Ramda. Whereas Lodash is the more down to earth library for every JavaScript developer, Ramda comes with a powerful core when functional programming comes into play.

Recommendations:

- JavaScript
- Lodash
- Ramda for FP

STYLING IN REACT

When it comes to styling in React, it becomes opinionated in the React ecosystem. Not only regarding the specific solutions that are already out there, but because of the overarching philosophies. For instance, is it okay to have inline style in JSX? Is it fine to colocate style with components?

When starting with React, it is just fine to use plain CSS. If your first project is setup with create-react-app, you will encounter only CSS and can decide to add inline style too.

```
const Headline = ({ children }) =>
  <h1 className="headline" style={{ color: 'lightblue' }}>
    {children}
```

```
    </h1>
```

In smaller applications, it can be just fine to go only with plain CSS and inline style. Once your application scales, I would advice you to have a look into CSS modules. It gives you a way to encapsulate your CSS so that it doesn't leak to other parts of the application. Parts of your application can still share style while other parts don't have to get access to it. CSS modules scale well in growing applications. In React these modules are most often colocated files to your React component files.

A different approach of styling a component in React is defining a Styled Component. This approach is brought to you by a library called styled-components. It colocates styling in your JavaScript to your React components and doesn't attempt to share the styling with other components. It only styles a specific component.

Last but not least, there is one neat helper library for styling in React: classnames. It enables you introducing conditional styling. In plain JavaScript, it would be possible to create a React class attribute with conditionals:

```
const Box = ({ status, children }) => {
  let classNames = ['box'];

  if (status === 'INFO') {
    classNames.push('box-info');
  }

  if (status === 'WARNING') {
    classNames.push('box-warning');
  }

  if (status === 'ERROR') {
    classNames.push('box-error');
  }

  return (
    <div className={classNames.join(' ')}>
      {children}
    </div>
  );
}
```

But it is so much easier with the classnames library:

```
import cs from 'classnames';

const Box = ({ status, children }) => {
  let classNames = cs('box', {
    'box-info': status === 'INFO',
    'box-warning': status === 'WARNING',
    'box-error': status === 'ERROR',
```

```
  });

  return (
    <div className={classNames}>
      {children}
    </div>
  );
}
```

It works perfectly with CSS modules too.

```
import cs from 'classnames';
import styles from './style.css';

const Box = ({ status, children }) => {
  let classNames = cs('box', {
    [styles.box_info]: status === 'INFO',
    [styles.box_warning]: status === 'WARNING',
    [styles.box_error]: status === 'ERROR',
  });

  return (
    <div className={classNames}>
      {children}
    </div>
  );
}
```



The library is for many people almost mandatory in applications when it comes to conditional stylings in React.

Recommendations:

- plain CSS and inline style
- almost mandatory classnames library
- CSS modules or Styled Components

ASYNCHRONOUS REQUESTS IN REACT

Beyond a Todo application in React, you will pretty soon have to make a request to a third party API. In the past, you would have often used jQuery for this kind of job.

Nowadays, recent browsers implement the native fetch API to conduct asynchronous requests. It uses promises under the hood. Basically a fetch looks like the following, for instance in a React lifecycle method when a component mounts:

```
componentDidMount() {
  fetch(my/api/domain)
    .then(response => response.json())
    .then(result => {
      // do success handling
      // e.g. store in local state
    });
}
```

Basically you wouldn't have to add any other library to do the job. However, there exist libraries which only purpose it is to provide sophisticated asynchronous requests. They come with more powerful functionalities yet are only a lightweight library. One of these libraries that I would recommend is called axios. It can be used instead of the native fetch API when your application grows in size. Another alternative is called superagent.

Recommendations:

- native fetch API
- axios

REACT'S HIGHER ORDER COMPONENTS

Eventually you get to the point where you want to abstract away functionalities for your components. These opt-in functionalities can be shared across components yet leave the components themselves lightweight. That's when React's higher order components come into play. These kind of components don't need any additional library in React.

However, there are common use cases for React's higher order components that are already solved in a library called recompose. For instance, having a higher order component for conditional rendering (branch). When you introduce higher order components to your React application, make sure that the use case is not already covered in recompose.

Another neat helper in the recompose library is the `compose()` function. It allows you to opt-in multiple higher order components in an elegant way. However, you could use a utility library such as Lodash or Ramda for the compose function too.

Recommendations:

- recompose for utility higher order components
- recompose or utility library (Lodash, Ramda) for compose

REACT TYPE CHECKING

Fortunately React comes with its own type checking abilities. With PropTypes you are able to define the incoming props for your React components.

```
import PropTypes from 'prop-types';

const List = ({ list }) =>
  <div>
    {list.map(item => <div key={item.id}>{item.title}</div>)}
  </div>

List.propTypes = {
  list: PropTypes.array.isRequired,
};
```

Whenever a wrong type is passed to the component, you will get an error message when running the application. But this form of type checking should only be used for smaller applications. Facebook recommends to use Flow instead.

In a larger React application, you can add sophisticated type checker such as Flow and TypeScript instead of React PropTypes. When using such a type checker, you can get errors already during development time. You wouldn't have to start your application in order to find about a bug that could have prevented with such type checking. That way a type checker might be able to improve your developer experience and avoids to introduce bugs in the first place.

Flow was introduced by Facebook and feels more natural in the React ecosystem than TypeScript. That's why I would recommend using it in a React application over TypeScript. But there are many people using TypeScript as well. In the end, both solutions should solve the same problem for you.

Recommendations:

- React's PropTypes
- Flow (or TypeScript)

REACT FORMATTING

Basically there are three options to have formatting rules in React. It should be quite similar to other ecosystems.

The first approach is to follow a style guide that is embraced by the community. One popular React style guide was open sourced by Airbnb. Even though you don't deliberately follow the style guide, it makes sense to read it once to get the basics of formatting in React.

The second approach is to use a linter such as ESLint. You can integrate it in your tool set when you are at the point of introducing new toolings to your project yourself.

The third and most popular approach is using Prettier. It is an opinionated code formatter. You can integrate it in your editor or IDE that it formats your code every time you save a file or commit it with git. Perhaps it doesn't match always your taste, but at least you never need to worry again about code formatting in your own or a team code base.

Recommendations:

- reading one popular React style guide
- Prettier



REACT UI COMPONENT LIBRARIES

There are many components in web development which you don't want to implement from scratch every time you start to implement a new application. These are things like datepickers, dropdowns, tables or navigation bars. That's why there are a couple of UI libraries out there which are working closely with libraries such as React and Angular. I don't want to introduce all of them to you, but after doing my own research, I found out that Semantic UI and Material UI are great choices. I use them for many of my own applications nowadays. Once you feel comfortable with one, you don't want to switch horses all the time. After all, they should only deliver the foundational components in web development for you.

Recommendations:

- Semantic UI

REACT STATE MANAGEMENT

Fortunately React comes with its own local state management in components. This is why it is just fine to learn plain React first. You will only master the fundamentals in React when using `this.state` and `this.setState()` for local state management.

Often beginners make the mistake to learn React altogether with Redux. So don't bother too early with a state management library when you are just starting to use React.

But what comes when you run into first scaling issues in React's local state management? There are two solutions you can choose from: Redux and MobX. Both come with their advantages and disadvantages. You can read the linked article to make a more informed decision about them.

Redux is so popular yet such an innovative place that it comes with its own ecosystem. When using it with React, you will certainly run into the bridging library react-redux. It is the official library to connect your view layer (React) to your state layer (Redux). A similar library comes into play when you decide to use MobX instead of Redux. If you want to learn Redux, checkout the Redux book Taming the State in React.

As mentioned, Redux comes with its own ecosystem. The next recommendations are far beyond a simple setup for a React application. But when you scaled your application to a certain point, where Redux becomes an inherent part for your application and you are confident in using Redux, I can recommend to have a look into these libraries: Redux Saga, Normalizr and Reselect.

Recommendations:

- React's local state
- Redux or MobX
  - only if you are doing great with React's local state
  - only if it's needed

REACT'S ROUTING

Routing is often introduced in an early stage in React applications. After all, React helps you implementing a view-layer that is most often used in a single page application. Thus routing is a crucial part of the application.

But before you introduce a heavy router in your application, when you are just about to learn React, you can give React's conditional rendering a shot first. It is not a valid replacement for routing, but in small applications it is often sufficient to exchange components that way. It doesn't change the URL though, but still you would be able to map different states to your view.

When introducing a sophisticated router, there are a few routing solutions out there for React. But the most anticipated solution is React Router. It works well along with an external state management library such as Redux or MobX too.

Recommendations:

- React's conditional rendering
- React Router

REACT AUTHENTICATION

In a major application, you may want to introduce an authentication flow in React by having sign up, sign in and sign out functionalities. Often these authentication mechanisms go far beyond it. After all, you should be able to reset or update a user's password or to store the user entities in your own database. That's why I have written the comprehensive tutorial and step by step guide to built your own authentication procedure in React with Firebase. Firebase's realtime database can be used to store your users. But it also comes with a whole authentication framework. Check out the tutorial if you are interested.

There are two alternatives in my eyes for authentication in React which are briefly mentioned in the Firebase tutorial: Auth0 and Passport.js.

Recommendations:

- Firebase

TESTING IN REACT

Writing tests is an essential part of software development to ensure a robust application. Tests enable us to automatically verify that our application is working on a certain level. The certain level depends on the quality, quantity (coverage) and type of your tests (unit tests, integration tests, end-to-end tests). So here comes my recommendation on the testing frameworks you could use for your React application.

Mocha and Chai are a pair of two testing frameworks which are used quite often together. Whereas Mocha is your test runner which makes it possible that your tests run via npm scripts and which encapsulates your tests in test suites, Chai is used for assertion such as `to.equal()`. It's the perfect combination for React applications to start out with simple testing use case such as pure functions. If you need to test asynchronous business logic eventually, you should definitely checkout Sinon.

Once you want to test your components, you can add Enzyme to your set of testing libraries for rendering actual React components. It's a library by Airbnb which makes it

possible to render your components, simulate events on HTML elements and make expectations on your props, state and DOM nodes. An alternative to Enzyme is React Testing Library which follows a different testing philosohpy than Enzyme.

Last but not least, there is also Jest. It's the official tetsing library by Facebook. Jest introduced so called snapshot tests for React components. Once you run your tests, a snapshot of your rendered DOM elements of the React component is created. When you run your tests again at some point, another snapshot is created which is used as diff for the previous snapshot. If the diff is not identical, Jest will complain and you either have to accept the snapshot or change the implementation of your component.

Another alternative for testing your React components is Ava. There are many developers who like it as alternative to all the other combined testing libraries. It has no snapshot tests, but if you don't depend on those, you can checkout Ava.

If the whole combo of Mocha (test runner), Chai (assertion), Enzyme/React Testing Library (component tests) and Jest (snapshot tests, but also test runner and assertion) is too much for you, you can only use Jest with Enzyme/React Testing Library. This way, you keep your React testing setup more lightweight. Check out this extensive React testing tutorial series on how to setup various testing frameworks for your React application. It makes totally sense to dive into this topic once before starting out with a larger application. After all, you want to implement a robust and maintainable application.

Recommendations:

- Unit/Integration/Snapshot Tests: Jest + Enzyme/React Testing Library
- E2E Tests: Cypress

TIME IN REACT

If your React application is dealing with dates and timezones, you should introduce a library which manages these things for you. Timestamps in an application can be a complex subject. That's why it's great that there are sophisticated libraries out for it. The most popular one is moment.js because I would argue it's the oldest and most sophisticated one in the JavaScript ecosystem. Another lightweight solution for dates is a library called date-fns.

Recommendations:

- date-fns

. . .

So in the end, the React ecosystem can be seen as a framework for React, but it stays flexible. It is a flexible framework where you can make own decisions on which libraries you want to opt-in. You can start small and add only libraries to solve specific problems for you. You can scale your building blocks along the way when your application grows. Otherwise you can stay lightweight by using plain React. Therefore here again a list of libraries that could complement React as the core of the application regarding different project sizes. Keep in mind that the list is opinionated, but I am keen to get your feedback too.

- Small Application
- **Boilerplate:** create-react-app
- **Utility:** JavaScript ES6 and beyond
- **Styling:** plain CSS and inline style
- **Asynchronous Requests:** fetch
- **Higher Order Components:** optional
- **Formatting:** none
- **Type Checking:** none or PropTypes
- **State Management:** local state
- **Routing:** none or conditional rendering
- **Authentication:** Firebase
- **Database:** Firebase
- **UI Components:** none
- **Time:** date-fns
- **Testing:** Jest
- Medium Application
- **Boilerplate:** create-react-app with eject
- **Utility:** JavaScript ES6 + Lodash or Ramda
- **Styling:** CSS modules or Styled Components
- **Asynchronous Requests:** fetch or axios
- **Higher Order Components:** maybe + optional recompose
- **Formatting:** Prettier
- **Type Checking:** none or Flow
- **State Management:** local state and very optional Redux
- **Routing:** React Router
- **Authentication:** Firebase
- **Database:** Firebase
- **UI Components:** none or Semantic UI
- **Time:** date-fns
- **Testing:** Jest with Enzyme/React Testing Library
- Large Application
- **Boilerplate:** create-react-app with eject or own boilerplate project
- **Utility:** JavaScript ES6 + Lodash or Ramda

- **Styling:** CSS modules or Styled Components
- **Asynchronous Requests:** axios
- **Higher Order Components:** maybe + optional recompose
- **Formatting:** Prettier
- **Type Checking:** Flow
- **State Management:** local state and Redux or MobX
- **Routing:** React Router
- **Authentication:** Solution with an own Express/Hapi/Koa Node.js Server with Passport.js
- **Database:** Solution with an own Express/Hapi/Koa Node.js Server with a SQL or NoSQL Database
- **UI Components:** Semantic UI or own implementation of UI components
- **Time:** date-fns
- **Testing:** Jest with Enzyme/React Testing Library and Cypress

The previous recommendations are opinionated. You can choose your own flexible framework for your ideal React application. Every "ideal" React setup is subjective to its needs of the developers and project. After all, there is no ideal React application setup.

KEEP READING RELATED ARTICLES›

REACT HOOKS: MIGRATION FROM CLASS TO FUNCTION COMPONENTS

React Hooks were introduced to React to make state and side-effects available in React Function Components. Before it was only possible to have these in React Class Components; but since React's way…

REACT COMPONENT TYPES: A COMPLETE OVERVIEW

Even though React didn't introduce a lot of breaking changes since it has been released 2013, different React Component Types emerged over time. A few of these component types and component patterns…