React's Boilerplate Decision

Even nowadays developers 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](https://github.com/facebookincubator/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](https://www.robinwieruch.de/minimal-react-webpack-babel-setup/). 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](https://github.com/zeit/next.js/). It is a zero-configuration React boilerplate as well, but for server-side rendered React. Another alternative is called [Gatsby.js](https://github.com/gatsbyjs/gatsby) 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](https://github.com/rwieruch/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
* create your own minimal boilerplate, when having a solid understanding of React
* otherwise narrow down your requirements and choose a minimal boilerplate for it

Utility Libraries for React

JavaScript ES6 and beyond gives you tons of built-in functionalities dealing with arrays, objects, numbers, objects, strings etc. One of the most used JavaScript built-in functionalities in React is the [map() method of arrays](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/map). 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 the point to choose a utility library that gives you more elaborated 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](https://lodash.com/). 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](http://ramdajs.com/). 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](https://github.com/lodash/lodash/wiki/FP-Guide)), 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 ES6 and beyond
* Lodash
* Ramda for functional programming

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](https://github.com/css-modules/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](https://github.com/styled-components/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](https://github.com/JedWatson/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](https://www.robinwieruch.de/what-is-an-api-javascript/). In the past, you would have often used jQuery for this kind of job.

Nowadays, recent browsers implement the [native fetch API](https://developer.mozilla.org/en/docs/Web/API/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](https://github.com/mzabriskie/axios). It can be used instead of the native fetch API when your application grows in size. Another alternative is called [superagent](https://github.com/visionmedia/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](https://www.robinwieruch.de/gentle-introduction-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](https://github.com/acdlite/recompose). For instance, having a higher order component for conditional rendering ([branch](https://github.com/acdlite/recompose/blob/master/docs/API.md#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](https://facebook.github.io/react/docs/typechecking-with-proptypes.html) 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](https://flow.org/) and [TypeScript](https://www.typescriptlang.org/) 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](https://github.com/airbnb/javascript/tree/master/react) 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](https://www.robinwieruch.de/react-eslint-webpack-babel/) when you are at the point of introducing new toolings to your project yourself.

The third and most popular approach is using [Prettier](https://github.com/prettier/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](https://www.robinwieruch.de/git-essential-commands/). 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](https://react.semantic-ui.com/) and [Material UI](http://www.material-ui.com/) are great choices. I use them for many of my [own applications](https://www.robinwieruch.de/how-to-build-your-own-course-platform/) 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](https://www.robinwieruch.de/learn-react-before-using-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](https://www.robinwieruch.de/redux-mobx-confusion/). 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](https://github.com/markerikson/redux-ecosystem-links). When using it with React, you will certainly run into the bridging library [react-redux](https://github.com/reactjs/react-redux). It is the official library to connect your view layer (React) to your state layer (Redux). A [similar library](https://github.com/mobxjs/mobx-react) 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](https://roadtoreact.com/).

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](https://github.com/redux-saga/redux-saga), [Normalizr](https://github.com/paularmstrong/normalizr) and [Reselect](https://github.com/reactjs/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](https://www.robinwieruch.de/conditional-rendering-react/) 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](https://github.com/ReactTraining/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](https://www.robinwieruch.de/complete-firebase-authentication-react-tutorial/). 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](https://auth0.com/) and [Passport.js](http://www.passportjs.org/).

**Recommendations:**

* Firebase

Testing in React

Testing in React is a too rarely touched topic. 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](https://github.com/mochajs/mocha) and [Chai](https://github.com/chaijs/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](https://github.com/sinonjs/sinon).

Once you want to test your components, you can add [Enzyme](https://github.com/airbnb/enzyme) to your set of testing frameworks for unit and integration tests. 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.

Last but not least, you can add [Jest](https://github.com/facebook/jest) to the mix. It’s the official library by Facebook. Jest introduced so called snapshot tests for 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.

If the whole combo of Mocha (test runner), Chai (assertion), Enzyme (component tests) and Jest (snapshot tests) is too much for you, you can only use Jest with Enzyme. Jest comes with its own assertion API (which was provided by Chai) and own test runner (which was provided by Mocha). That way you can keep your testing environment lightweight.

Another alternative for testing your React components is [Ava](https://github.com/avajs/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.

In conclusion, checkout [this extensive React testing tutorial](https://www.robinwieruch.de/react-testing-tutorial) on how to setup various testing frameworks for your React application. You can either setup Mocha, Chai, Enzyme, Jest (and Sinon) or only Jest with Enzyme (and Sinon). 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:**

* Mocha (Test Runner), Chai (Assertion Library) for Business Logic and Sinon for async Business Logic
* Mocha, Chai, Enzyme, Jest (and Sinon)
* Jest (Test Runner, Assertion Library, Snapshot Tests) without Mocha and Chai but with Enzyme (and Sinon)

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](https://momentjs.com/) 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](https://github.com/date-fns/date-fns).

**Recommendations:**

* moment or 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:** moment or 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:** moment or date-fns
  + **Testing:** Jest with Enzyme
* 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:** moment or date-fns
  + **Testing:** Jest with Enzyme