BULLETPROOF-REACT Docs - English

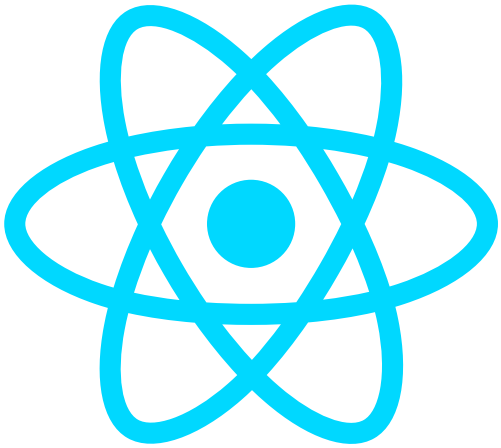

Table of contents

Additional Resources

React

- Official Documentation
- Beta Official Documentation
- The Beginner's Guide to React
- patterns.dev
- React Bits
- React Patterns
- Tao Of React

JavaScript

- You Dont Know JS
- JavaScript Info
- 33 Concepts Every JavaScript Developer Should Know
- JavaScript to Know for React

Go to TOC

📡 API Layer

Use a single instance of the API client

No matter if your application is consuming RESTful or GraphQL API, have a single instance of the API client that's been pre-configured and reused throughout the application. E.g have a single API client (axios / graphql-request / apollo-client) instance with pre-defined configuration.

API Client Example Code

Define and export request declarations

Instead of declaring API requests on the go, have them defined and exported separately. If it's a RESTful API a declaration would be a fetcher function that calls an endpoint. On the other hand, requests for GraphQL APIs are declared via queries and mutations that could be consumed by data fetching libraries such as react-query, apollo-client, urql, etc. This makes it easier to track which endpoints are defined and available in the application. You can also type the responses and infer them further for a good type safety of the data. You can also define and export corresponding API hooks from there.

API Request Declaration Example Code

Go to TOC

💻 Application Overview

The application is pretty simple. Users can create teams where other users can join, and they start discussions on different topics between each other.

A team is created during the registration if the user didn't choose to join an existing team and the user becomes the admin of it.

Demo

Data model

The application contains the following models:

- User - can have one of these roles:
  - `ADMIN` can:
    - create/edit/delete discussions
    - create/delete all comments
    - delete users
    - edit own profile
  - `USER` - can:
    - edit own profile
    - create/delete own comments
- Team: represents a team that has 1 admin and many users that can participate in discussions between each other.
- Discussion: represents discussions created by team members.
- Comment: represents all the messages in a discussion.

Get Started

Prerequisites:

- Node 14+
- Yarn 1.22+

To set up the app execute the following commands.

```
git clone https://github.com/alan2207/bulletproof-react.git
cd bulletproof-react
cp .env.example .env
yarn install
```

`yarn start`

Runs the app in the development mode.
Open http://localhost:3000 to view it in the browser.

`yarn build`

Builds the app for production to the `build` folder.
It correctly bundles React in production mode and optimizes the build for the best performance.

The build is minified and the filenames include the hashes.
Your app is ready to be deployed!

See the section about deployment for more information.

Go to TOC

Components And Styling

Components Best Practices

Colocate things as close as possible to where it's being used

Keep components, functions, styles, state, etc. as close as possible to the component where it's being used. This will not only make your codebase more readable and easier to understand but it will also improve your application performance since it will reduce redundant re-renders on state updates.

Avoid large components with nested rendering functions

Do not add multiple rendering functions inside your application, this gets out of control pretty quickly. What you should do instead is if there is a piece of UI that can be considered as a unit, is to extract it in a separate component.

```
// this is very difficult to maintain as soon as the component starts growing
function Component() {
  function renderItems() {
    return <ul>...</ul>;
  }
  return <div>{renderItems()}</div>;
}

// extract it in a separate component
function Items() {
  return <ul>...</ul>;
}

function Component() {
  return (
    <div>
      <Items />
    </div>
  );
}
```

Stay consistent

Keep your code style consistent. e.g If you name your components by using pascal case, do it everywhere. More on this can be found here

Limit the number of props a component is accepting as input

If your component is accepting too many props you might consider splitting it into multiple components or use the composition technique via children or slots.

Composition Example Code

Abstract shared components into a component library

For larger projects, it is a good idea to build abstractions around all the shared components. It makes the application more consistent and easier to maintain. Identify repetitions before creating the components to avoid wrong abstractions.

Component Library Example Code

It is a good idea to wrap 3rd party components as well in order to adapt them to the application's needs. It might be easier to make the underlying changes in the future without affecting the application's functionality.

3rd Party Component Example Code

Component libraries

Every project requires some UI components such as modals, tabs, sidebars, menus, etc. Instead of building those from scratch, you might want to use some of the existing, battle-tested component libraries.

Fully featured component libraries:

These component libraries come with their components fully styled.

- Chakra UI - great library with probably the best developer experience, allows very fast prototyping with decent design defaults. Plenty of components that are very customizable and flexible with accessibility already configured out of the box.
- AntD - another great component library that has a lot of different components. Best suitable for creating admin dashboards. However, it might be a bit difficult to change the styles in order to adapt them to a custom design.
- MUI - the most popular component library for React. Has a lot of different components. Can be used as a styled solution by implementing Material Design or as unstyled headless component library.

Headless component libraries:

These component libraries come with their components unstyled. If you have a specific design system to implement, it might be easier and better solution to go with headless components that come unstyled than to adapt a styled components library such as Material UI to your needs. Some good options are:

- Reakit
- Headless UI
- Radix UI
- react-aria

Styling Solutions

There are multiple ways to style a react application. Some good options are:

- tailwind
- styled-components
- emotion
- stitches
- vanilla-extract
- CSS modules
- linaria

Good combinations

Some good combinations of component library + styling

- Chakra UI + emotion - The best choice for most applications
- Headless UI + tailwind
- Radix UI + stitches

Storybook

Storybook is a great tool for developing and testing components in isolation. Think of it as a catalogue of all the components your application is using. Very useful for developing and discoverability of components.

Storybook Story Example Code

Go to TOC

🌐 Deployment

Deploy and serve your applications and assets over a CDN for best delivery and performance. Good options for that are:

- Vercel
- Netlify
- AWS
- CloudFlare

Go to TOC

⚠ Error Handling

API Errors

Set up an interceptor for handling errors. You can use it to fire a notification toast to notify users that something went wrong, log out unauthorized users, or send new requests for refreshing tokens.

API Errors Notification Example Code

In App Errors

Use error boundaries to handle errors that happen in the React tree. It is very popular to set only 1 single error boundary for the entire application, which would break the entire application when an error occurs. That's why you should have more error boundaries on more specific parts of the application. That way if an error occurs the app will still work without the need to restart it.

Error Boundary Example Code

Error Tracking

You should track any errors that occur in production. Although it's possible to implement your own solution, it is a better idea to use tools like Sentry. It will report any issue that breaks the app. You will also be able to see on which platform, browser, etc. did it occur. Make sure to upload source maps to sentry to see where in your source code did the error happen.

Go to TOC

🚄 Performance

Code Splitting

Code splitting is a technique of splitting production JavaScript into smaller files, thus allowing the application to be only partially downloaded. Any unused code will not be downloaded until it is required by the application.

Most of the time code splitting should be done on the routes level, but can also be used for other lazy loaded parts of application.

Do not code split everything as it might even worsen your application's performance.

Code Splitting Example Code

Component and state optimizations

- Do not put everything in a single state. That might trigger unnecessary re-renders. Instead split the global state into multiple stores according to where it is being used.
- Keep the state as close as possible to where it is being used. This will prevent re-rendering components that do not depend on the updated state.
- If you have a piece of state that is initialized by an expensive computation, use the state initializer function instead of executing it directly because the expensive function will be run only once as it is supposed to. e.g:

```
// instead of this which would be executed on every re-render:
const [state, setState] = React.useState(myExpensiveFn());

// prefer this which is executed only once:
const [state, setState] = React.useState(() => myExpensiveFn());
```

- If you develop an application that requires the state to track many elements at once, you might consider state management libraries with atomic updates such as recoil or jotai.
- If your application is expected to have frequent updates that might affect performance, consider switching from runtime styling solutions (Chakra UI, emotion, styled-components that generate styles during runtime) to zero runtime styling solutions (tailwind, linaria, vanilla-extract, CSS modules which generate styles during build time).

Image optimizations

Consider lazy loading images that are not in the viewport.

Use modern image formats such as WEBP for faster image loading.

Use `srcset` to load the most optimal image for the clients screen size.

Web vitals

Since Google started taking web vitals in account when indexing websites, you should keep an eye on web vitals scores from Lighthouse and Pagespeed Insights.

Go to TOC

⚙️ Project Configuration

The application has been bootstrapped using `Create React App` for simplicity reasons. It allows us to create applications quickly without dealing with a complex tooling setup such as bundling, transpiling etc.

You should always configure and use the following tools:

ESLint

ESLint is a linting tool for JavaScript. By providing specific configuration defined in the `.eslintrc.js` file it prevents developers from making silly mistakes in their code and enforces consistency in the codebase.

ESLint Configuration Example Code

Prettier

This is a great tool for formatting code. It enforces a consistent code style across your entire codebase. By utilizing the "format on save" feature in your IDE you can automatically format the code based on the configuration provided in the `.prettierrc` file. It will also give you good feedback when something is wrong with the code. If the auto-format doesn't work, something is wrong with the code.

Prettier Configuration Example Code

TypeScript

ESLint is great for catching some of the bugs related to the language, but since JavaScript is a dynamic language ESLint cannot check data that run through the applications, which can lead to bugs, especially on larger projects. That is why TypeScript should be used. It is very useful during large refactors because it reports any issues you might miss otherwise. When refactoring, change the type declaration first, then fix all the TypeScript errors throughout the project and you are done. One thing you should keep in mind is that TypeScript does not protect your application from failing during runtime, it only does type checking during build time, but it increases development confidence drastically anyways. Here is a great resource on using TypeScript with React.

Husky

Husky is a tool for executing git hooks. Use Husky to run your code validations before every commit, thus making sure the code is in the best shape possible at any point of time and no faulty commits get into the repo. It can run linting, code formatting and type checking, etc. before it allows pushing the code. You can check how to configure it here.

Absolute imports

Absolute imports should always be configured and used because it makes it easier to move files around and avoid messy import paths such as `../../../Component`. Wherever you move the file, all the imports will remain intact. Here is how to configure it:

For JavaScript (`jsconfig.json`) projects:

```
"compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@/*": [https://github.com/alan2207/bulletproof-react/tree/master/src/*"]
    }
  }
```

For TypeScript (`tsconfig.json`) projects:

```
"compilerOptions": {
    "baseUrl": ".",
    "paths": {
      "@/*": [https://github.com/alan2207/bulletproof-react/tree/master/src/*"]
    }
  }
```

Paths Configuration Example Code

In this project we have to create another tsconfig file `tsconfig.paths.json` where we configure the paths and merge it with the base configuration, because CRA will override it otherwise.

It is also possible to define multiple paths for various folders(such as `@components` , `@hooks` , etc.), but using `@/*` works very well because it is short enough so there is no need to configure multiple paths and it differs from other dependency modules so there is no confusion in what comes from `node_modules` and what is our source folder. That means that anything in the `src` folder can be accessed via `@` , e.g some file that lives in `src/components/MyComponent` can be accessed using `@/components/MyComponents` .

Go to TOC

🗄 Project Structure

Most of the code lives in the `src` folder and looks like this:

```
src
|
+-- assets            # assets folder can contain all the static files such as
images, fonts, etc.
|
+-- components        # shared components used across the entire application
|
+-- config            # all the global configuration, env variables etc. get
exported from here and used in the app
|
+-- features          # feature based modules
|
+-- hooks             # shared hooks used across the entire application
|
+-- lib               # re-exporting different libraries preconfigured for the
application
|
+-- providers         # all of the application providers
|
+-- routes            # routes configuration
|
+-- stores            # global state stores
|
+-- test              # test utilities and mock server
|
+-- types             # base types used across the application
|
+-- utils             # shared utility functions
```

In order to scale the application in the easiest and most maintainable way, keep most of the code inside the `features` folder, which should contain different feature-based things. Every `feature` folder should contain domain specific code for a given feature. This will allow you to keep functionalities scoped to a feature and not mix its declarations with shared things. This is much easier to maintain than a flat folder structure with many files.

A feature could have the following structure:

```
src/features/awesome-feature
|
+-- api         # exported API request declarations and api hooks related to a
specific feature
|
+-- assets      # assets folder can contain all the static files for a specific
feature
|
+-- components  # components scoped to a specific feature
|
+-- hooks       # hooks scoped to a specific feature
|
+-- routes      # route components for a specific feature pages
|
+-- stores      # state stores for a specific feature
```

```
|
+-- types       # typescript types for TS specific feature domain
|
+-- utils       # utility functions for a specific feature
|
+-- index.ts    # entry point for the feature, it should serve as the public API
of the given feature and exports everything that should be used outside the
feature
```

Everything from a feature should be exported from the `index.ts` file which behaves as the public API of the feature.

You should import stuff from other features only by using:

```
import {AwesomeComponent} from "@/features/awesome-feature"
```

and not

```
import {AwesomeComponent} from "@/features/awesome-feature/components/AwesomeComponent
```

This can also be configured in the ESLint configuration to disallow the later import by the following rule:

```
{
    rules: {
        'no-restricted-imports': [
            'error',
            {
                patterns: ['@/features/*/*'],
            },
        ],

    // ...rest of the configuration
}
```

This was inspired by how NX handles libraries that are isolated but available to be used by the other modules. Think of a feature as a library or a module that is self-contained but can expose different parts to other features via its entry point.

Go to TOC

🔐 Security

Auth

NOTE: Handling Auth on the client doesn't mean it shouldn't be handled on the server. As the matter of fact, it is more important to protect the resources on the server, but it should be handled on the client as well for better user experience.

There are 2 parts of Auth:

Authentication

Authentication is a process of identifying who the user is. The most common way of authenticating users in single page applications is via JWT. During logging in / registration you receive a token that you store in your application, and then on each authenticated request you send the token in the header or via cookie along with the request.

The safest option is to store the token in the app state, but if the user refreshes the app, its token will be lost.

That is why tokens are stored in `localStorage/sessionStorage` or in a cookie.

`localStorage` vs cookie for storing tokens

Storing it in `localStorage` could bring a security issue, if your application is vulnerable to XSS someone could steal your token.

Storing tokens in a cookie might be safer if the cookie is set to be `HttpOnly` which would mean it wouldn't be accessible from the client side JavaScript. The `localStorage` way is being used here for simplicity reasons, if you want to be more secure, you should consider using cookies but that is a decision that should be made together with the backend team.

To keep the application safe, instead of focusing only on where to store the token safely, it would be recommended to make the entire application as resistant as possible to XSS attacks E.g - every input from the user should be sanitized before it's injected into the DOM.

HTML Sanitization Example Code

Handling user data

User info should be considered a global piece of state which should be available from anywhere in the application. If you are already using `react-query`, you can use react-query-auth library for handling user state which will handle all the things for you after you provide it some configuration. Otherwise, you can use react context + hooks, or some 3rd party state management library.

Auth Configuration Example Code

The application will assume the user is authenticated if a user object is present.

Authenticated Route Protection Example Code

Authorization

Authorization is a process of determining if the user is allowed to access a resource.

RBAC (Role based access control)

Authorization Configuration Example Code

The most common method. Define allowed roles for a resource and then check if a user has the allowed role in order to access a resource. Good example is `USER` and `ADMIN` roles. You want to restrict some things for users and let admins access it.

RBAC Example Code

PBAC (Permission based access control)

Sometimes RBAC is not enough. Some of the operations should be allowed only by the owner of the resource. For example user's comment - only the author of the comment should be able to delete it. That's why you might want to use PBAC, as it is more flexible.

For RBAC protection you can use the `RBAC` component by passing allowed roles to it. On the other hand if you need more strict protection, you can pass policies check to it.

PBAC Example Code

Go to TOC

State Management

There is no need to keep all of your state in a single centralized store. There are different needs for different types of state that can be split into several types:

Component State

This is the state that only a component needs, and it is not meant to be shared anywhere else. But you can pass it as prop to children components if needed. Most of the time, you want to start from here and lift the state up if needed elsewhere. For this type of state, you will usually need:

- useState - for simpler states that are independent
- useReducer - for more complex states where on a single action you want to update several pieces of state

Component State Example Code

Application State

This is the state that controls interactive parts of an application. Opening modals, notifications, changing color mode, etc. For best performance and maintainability, keep the state as close as possible to the components that are using it. Don't make everything global out of the box.

Good Application State Solutions:

- context + hooks
- redux + redux toolkit
- mobx
- zustand
- constate
- jotai
- recoil
- xstate

UI State Example Code

Server Cache State

This is the state that comes from the server which is being cached on the client for further usage. It is possible to store remote data inside a state management store such as redux, but there are better solutions for that.

Good Server Cache Libraries:

- react-query - REST + GraphQL
- swr - REST + GraphQL

- apollo client - GraphQL
- urql - GraphQl

Server State Example Code

Form State

This is a state that tracks users inputs in a form.

Forms in React can be controlled and uncontrolled.

Depending on the application needs, they might be pretty complex with many different fields which require validation.

Although it is possible to build any form using only React, there are pretty good solutions out there that help with handling forms such as:

- React Hook Form
- Formik
- React Final Form

Create abstracted `Form` component and all the input field components that wrap the library functionality and are adapted to the application needs. You can reuse it then throughout the application.

Form Example Code

Input Field Example Code

You can also integrate validation libraries with the mentioned solutions to validate inputs on the client. Some good options are:

- zod
- yup

Validation Example Code

URL State

State that is being kept in the address bar of the browser. It is usually tracked via url params (`/app/${dynamicParam}`) or query params (`/app?dynamicParam=1`). It can be accessed and controlled via your routing solution such as `react-router-dom` .

URL State Example Code

Go to TOC

👁 Style Guide

When you work with large projects, it's important that you remain consistent throughout the codebase and follow the best practices. To guarantee the quality of your codebase, you need to analyze different levels of the applications code.

Clean Code

This is the most abstract level of code standardization. It's related to the implementations independent of the programming language. It will help the readability of your code.

Clean Code Javascript

Naming

One of the most important points of the Clean Code is how you name your functions, variables, components, etc. Use this amazing guide to understand how to write better variable names.

Naming Cheatsheet

Go to TOC

🧪 Testing

This tweet explains in a concise way how to think about testing. You will get the most benefit from having integration and e2e tests. Unit tests are fine, but they will not give you as much confidence that your application is working as integration tests do.

Types of tests:

Unit Tests

Unit testing, as the naming already reveals is a type of testing where units of an application are being tested in isolation. You should write unit tests for shared components and functions that are used throughout the entire application as they might be used in different scenarios which might be difficult to reproduce in integration tests.

Unit Test Example Code

Integration Tests

Integration testing is a method of testing multiple parts of an application at once. Most of your tests should be integration tests, as these will give you the most benefits and confidence for your invested effort. Unit tests on their own don't guarantee that your app will work even if those tests pass, because the relationship between the units might be wrong. You should test different features with integration tests.

Integration Test Example Code

E2E

End-To-End Testing is a testing method where an application is tested as a complete entity. Usually these tests consist of running the entire application with the frontend and the backend in an automated way and verifying that the entire system works. It is usually written in the way the application should be used by the user.

E2E Example Code

Tooling:

Jest

Jest is a fully featured testing framework and is de-facto standard when it comes to testing JavaScript applications. It is very flexible and configurable to test both frontends and backends.

Testing Library

Testing library is a set of libraries and tools that makes testing easier than ever before. Its philosophy is to test your app in a way it is being used by a real world user instead of testing implementation details. For example, don't test what is the current state value in a component, but test what that component renders on the screen for its user. If you refactor your app to use a different state management solution, the tests will still be relevant as the actual component output to the user didn't change.

Cypress

Cypress is a tool for running e2e tests in an automated way. You define all the commands a real world user would execute when using the app and then start the test. It can be started in 2 modes:

- Browser mode - it will open a dedicated browser and run your application from start to finish. You get a nice set of tools to visualize and inspect your application on each step. Since this is a more expensive option, you want to run it only locally when developing the application.
- Headless mode - it will start a headless browser and run your application. Very useful for integrating with CI/CD to run it on every deploy.

It is very configurable with plugins and commands. You can even pair it with Testing Library which is makes your tests even easier to write.

You can also write custom commands to abstract some common tasks.

Custom Cypress Commands Example Code

MSW

For prototyping the API use msw, which is a great tool for quickly creating frontends without worrying about servers. It is not an actual backend, but a mocked server inside a service worker that intercepts all HTTP requests and returns desired responses based on the handlers you define. This is especially useful if you only have access to the frontend and are blocked by some not implemented features on the backend. This way, you will not be forced to wait for the feature to be completed or hardcode response data in the code, but use actual HTTP calls to build frontend features.

It can be used for designing API endpoints. The business logic of the mocked API can be created in its handlers.

API Handlers Example Code

Data Models Example Code

Having fully functional mocked API server also handy when it comes to testing, you don't have to mock fetch, but make requests to the mocked server instead with the data your application would expect.

Go to TOC

Colophon

This book is created by using the following sources:

- Bulletproof-react - English
- GitHub source: alan2207/bulletproof-react/docs
- Created: 2022-12-10
- Bash v5.2.2
- Vivliostyle, https://vivliostyle.org/
- By: @shinokada
- Viewer: https://read-html-download-pdf.vercel.app/
- GitHub repo: https://github.com/shinokada/markdown-docs-as-pdf
- Viewer repo: https://github.com/shinokada/read-html-download-pdf