Design Principles

Composition

The key feature of React is composition of components. Components written by different people should work well together. It is important to us that you can add functionality to a component without causing rippling changes throughout the codebase.

For example, it should be possible to introduce some local state into a component without changing any of the components using it. Similarly, it should be possible to add some initialization and teardown code to any component when necessary.

There is nothing "bad" about using state or lifecycle methods in components. Like any powerful feature, they should be used in moderation, but we have no intention to remove them. On the contrary, we think they are integral parts of what makes React useful. We might enable more functional patterns in the future, but both local state and lifecycle methods will be a part of that model.

Components are often described as "just functions" but in our view they need to be more than that to be useful. In React, components describe any composable behavior, and this includes rendering, lifecycle, and state. Some external libraries like Relay augment components with other responsibilities such as describing data dependencies. It is possible that those ideas might make it back into React too in some form.

Common Abstraction

In general we resist adding features that can be implemented in userland. We don't want to bloat your apps with useless library code. However, there are exceptions to this.

For example, if React didn't provide support for local state or lifecycle methods, people would create custom abstractions for them. When there are multiple abstractions competing, React can't enforce or take advantage of the properties of either of them. It has to work with the lowest common denominator.

This is why sometimes we add features to React itself. If we notice that many components implement a certain feature in incompatible or inefficient ways, we might prefer to bake it into React. We don't do it lightly. When we do it, it's because we are confident that raising the abstraction level benefits the whole ecosystem. State, lifecycle methods, cross-browser event normalization are good examples of this.

We always discuss such improvement proposals with the community. You can find some of those discussions by the "big picture" label on the React issue tracker.

Escape Hatches

React is pragmatic. It is driven by the needs of the products written at Facebook. While it is influenced by some paradigms that are not yet fully mainstream such as functional programming, staying accessible to a wide range of developers with different skills and experience levels is an explicit goal of the project.

If we want to deprecate a pattern that we don't like, it is our responsibility to consider all existing use cases for it and educate the community about the alternatives before we deprecate it. If some pattern that is useful for building apps is hard to express in a declarative way, we will provide an imperative API for it. If we can't figure out a perfect API for something that we found necessary in many apps, we will provide a temporary subpar working API as long as it is possible to get rid of it later and it leaves the door open for future improvements.

Stability

We value API stability. At Facebook, we have more than 50 thousand components using React. Many other companies, including Twitter and Airbnb, are also heavy users of React. This is why we are usually reluctant to change public APIs or behavior.

However we think stability in the sense of "nothing changes" is overrated. It quickly turns into stagnation. Instead, we prefer the stability in the sense of "It is heavily used in production, and when something changes, there is a clear (and preferably automated) migration path."

When we deprecate a pattern, we study its internal usage at Facebook and add deprecation warnings. They let us assess the impact of the change. Sometimes we back out if we see that it is too early, and we need to think more strategically about getting the codebases to the point where they are ready for this change.

If we are confident that the change is not too disruptive and the migration strategy is viable for all use cases, we release the deprecation warning to the open source community. We are closely in touch with many users of React outside of Facebook, and we monitor popular open source projects and guide them in fixing those deprecations.

Given the sheer size of the Facebook React codebase, successful internal migration is often a good indicator that other companies won't have problems either. Nevertheless sometimes people point out additional use cases we haven't thought of, and we add escape hatches for them or rethink our approach.

We don't deprecate anything without a good reason. We recognize that sometimes deprecations warnings cause frustration but we add them because deprecations clean up the road for the improvements and new features that we and many people in the community consider valuable.

For example, we added a warning about unknown DOM props in React 15.2.0. Many projects were affected by this. However fixing this warning is important so that we can introduce the support for custom attributes to React. There is a reason like this behind every deprecation that we add.

When we add a deprecation warning, we keep it for the rest of the current major version, and change the behavior in the next major version. If there is a lot of repetitive manual work involved, we release a codemod script that automates most of the change. Codemods enable us to move forward without stagnation in a massive codebase, and we encourage you to use them as well.

You can find the codemods that we released in the react-codemod repository.

Interoperability

We place high value in interoperability with existing systems and gradual adoption. Facebook has a massive non-React codebase. Its website uses a mix of a server-side component system called XHP, internal UI libraries that came before React, and React itself. It is important to us that any product team can start using React for a small feature rather than rewrite their code to bet on it.

This is why React provides escape hatches to work with mutable models, and tries to work well together with other UI libraries. You can wrap an existing imperative UI into a declarative component, and vice versa. This is crucial for gradual adoption.

Scheduling

Even when your components are described as functions, when you use React you don't call them directly. Every component returns a description of what needs to be rendered, and that description may include both user-written components like `<LikeButton>` and platform-specific components like `<div>`. It is up to React to "unroll" `<LikeButton>` at some point in the future and actually apply changes to the UI tree according to the render results of the components recursively.

This is a subtle distinction but a powerful one. Since you don't call that component function but let React call it, it means React has the power to delay calling it if necessary. In its current implementation React walks the tree recursively and calls render functions of the whole updated tree during a single tick. However in the future it might start delaying some updates to avoid dropping frames.

This is a common theme in React design. Some popular libraries implement the "push" approach where computations are performed when the new data is available. React, however, sticks to the "pull" approach where computations can be delayed until necessary.

React is not a generic data processing library. It is a library for building user interfaces. We think that it is uniquely positioned in an app to know which computations are relevant right now and which are not.

If something is offscreen, we can delay any logic related to it. If data is arriving faster than the frame rate, we can coalesce and batch updates. We can prioritize work coming from user interactions (such as an animation caused by a button click) over less important background work (such as rendering new content just loaded from the network) to avoid dropping frames.

To be clear, we are not taking advantage of this right now. However the freedom to do something like this is why we prefer to have control over scheduling, and why `setState()` is asynchronous. Conceptually, we think of it as "scheduling an update".

The control over scheduling would be harder for us to gain if we let the user directly compose views with a "push" based paradigm common in some variations of Functional Reactive Programming. We want to own the "glue" code.

It is a key goal for React that the amount of the user code that executes before yielding back into React is minimal (**Your component's code (user code) should run as quickly as possible, and then return control back to React to handle rendering, updates, scheduling, etc**.). This ensures that React retains the capability to schedule and split work in chunks according to what it knows about the UI.

There is an internal joke in the team that React should have been called "Schedule" because React does not want to be fully "reactive".

Developer Experience

Providing a good developer experience is important to us.

For example, we maintain React DevTools which let you inspect the React component tree in Chrome and Firefox. We have heard that it brings a big productivity boost both to the Facebook engineers and to the community.

We also try to go an extra mile to provide helpful developer warnings. For example, React warns you in development if you nest tags in a way that the browser doesn't understand, or if you make a common typo in the API. Developer warnings and the related checks are the main reason why the development version of React is slower than the production version.

The usage patterns that we see internally at Facebook help us understand what the common mistakes are, and how to prevent them early. When we add new features, we try to anticipate the common mistakes and warn about them.

We are always looking out for ways to improve the developer experience. We love to hear your suggestions and accept your contributions to make it even better.

Debugging

When something goes wrong, it is important that you have breadcrumbs to trace the mistake to its source in the codebase. In React, props and state are those breadcrumbs.

If you see something wrong on the screen, you can open React DevTools, find the component responsible for rendering, and then see if the props and state are correct. If they are, you know that the problem is in the component's `render()` function, or some function that is called by `render()`. The problem is isolated.

If the state is wrong, you know that the problem is caused by one of the `setState()` calls in this file. This, too, is relatively simple to locate and fix because usually there are only a few `setState()` calls in a single file.

If the props are wrong, you can traverse the tree up in the inspector, looking for the component that first "poisoned the well" by passing bad props down.

This ability to trace any UI to the data that produced it in the form of current props and state is very important to React. It is an explicit design goal that state is not "trapped" in closures and combinators, and is available to React directly.

While the UI is dynamic, we believe that synchronous `render()` functions of props and state turn debugging from guesswork into a boring but finite procedure. We would like to preserve this constraint in React even though it makes some use cases, like complex animations, harder.

Configuration

We find global runtime configuration options to be problematic.

For example, it is occasionally requested that we implement a function like `React.configure(options)` or `React.register(component)`. However this poses multiple problems, and we are not aware of good solutions to them.

What if somebody calls such a function from a third-party component library? What if one React app embeds another React app, and their desired configurations are incompatible? How can a third-party component specify that it requires a particular configuration? We think that global configuration doesn't work well with composition. Since composition is central to React, we don't provide global configuration in code.

We do, however, provide some global configuration on the build level. For example, we provide separate development and production builds. We may also add a profiling build in the future, and we are open to considering other build flags.

Beyond the DOM

We see the value of React in the way it allows us to write components that have fewer bugs and compose together well. DOM is the original rendering target for React but React Native is just as important both to Facebook and the community.

Being renderer-agnostic is an important design constraint of React. It adds some overhead in the internal representations. On the other hand, any improvements to the core translate across platforms.

Having a single programming model lets us form engineering teams around products instead of platforms. So far the tradeoff has been worth it for us.

Implementation

We try to provide elegant APIs where possible. We are much less concerned with the implementation being elegant. The real world is far from perfect, and to a reasonable extent we prefer to put the ugly code into the library if it means the user does not have to write it. When we evaluate new code, we are looking for an implementation that is correct, performant and affords a good developer experience. Elegance is secondary.

We prefer boring code to clever code. Code is disposable and often changes. So it is important that it doesn't introduce new internal abstractions unless absolutely necessary. Verbose code that is easy to move around, change and remove is preferred to elegant code that is prematurely abstracted and hard to change.

Optimized for Tooling

Some commonly used APIs have verbose names. For example, we use `componentDidMount()` instead of `didMount()` or `onMount()`. This is intentional. The goal is to make the points of interaction with the library highly visible.

In a massive codebase like Facebook, being able to search for uses of specific APIs is very important. We value distinct verbose names, and especially for the features that should be used sparingly. For example, `dangerouslySetInnerHTML` is hard to miss in a code review.

Optimizing for search is also important because of our reliance on codemods to make breaking changes. We want it to be easy and safe to apply vast automated changes across the codebase, and unique verbose names help us achieve this. Similarly, distinctive names make it easy to write custom lint rules about using React without worrying about potential false positives.

JSX plays a similar role. While it is not required with React, we use it extensively at Facebook both for aesthetic and pragmatic reasons.

In our codebase, JSX provides an unambiguous hint to the tools that they are dealing with a React element tree. This makes it possible to add build-time optimizations such as hoisting constant elements, safely lint and codemod internal component usage, and include JSX source location into the warnings.

Dogfooding

We try our best to address the problems raised by the community. However we are likely to prioritize the issues that people are *also* experiencing internally at Facebook. Perhaps counter-intuitively, we think this is the main reason why the community can bet on React.

Heavy internal usage gives us the confidence that React won't disappear tomorrow. React was created at Facebook to solve its problems. It brings tangible business value to the company and is used in many of its products. Dogfooding it means that our vision stays sharp and we have a focused direction going forward.

This doesn't mean that we ignore the issues raised by the community. For example, we added support for web components and SVG to React even though we don't rely on either of them internally. We are actively listening to your pain points and address them to the best of our ability. The community is what makes React special to us, and we are honored to contribute back.

After releasing many open source projects at Facebook, we have learned that trying to make everyone happy at the same time produced projects with poor focus that didn't grow well. Instead, we found that picking a small audience and focusing on making them happy brings a positive net effect. That's exactly what we did with React, and so far solving the problems encountered by Facebook product teams has translated well to the open source community.

The downside of this approach is that sometimes we fail to give enough focus to the things that Facebook teams don't have to deal with, such as the "getting started" experience. We are acutely aware of this, and we are thinking of how to improve in a way that would benefit everyone in the community without making the same mistakes we did with open source projects before.