UI as an afterthought

UI 是事后烟

Or: how will React context and hooks change the game of state management?

或者：React 上下文和钩子将如何改变状态管理的游戏？

博文地址

A question people ask me regularly: “How do all the new React features (context, hooks, suspense) affect how we build (web) apps in the future? Do they make state management libraries like Redux or MobX obsolete?”

人们经常问我的一个问题：“所有新的 React 特性（上下文、钩子、悬念）如何影响我们未来构建（Web）应用程序的方式？他们会让像 Redux 或 MobX 这样的状态管理库过时吗？”

With this post, I’ll try to answer that question once and for all! To truly understand the question, we’ll need to do a little groundwork. Let’s step back, and leave React, Redux and MobX alone while we answer a more fundamental question.

通过这篇文章，我将尝试一劳永逸地回答这个问题！要真正理解这个问题，我们需要做一些基础工作。让我们退后一步，把 React、Redux 和 MobX放一边 ，我们回答一个更基本的问题。

What is a web application? For the purpose of this post: A web application is *a* user interface that allows customers to interact with your business. The key take-away here is that it is ***a*** user interface. Not ***the***. The goal of a nice front-end: provide a nice, friction-less experience to your customers to interact with your business processes. But the front-end is not the business itself!

什么是网络应用程序？就本文而言：Web 应用程序是一个用户界面，允许客户与您的业务进行交互。这里的关键是它是一个用户界面。不是。良好前端的目标：为您的客户提供良好、无摩擦的体验，以与您的业务流程进行交互。但前端不是业务本身！

As thought experiment, imagine that inbox.google.com stops working (oh, wait, it will…😭). Theoretically users could pick up the phone, call google, identify themselves and ask a google employee: please tell me, what messages do you have waiting for me? This mental exercise is a great way to figure out what your business is about. If a customer walked by your office, what questions would they ask? What pieces of information would you try to save if the office is about to burn down? What user interactions ultimately generate money of your business?

作为思想实验，想象一下 inbox.google.com 停止工作（哦，等等，它会……😭）。理论上用户可以拿起电话，打电话给谷歌，表明自己的身份，然后问谷歌员工：请告诉我，你有什么消息在等我？这

种心理锻炼是了解您的业务内容的好方法。如果客户路过您的办公室，他们会问什么问题？如果办公室即将被烧毁，您会尝试保存哪些信息？哪些用户交互最终会为您的业务带来收益？

I notice that in front-end development we often approach user interfaces from the opposite angle: We start from some mock-ups and then add pieces of state at almost arbitrarily places to make the whole thing come alive. Basically, state and data is an afterthought, a necessary evil that is needed to make that beautiful UI work. Working your application from that side inevitably leads to the conclusion: *State is the root of all evil*. It’ s that horrible thing that makes everything that was beautiful at first, ugly and complicated. But here is a counter-thought:

我注意到在前端开发中，我们经常从相反的角度处理用户界面：我们从一些模型开始，然后在几乎任意的地方添加状态片段，使整个事物变得生动起来。基本上，状态和数据是事后才想到的，是使漂亮的 UI 工作所必需的必要之恶。从那方面处理您的应用程序不可避免地会得出结论：状态是万恶之源。正是这种可怕的东西让起初美好的一切变得丑陋而复杂。但这里有一个相反的想法：

> ## State is the root of all revenue.
>
> 状态是一切变化的源头

Information. The opportunity for customers to interact with business processes are ultimately the only thing that make money. Yes, a better UI experience will most likely lead to more money. But it is not the money generator itself.

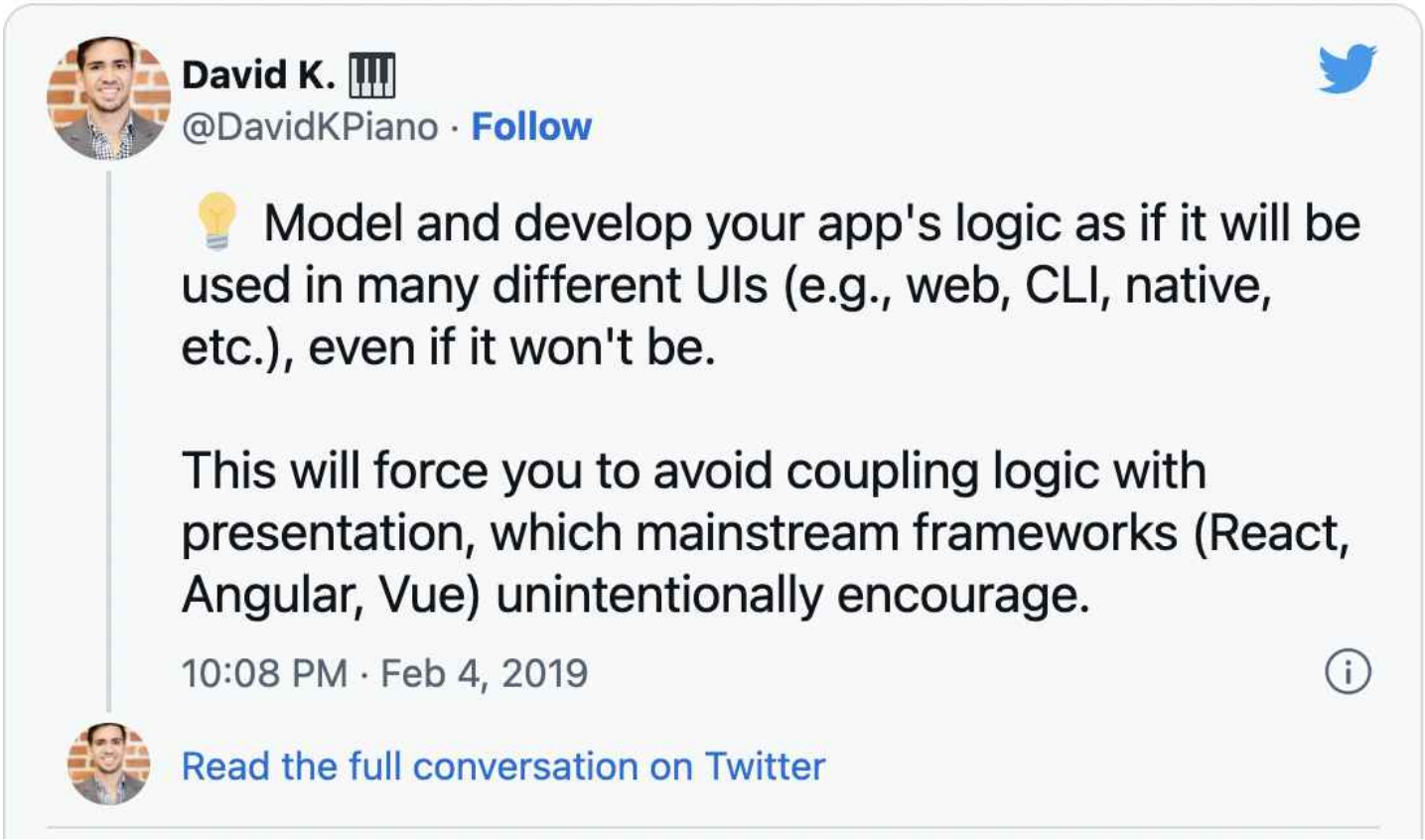
信息。客户与业务流程交互的机会最终是唯一赚钱的东西。是的，更好的 UI 体验很可能会带来更多的钱。但它不是货币发生器本身。

So, in my humble opinion, we should approach building web apps from the opposite direction, and first encode what interactions our customers will have with our systems. What are the processes. What is the information he will need? What is the information he will send? In other words, let’ s start with modelling our problem domain.

因此，以我的拙见，我们应该从相反的方向构建 Web 应用程序，并首先编码我们的客户将与我们的系统进行的交互。有哪些流程。他需要什么信息？他将发送什么信息？换句话说，让我们从建模我们的问题域开始。

The solutions to these problems are things we can code without reaching for a UI library. We can program the interactions in abstract terms. Unit test them. Build a deep understanding of what different states all these processes can be in.

这些问题的解决方案是我们无需使用 UI 库即可编写代码。我们可以用抽象的术语对交互进行编程。对它们进行单元测试。深入了解所有这些过程可能处于哪些不同状态。



Model and develop your app's logic as if it will be used in many different UIs (e.g., web, CLI, native, etc.), even if it won't be.

This will force you to avoid coupling logic with presentation, which mainstream frameworks (React, Angular, Vue) unintentionally encourage.

At this point, it doesn’t matter yet what the nature of the tool is that the customers use to interact with your business. A web app? A React native application? An SDK as NPM module? A CLI? It doesn’t matter! So:

在这一点上，客户用来与您的业务交互的工具的性质是什么并不重要。网络应用程序？一个 React 原生应用程序？作为 NPM 模块的 SDK？命令行界面？没关系！所以：

> *Initially, design your state, stores, processes as if you were building a CLI, not a web app.*

> 最初，设计您的状态、存储、流程，就好像您正在构建一个 CLI，而不是一个 Web 应用程序。

Now you might be wondering: “Aren’t you horrible over-engineering? Why should I build my app as if I am about to release a CLI? I am clearly never going to do that…. Are you unicorn-

puking me?”

现在你可能想知道：“你不是很可怕的过度设计吗？为什么我应该像即将发布 CLI 一样构建我的应用程序？我显然永远不会那样做.....你是在吐我的独角兽吗？”

Now, stop reading this blog for a moment and go back to the project you are currently procrastinating on, and fire up your test runner.... Now tell me again: Does your app have a CLI or not? Every dev on your team has a CLI (hopefully): the test runner. It interacts with and verifies your business processes. The fewer levels of indirection that your unit tests need to interact with your processes, the better. Unit tests are the second UI to your system. Or even the first if you apply TDD.

现在，暂时停止阅读这篇博客，回到你目前正在拖延的项目，并启动你的测试运行器.....现在再次告诉我：您的应用程序是否有 CLI？您团队中的每个开发人员都有一个 CLI（希望如此）：测试运行程序。它与您的业务流程交互并验证您的业务流程。单元测试与流程交互所需的间接级别越少越好。单元测试是系统的第二个 UI。如果您应用 TDD，甚至是第一个。

React does a really awesome job to allow unit tests to understand a component UI and interact with it (without having a browser and such). But still, you should be able to test without the indirections introduced by concepts such “mounting”, “rendering” (shallow or not?), “firing events”, “snapshotting UI”. These are all concepts that don’t matter for the business domain, and unnecessarily bind your logic to React.

React 在允许单元测试理解组件 UI 并与之交互（无需浏览器等）方面做得非常棒。但是，您应该能够在没有诸如“安装”、“渲染”（浅或不浅？）、“触发事件”、“快照 UI”等概念引入的间接性的情况下进行测试。这些都是对业务领域无关紧要的概念，并且不必要地将您的逻辑绑定到 React。

> *Nothing beats the simplicity of invoking your business processes directly as a set of functions.*

> 没有什么比将业务流程作为一组函数直接调用的简单性更好的了。

So, at this point you might have some clue why I always have been an opponent of directly capturing domain state in React component state. It makes the decoupling of business processes and UI unnecessarily complicated.

所以，到这里你可能已经有了一些线索，为什么我一直反对在 React 组件状态中直接捕获域状态。它使业务流程和 UI 的解耦变得不必要地复杂。

If I was to build a CLI for my app, I would probably use something like yargs or commander. But that doesn’t mean that because a CLI happens to be my UI, that I expect those libraries to suddenly manage the state of my business processes. In other words, that I would be willing to pay for a full rewrite, just to switch between yargs and commander. React is to me like a CLI lib, a

tool that helps to capture user input, fire of processes, and to transform business data into a nice output. It’ s a library to build user interfaces. Not business processes.

如果我要为我的应用程序构建 CLI，我可能会使用 yargs 或指挥官之类的东西。但这并不意味着因为 CLI 恰好是我的 UI，我希望这些库突然管理我的业务流程的状态。换句话说，我愿意为完全重写付费，只是为了在 yargs 和指挥官之间切换。 React 对我来说就像一个 CLI 库，一个有助于捕获用户输入、进程触发以及将业务数据转换为良好输出的工具。这是一个构建用户界面的库。不是业务流程。

Only when you have captured the customer processes, tested, and verified them, it starts to matter what the actual UI should look like. What technology it is built with. You will find yourself in a very comfortable position: As you begin building components, you will find that they don’ t need that much state. Some components will have some state of their own, as not all UI state is relevant for your business processes (all volatile state like current selection, tabs, routing, etc). But,

> *most components will be dumb.*

> 大多数组件都是愚蠢的。

You will also discover that testing becomes simpler; you will write way less tests that mount components, fire events etc. You still want some, to verify that you wired everything correctly, but there is no need to test all possible combinations.

你还会发现测试变得更简单了；您将编写更少的测试来安装组件、触发事件等。您仍然需要一些测试来验证您是否正确连接了所有内容，但无需测试所有可能的组合。

The great decoupling allows very rapid UI iteration, A/B testing, etc. Because once the domain state and UI have been decoupled, you are much more free to restructure your UI. Heck, even switch to a completely different UI lib or paradigm becomes cheaper. Because state is largely unaffected by it. Which is great, as in most applications I’ ve seen the UI develops in much higher pace than the actual business logic.

出色的解耦允许非常快速的 UI 迭代、A/B 测试等。因为一旦域状态和 UI 解耦，您可以更自由地重构 UI。哎呀，甚至切换到完全不同的 UI 库或范例变得更便宜。因为状态在很大程度上不受它的影响。这很棒，因为在我见过的大多数应用程序中，UI 的开发速度远高于实际的业务逻辑。

For example, at Mendix we used the above model with great success. This separation has become the paradigm everyone naturally follows. An example: The user needs to upload an excel sheet, next we run some client side validations on it, then we interact with the server, and finally we kick off some processes. Such a new feature would first result in a new store (just a plain JS class) that captures the internal state and methods for every step of the process. It would capture the logic for verification. The interactions with the back-end. And we would create unit tests to verify that the right validations messages are generated and that the whole the process works correctly under all it’ s state permutations and error conditions. Only after

that point, people start building the UI. Pick a nice upload component, build forms for all the steps. And maybe change the original validation messages if they don’t sound right.

例如，在 Mendix，我们成功地使用了上述模型。这种分离已经成为大家自然遵循的范式。一个例子：用户需要上传一个excel表格，接下来我们在上面运行一些客户端验证，然后我们与服务器交互，最后我们启动一些流程。这样的新特性首先会产生一个新的存储（只是一个普通的 JS 类），它为流程的每个步骤捕获内部状态和方法。它将捕获用于验证的逻辑。与后端的交互。我们将创建单元测试来验证是否生成了正确的验证消息，并且整个过程在所有状态排列和错误条件下都能正常工作。只有在那之后，人们才开始构建 UI。选择一个不错的上传组件，为所有步骤构建表单。如果它们听起来不正确，可能会更改原始验证消息。

At this point you might also understand why I am not a fan of the things that mix back-end interaction directly into the UI. Such as the react-apollo bindings as means to interact with graphQL. Back-end interaction like submitting mutations or fetching data is the responsibility of my domain stores. Not the UI layer. React-Apollo so far feels to me as a shortcut that too easily leads to a tightly coupled setup.

在这一点上，您可能也理解了为什么我不喜欢将后端交互直接混合到 UI 中的东西。例如 react-apollo 绑定作为与 graphQL 交互的手段。诸如提交突变或获取数据之类的后端交互是我的域存储的责任。不是 UI 层。到目前为止，React-Apollo 在我看来是一条捷径，很容易导致紧密耦合的设置。

Finally! It is time to go back to our original question: “How do all the new react features (context, hooks, suspense) affect how we build (web) apps in the future? Do they make state management libraries like Redux or MobX obsolete?”.

最后！是时候回到我们最初的问题了：“所有新的 React 功能（上下文、挂钩、悬念）如何影响我们未来构建（Web）应用程序的方式？他们会让 Redux 或 MobX 等状态管理库过时吗？”。

The answer for me is: The new features don’t change the game of state management. The context and hooks features don’t enable React to do new tricks. These are just the same tricks, significantly better organized, more composable and in a less error prone way (clearly, I’m a fan!). But React, out of the box, can only respond to state that is owned by components. If you want your domain state to live outside your component tree, you will need a separate state management pattern, abstraction, architecture, library, to organize that.

我的答案是：新功能不会改变状态管理的游戏规则。上下文和钩子特性不能让 React 做新的技巧。这些只是相同的技巧，组织得更好，更可组合并且更不容易出错（显然，我是粉丝！）。但是开箱即用的 React 只能响应组件拥有的状态。如果您希望域状态位于组件树之外，则需要一个单独的状态管理模式、抽象、架构、库来组织它。

> Recent React additions don’ t fundamentally change anything in the state management game.

> 最近的 React 添加并没有从根本上改变状态管理游戏中的任何内容。

In other words: if you just figured you don’ t need Redux or MobX anymore since the introduction of Context and hooks: Then you didn’ t need those before either.

换句话说：如果你只是认为自从引入 Context 和 hooks 之后你不再需要 Redux 或 MobX：那么你之前也不需要它们。

Note that with hooks, there is now less reason to use MobX to manage your local component state. Especially considering that MobX observables as *component* state won’ t be able to leverage the benefits of suspense.

请注意，有了hooks，现在使用 MobX 来管理本地组件状态的理由就更少了。特别是考虑到 MobX 的 observables 作为组件状态将无法利用悬念的好处。

Talking about suspense versus state management in general: I think it just proves the sanity of the separation of concerns: Suspense + React state is great to manage all the UI state, so that there can be concurrent rendering and such. Supporting concurrency makes a lot of sense for volatile state like UI state. But for my business processes? Business processes should be exactly in one state only at all times.

总体上谈论 suspense 与状态管理：我认为它只是证明了关注点分离的合理性：Suspense + React 状态非常适合管理所有 UI 状态，因此可以进行并发渲染等。支持并发对于 UI 状态等易变状态非常有意义。但是对于我的业务流程呢？业务流程应该始终只处于一种状态。

So, with that, we hopefully answered the question about modern React versus state management:

所以，有了这个，我们希望能回答关于现代 React 与状态管理的问题：

> React should manage volatile UI state, not your business processes. And for that reason, nothing really changed.

React 应该管理易变的 UI 状态，而不是你的业务流程。出于这个原因，没有什么真正改变。

Finally, a quick note about MobX / MobX-state-tree, you might now better understand their underlying goals. They were designed to:

1. Be able to manage state independently of any UI abstraction.
2. Neatly and transparently hook up the state they mange to the UI.

3. Avoid the error prone business of managing subscriptions, selectors, and other manual optimizations to make sure events don’ t cause to many components to re-render.

If you want to know how cool working with separately organized domain state is, watch my “Complexity: Divide & Conquer” talk or read this previous blog: “How to decouple State and UI”

Olufemi Adeojowrote about this recently as well: “The curious case of reusable state management” .

Before we part, let’ s go meta: every blogger knows that a blog needs images to engage users. This blog didn’ t feature any images yet and hence has a poor, sub-optimal UI. But still can perform all it’ s “business goals”: Sharing the above thoughts with you. Because, although extremely crucial, from an implementation perspective: