Automation & Full Stack JavaScript

WHITEPAPER

Ashton Morris

TABLE OF CONTENTS

Introduction

Fast, good or cheap? Pick two.

The “Iron Triangle” of project management dictates that “fast” translates to schedule; “good” to scope; and “cheap” to budget.

“Pick two,” bluntly states we are constrained to optimizing two of those three. The goal of this whitepaper is to document high-level processes to “pick three” with the following core values in mind: abstract, automate, and simplify. There is a secondary focus on writing platform, library, and UI agnostic software that scales.

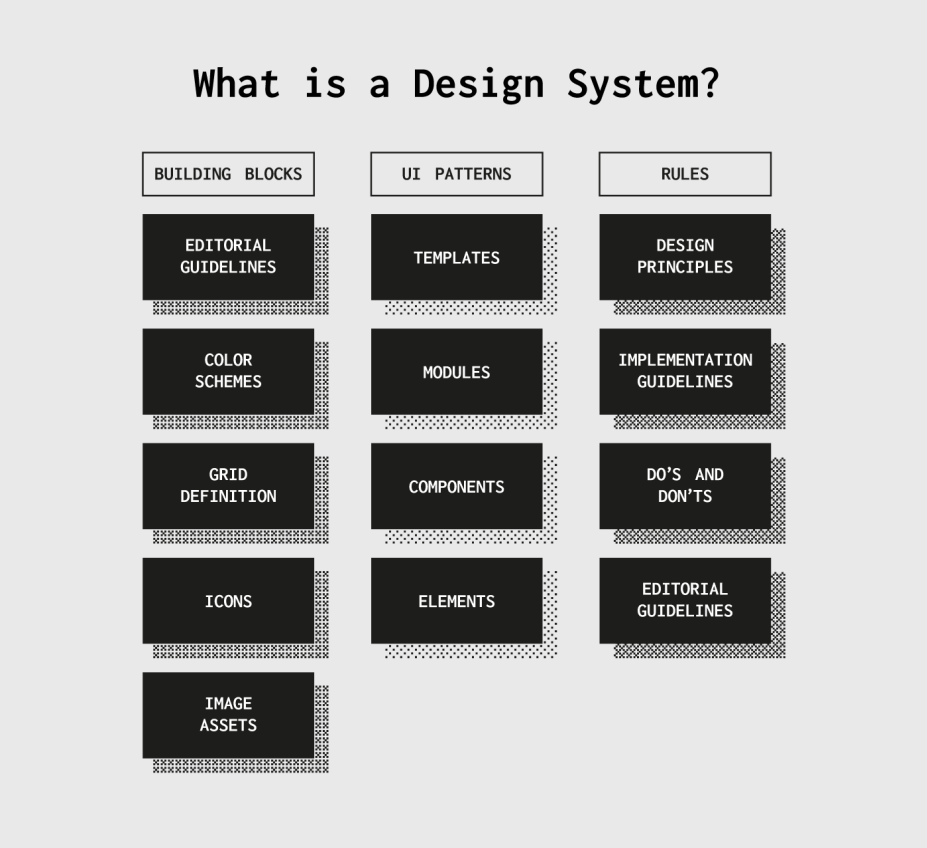
Breaking The Frontend Monolith: Design Systems

It’s about systemizing the design. As a noun. That’s literally all a design system means. That’s not to say only designers can do it.

-- Jina (@jina)

A holistic deep-dive into design systems is a herculean task that’s too heavy for this document. An amazing resource for further learning about this topic is “Design Systems and Front-End Architecture” by Stuart Robson.

The building blocks of a Design System is succinctly shown here:



Design Systems vs. Pattern Libraries vs. Style Guides – What’s the Difference?

We’re adding a new column named “Generation” as the column group name and the following blocks appear beneath it: state, operations, resolvers, and types. These are the building blocks of business logic.

> “The business logic layer is where you tackle the problems your program was created to solve.”
>
> *-- David Wall, in Multi-Tier Application Programming with PHP, 2004*

It’s unlikely that you’ll see business logic in this context appear as a candidate for abstraction due to it being tightly coupled with the unique problems you’re intending to solve.

As far as the back-end is concerned, business logic refers to how outbound and inbound data is handled, or resolvers. The definition of front-end business logic can be distilled into two concepts: operations that facilitate communication with the back-end, and state that manages the program’s behavior.

It’s JavaScript, all the way down

> Some ancient Asian cosmological views are close to the idea of an infinite regression of causes, as exemplified in the following apocryphal story: A Western traveler encountering an Oriental philosopher asks him to describe the nature of the world: “It is a great ball resting on the flat back of the world turtle.” “Ah yes, but what does the world turtle stand on?” “On the back of a still larger turtle.” “Yes, but what does he stand on?” “A very perceptive question. But it’s no use, mister; it’s turtles all the way down.”
>
> *-- Carl Sagan, Gott and the Turtles (1974)*

For the purpose of this paper, we’re going to assume a system architecture which has TypeScript as the language of choice in a monolingual, full stack design. This produces multiple benefits for the engineers contributing to this system. We’re going to focus on codeshare, with the primary goal of being intentional about coding for portability.

We can split our observational view of “codeshare” into two distinct levels: macro, the organizational level; and micro, an individual application. The macro level covers obvious use cases such as libraries and design systems.

When constraining our vision to that of an individual application, the micro level, rarely do we see any opportunities for codeshare. However, an underutilized strategy that shines in a monolingual system is code generation.

The solution I’m proposing uses GraphQL schemas, which brings a degree of vendor lock. I won’t evangelize you on my choice, as these results can also be achieved with gRPC. With that said, the goal is very straightforward: generate three of the four building blocks of application business logic: types, resolvers and operations.

Automate the Boring Stuff

The underlying goal of code generation is to reduce labor and increase productivity and velocity. This is accomplished by essentially automating the task of instructing a frontend application on how to communicate with the server – operations.

By doing so, a huge chunk of the frontend work is completed and maintained whenever the API layer of the system is updated. This architecture is expensive due to having to stitch together several packages to produce an optimal solution. However, maintenance thereafter is relatively cheap considering it (read: should) synchronize with all API updates.

This paper proposes a solution with the “graphql-codegen” package, a plugin based tool. By parsing a GraphQL schema, it can generate TypeScript types and operations written in a number of popular graphql clients such as react-query and Apollo. And, with relative ease, this covers two of three business logic blocks.

The third, resolvers, refers to the logic that handles any request sent to the API. Barring edge-cases where the logic involved is too unique to predict, we’re at least able to generate basic CRUD operations given the information provided from a GraphQL schema. Bare in mind, this is purely speculative as I haven’t encountered a plugin that tackles this specific problem – yet.

Making State Portable

The front-end development world is the wild west, and it could stand to learn from what other engineering disciplines have known and employed for years.

-- John Yanarella, [xState Docs]

How many ways can one build an authentication form? xState is a JavaScript state management library that is itself library and framework agnostic – that is to say, written in vanilla JavaScript. The core concept of xState is that it manages state with finite state machines. For anyone that's worked with AWS Step Functions, this may ring a bell.

In lieu of generation, for tackling the state portion of our business logic we'll lean on codeshare or portability. To accomplish this, one has to architect ahead of time a reliable way to abstract common state flows, for what we'll call machines moving forward. Examples could include a form machine; authentication, checkout, product display, or comment section, etc.

If generalized appropriately, one can port these state flows between an infinite number of front-end systems. Not to mention the other benefits that ship with the xState ecosystem such as observability in the form of their state visualizer; and advanced features such as auto-generating tests with Jest or Cypress. The key is to make the machines highly available, perhaps in an npm package, so the machines' relevance can be maintained independently of the system using it.

Relevance in this context pertains to how well the problem has been solved. For example: updating an authentication machine to support multi-factor authentication, rather than implementing authentication infinite times across infinite applications. This approach switches the developers' focus from redundancy to maintenance and innovation whilst ensuring consistency.

Conclusion

At this vantage point, it may be difficult to string together the concepts, technologies, and strategies presented in this document. As such, there is an open source repository that showcases what has been presented here: scalable fullstack javascript.