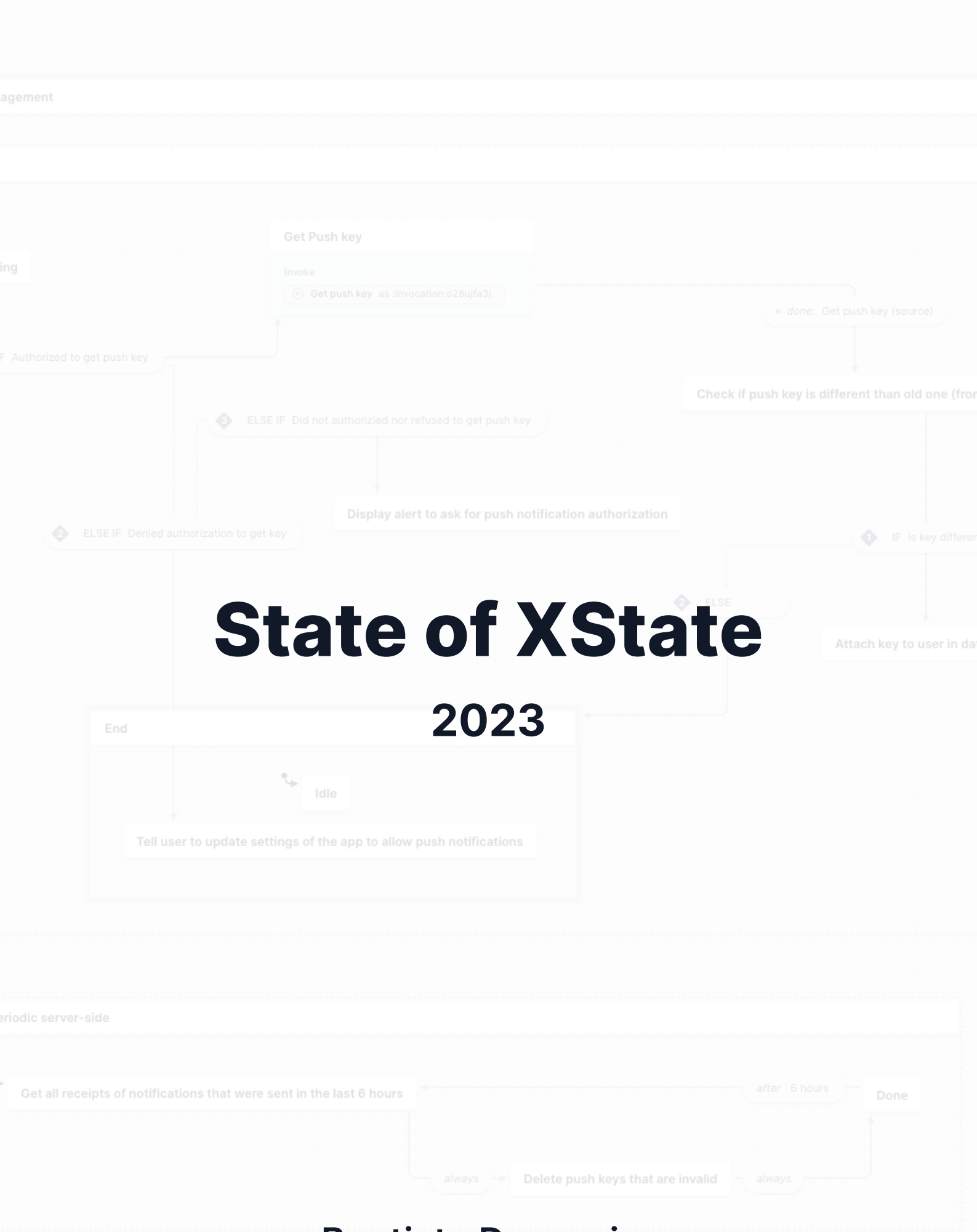


State of XState

2023

Baptiste Devessier

Preface

Thank you for downloading my report on the State of XState for 2023. This report details the most important areas of interest shared across the XState community, based on the responses I got to my survey and the interviews I conducted with the people willing to talk about their experience with XState.

Thirty-nine people responded to my survey, which took place between the 13th and 30th of May 2023. I contacted XState users before creating the survey to gather advice on which topics my final report should focus on. From the invaluable feedback received, I chose twenty questions for the survey. They focus on the subjects that XState developers want to learn about more from others in the industry.

I interviewed seven people who shared their feedback about XState with me. This feedback allowed me to provide real examples for all of the topics covered in the report.

I want to thank everybody who took the time to answer my emails, respond to the survey, and get interviewed. It matters a lot to me, so thank you! Their help made this report accurate on the current State of XState.

Summary

In this report, I'll cover five topics that matter to XState developers.

Onboarding new developers

XState reports being hard to learn. Teams employ different methods to teach XState to developers new to state machines. Reading XState documentation, watching video courses, and practicing modeling with exercises are a few ways to teach XState.

Modeling state machines

Modeling is known to be the most complicated part of creating state machines. Finding out how to best solve a problem by modeling a state machine is a skill that takes practice. Sometimes it may also be better to stick to a simple solution using a more scoped tool instead of a state machine.

Typing machines

Types are crucial for building scalable software. XState supports typing machines. Throughout its history, XState has released different ways to type state machines. People expect different levels of type-safety for their state machines, and they have their preferences about the method to achieve it.

Splitting machines

It is common for state machines to quickly grow in size due to adding more and more features. Splitting machines into smaller ones is a way to decouple independent blocks, as we do with separate functions. XStaters adopted several patterns to break their state machines and to be able to maintain complex systems of actors.

Testing with XState

Testing usually leads to divergent opinions, and it is the same for testing XState machines. Some XStaters choose to unit-test their state machines. Others run integration or e2e tests that don’t worry about implementation details. Model-Based Testing is also a pattern used to improve the reliability of software.

Onboarding new developers

When discovering XState, most people also learn the foundational concepts of XState: state machines, finite states, transitions, services, statecharts, hierarchical states, etc.

Though XState's API is complete and offers tools to solve advanced issues, it is not learning the API of the library that takes most of the time; it's learning how to use it efficiently to model things properly.

Many teams worldwide are using XState to develop software, and they are all facing the same problem: **How to teach XState to other developers and newcomers in their team?**

On this aspect, XState does not resemble most of the libraries, like React, TanStack Query, or date-fns, that solve a specific problem and are well-bounded: XState, on its own, does *nothing*; it's a tool that allows expressing any logic declaratively and explicitly, and thus it can be used to do *anything*.

The scope of XState is less identifiable, and it can be challenging for people to see the benefits of XState at first.

Learn with the documentation

83% of respondents to the survey declare asking newcomers to read XState's documentation to learn XState.

It has been the official way to learn XState for a long time, and this is how I learned XState. I visited https://xstate.js.org/docs/ so much that my browser now forces me to go there when I type *xstate*.

My browser forces me to go to XState's documentation when I type `xstate`.

Stately's team is working on updating the documentation, which will now be live at https://stately.ai/docs.

Be sure to follow the work they'll be doing there, as it will be amazing! However, my browser must learn the new URL of XState's documentation.

Learn with video courses

One new way to learn XState is through David Khourshid's courses on Frontend Masters. At the time of this writing, the most recent courses are about State Machines in JavaScript with XState and State Modeling in React with XState.

44% of respondents declare they ask newcomers to learn XState with Frontend Masters' courses and others, like Kyle Shevlin's introduction to State Machines with XState.

I watched these courses and recommend them too; they are great ways to get an overview of XState's API and modeling techniques, the two major difficulties with learning XState.

Learn from your company's specificities

Another exciting way to learn XState is to understand it based on real-world scenarios from companies' issues.

Christian Grøngaard pair-programs with newcomers to teach them how to use XState to solve specific issues, and so do **67%** of the respondents.

Richard Seaman also bets on modeling exercises based on the company's business problems: newcomers have to model with XState a problem they have already faced. **25%** of the respondents use **modeling exercises** to teach XState.

Companies do teach XState to newcomers

Interestingly, **no respondent** has said that their team **requires XState skills to join it**, even if **15%** of respondents said they use XState **everywhere**, and **21%** use XState for **most of their logic**.

I wonder if we would get the same numbers if we asked teams developing with React whether React is a required skill. It's mainly because XState developers are rarer than React developers, and training developers in XState is part of the hiring process.

But it's also due to XState being more versatile and used in many different ways to solve various problems.

React is mainly used to build web and native apps with React Native, though it can also be used to create CLI.

At the same time, based on the survey results, XState is used for:

- Frontend (87%)
- Short-lived machines on the backend (23%)
- Back-end workflows (23%)
- Hardware (3%)

Master XState by mixing different learning paths

My advice to onboard developers to XState would be to mix reading the documentation with watching courses and applying XState to real problems with your teammates.

Most of the respondents combine these ways of learning. If you are learning XState on your own, Stately's community is here to help you on Discord.

Learning by example suits many people, and the XState Catalogue remains a good reference.

It takes time to get good at XState, so take it slow!

Modeling state machines

As I outlined in the first section about Onboarding new developers, learning how to model things is the most challenging part of learning XState overall. For each problem, there are several ways to model a state machine to solve it.

One of Chris Shank's hobbies is modeling: he helps people on Twitter and Stately's Discord to reason out statechart patterns applied to real-world problems.

You quickly see yourself modeling the behavior of objects from daily life, like a microwave.

Little state machines are favored

About modeling, one thing people asked was how big are other XState users' state machines. **59%** of the respondents create state machines with **less than ten states**, and **62%** create state machines with **between 10 and 49 states**.

Nobody declared making state machines with more than 50 states, and it surprised me quite a bit, as I sometimes do.

Creating mostly little state machines means solving more than just fundamental problems.

Christian Grøngaard writes a lot of little state machines to split independent parts of the logic into state machines that are easier to think about. He invokes and spawns these little machines and makes them communicate to implement the overall behavior.

Favoring little state machines over bigger ones is a trend shared across the community.

86% of the respondents **split their machines into small logic units**, like a machine for notifications or a machine per step in a multi-step form. At the same time, **utility machines**, like machines to fetch data with automatic revalidation, are only used by **37%** of the respondents.

Richard Seaman uses XState everywhere in his application and does a lot of HTTP calls, but he only uses utility machines occasionally. In the case of HTTP requests, he doesn't need things like revalidation, so the interest in having a utility machine for that is less evident. This is AHA programming: Avoid Hasty Abstractions until an abstraction is apparent.

Different ways of creating state machines

The most significant benefit of state machines over other ways of coding is their visual representation. The Stately team is developing tools to visualize and edit state machines and collaborate on them.

My workflow to create and edit state machines is to use the Stately extension for VS Code; I'm using it to perform 90% of the work on my machines, and I'm editing code directly to add types and make larger-scale changes like moving a lot of states, which I find easier with code at the moment.

33% of the respondents use Stately Editor embedded into VS Code, and **33%** of the respondents use Stately Editor online to collaborate on machines.

Cédric Radicia uses the online Stately Editor to collaborate with designers and product managers. He loves the tool's ability to convey meaning without words, through colors for groups of states and events, or screenshots of UI elements as states' descriptions (Stately Editor supports Markdown in descriptions!).

Naming conventions

When modeling things with XState, the most basic act is to create states and events. But how should they be named?

Most respondents use conventions for states and events from a technical background.

Indeed, **76%** of them declare using **camel-case** (`waitingForServerResponse`), and **39%** of them use **all-caps case** (`WAITING_FOR_SERVER_RESPONSE`). **24%** of the respondents use **sentence-case** (`Waiting for server response`).

I'm using sentence-case for most of my machines now. Since Stately released visual tools, I have written my state machines in plain English. Using words a non-developer can understand instead of technical jargon is helpful.

Deciding when to use a state machine

Another difficulty with modeling state machines is that sometimes, not using a state machine could be a better choice.

71% of the respondents use XState for **data fetching**.

Most of the time, when building a SPA or a React Native application, I prefer to use TanStack Query as it offers a caching system, automatic revalidation of queries, and many more things I would prefer not to reimplement across projects.

Javier Madueño uses TanStack Query for fetching instead of XState. Still, he points out that if there are dependencies between calls, he will use XState to orchestrate the invalidation of queries.

Richard Seaman finds it correct to use some `useState` hooks to solve simple problems, but he directly reaches for state machines when problems evolve and require more logic.

Getting good at modeling

The best way to get good at modeling is to start by reimplementing simple systems with XState: a little Tetris game you made, an authentication form, or a wizard form.

It's easier to start reasoning about state machines through the reimplementation of visual and interactive applications.

Then, less obvious features like delayed transitions (`after` transitions) become more intuitive, allowing you to implement debouncing and throttling better than usual.

After some time using state machines, the benefits of using XState for the state management of your applications become evident, and state machines proliferate quickly across your code base.

They can even reach your backend to drive your workflows.

This is how I learned XState, and I'm still constantly discovering new areas to use XState to solve complex problems.

Typing machines

For a few years, TypeScript has established its supremacy over all other type systems for JavaScript.

All libraries provide TypeScript types, most written in TypeScript.

Application code is also largely written in TypeScript, and developers expect all libraries to have strong types that make them more productive and more confident with the safety of their code.

XState complies with this convention and has provided several ways to type machines over time.

The evolution of ways to type machines

2019

In 2019, machines were defined through the `Machine` function. Types were provided as three generics to the `Machine` function:

- The shape of the context
- A union of events handled by the machine
- The *state schema*, which was a nested object mapping all the states of the machine and allowed only predefined states to be declared in the machine

2020

In 2020, the state schema was upgraded to become *typestates*, which allows declaring the shape of the context, depending on the state.

It will enable specifying that the data are not yet available in the `loading` state but are necessarily defined in the `succeed` state.

This is still in use in 2023 with XState v4, but it's no longer the recommended way to type machines.

2021

In 2021, the `createMachine` function deprecated the `Machine` function we all know and use nowadays.

Since 2022

In 2022, the typegen was released and became the recommended and most robust way to type machines.

To type the context and the events, we needed to define them through the schema property described in the configuration object of machines.

Initially, types could only be generated by Stately's VS Code extension, but the Stately team quickly released a CLI independent of the editor developers use.

It's interesting to note that, at first, it was recommended to commit the generated files due to the lack of a tool that could run in CI/CD environments.

When the CLI was released, the Stately team recommended *not* committing these files; some people are still committing these files for simplicity.

Thanks to the typegen, actions, guards, delays, and services receive an event argument restricted to the only events (the user-defined events and the internal events, such as the done event of an invocation) that can lead to them being called.

This drastically improves the type-safety of machines and eases development, as previously, we had to narrow the event type at runtime with functions like `assertEventType`.

Currently, the typegen is used by **59%** of the respondents to the survey.

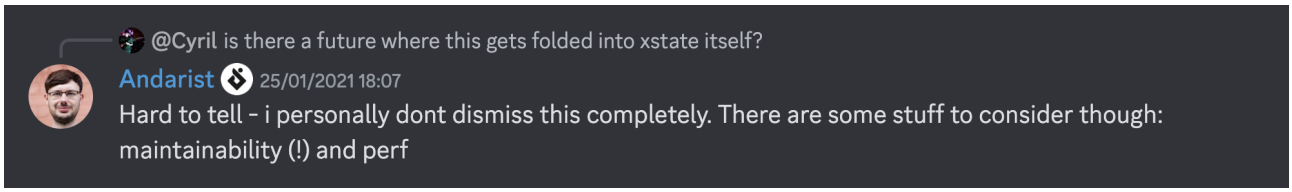
Typing without type-generation feels more natural

Though the typegen helps deal with types, some think it could be enhanced. Some people point out that it doesn't yet work with all of XState's features, like eventless transitions and send action.

Some people find that using a typegen system is a bit hacky and would expect XState to provide strong types inference without dealing with an external tool.

txstate is a project that tried to solve these problems without using a typegen system.

The possible issues with a system like `txstate` are that types become hard to maintain, and performances can become harmful, as Mateusz Burzyński pointed out in a discussion on Stately's Discord about `txstate`:



Determining how XState should implement types without relying on type-generation led to many discussions on Stately's Discord and in the GitHub Discussions in the XState repository.

Community does not yet agree on typing machines

Typing state machines is definitely the most controversial subject about XState.

Only **59%** of the respondents use **typegen**, and **28%** type their machines by **providing types for the context and the events without using the typegen**. **15%** are **not typing their machines** at all, and **8%** still use **typestates**, though it was sort of deprecated.

Please read this carefully: not using typegen doesn't mean unlinking XState!

Christian Grøngaard considers that he is benefiting so much from XState, as it's helping him solve problems that would have been too hard to solve without it, that he is okay with not having fully typed machines. He even says, ironically, that *he wouldn't know how to do frontend without XState now that he's so used to it*.

There are not a lot of tools that lead to so much love! I don't know if a construction worker would say the same about his hammer.

The future of typing XState machines

With XState v5, tools are coming to implement the *receptionist pattern* — a way to make actors across the hierarchy of machines communicate with each other directly.

People I interviewed hope there will be a way to create systems of actors fully typed so that actors, however they are deep in the actor tree, can send events to other actors in a typed way, that is, being able to send only the events that the actor is expecting. Nothing would prevent creating safe huge systems of actors anymore.

I have been using typegen since it was released, and I've loved the type-safety it provides to my machines. The first thing I do when creating a new machine is to enable the typegen on it.

I'm excited about the future of types with XState, dreaming of a way to type systems of actors fully.

Splitting machines

According to the results of my survey, XState developers are worried about creating and maintaining big state machines: **nobody** declared making state machines with **more than 50 states**.

Conversely, **59%** of respondents reported creating state machines with **less than ten states**.

Creating small state machines does not mean XState developers only use XState to solve less complex issues: **86%** of respondents make **separate machines for independent behaviors**.

One major foundation of XState is the *actor model* — pieces of logic communicate and can not mutate each other's state; they need to send *events*.

Solving complex problems with XState usually means finding which parts of a system are independent of others in order to extract them and make them more generic and less constrained to a specific use case.

One great example is an actor to handle notifications or toasts. Notifications can be triggered from any part of an application: after checking out an item or when receiving a new message in a chat room.

If you are using XState to implement that instead of a library taking care of those things for you, you will have to:

- Choose if the application can show multiple notifications simultaneously or if they need to be queued or replaced by each other.
- Automatically hide notifications after a delay of inactivity, or make the notification permanent.
- Hide the notifications if the user clicks a *Cancel* button or swipes the notification like on iOS or macOS.

It would be better to avoid reimplementing this complex logic whenever you want to display a notification.

Furthermore, I assume you would like a global notifications store: you are good to go with a global actor to handle notifications!

Actors are not a shiny way to make code look smart: they are an excellent way to split code into business logic chunks.

Nick Worrall uses the actor model a lot. For him, the actor model is safer and prevents unwanted side effects from happening. He tries to keep his state machines as small as possible to scope independent behaviors. He prefers systems of 50 little state machines over machines with 50 states.

Split generic features

We can not only split state machines by independent features but also split them to make some generic mechanisms reusable.

37% of respondents split machines into **reusable utility machines**.

A great example is a machine to fetch data. Fetching data seems trivial, but it becomes a lot more complex when implementing things like caching or revalidation that specialized tools like TanStack Query do.

The internal workings of TanStack Query (previously known as React Query) are impressive and could be implemented as a state machine.

You wouldn't want to implement revalidation when the user re-focuses the window for every HTTP call you make inside your state machines. The smarter thing to do would be to extract this logic into a reusable `**requestMachine**` that could support any RPC protocol.

Composing logic

People also compose machines by creating higher-order functions, like a `createRetryMachine`.

31% of respondents declare creating **mixin functions to compose machines**.

With XState v4, developers found it quite cumbersome to do so.

The Stately team understood that composing actors was difficult with XState v4.

They worked on a unified API for actors in XState v5, whatever their type is (a promise actor, a callback actor, an observable, or a machine). For instance, to define a callback actor, we must do so through a `fromCallback` function which will return the same object as the `fromPromise` function used to create a promise actor.

That way, composing actors will be so much easier to handle. Here is an example taken from the documentation:

```
function withLogging(actorLogic) {
  const enhancedLogic = {
    ...actorLogic,
    transition: (state, event, actorCtx) => {
      console.log('State:', state);
      return actorLogic.transition(state, event, actorCtx);
    },
  };

  return enhancedLogic;
}

const loggingToggleLogic = withLogging(toggleLogic);
```

Executing actors

Splitting machines is one thing; making XState run them is another one.

To make XState run a split machine part of a system, we have two choices: invoking or spawning it.

24% of respondents declare **invoking machine actors**, **89%** declare **invoking promise actors**, and **55%** say that they **invoke callback actors**.

At the same time, **63%** of respondents claimed to **spawn machines**, while **34%** declared **spawning promises** and **16%** declared **spawning callbacks**.

Machines are more often spawned than invoked.

When invoking an actor, this actor is attached to the state it was invoked by: if the machine exits the state, the machine stops the actor.

While a spawned actor lives until the machine it was invoked by is stopped, living across state transitions and being more independent of its parent machine than an invoked machine.

Spawned machines are handy when dealing with a non-finite number of business entities, like a list of conversations. Each chat conversation might be a machine spawned from a parent machine holding all the references to these spawned machines within its context.

I like to invoke machines that can only exist once in an application and those with a lifetime scoped to a specific *state*. If I were creating a video call application, I would invoke a `` `videoCallMachine` `` when my `` `appMachine` `` is in the `` `authenticated` `` state, as the user can start a video call at any time while being authenticated, but can no longer do so once signed out.

The actor model is the best part of XState, making it scalable to any system complexity.

XState v5 will greatly improve things, notably through the receptionist pattern and an API for higher-level actor logic.

Dealing with complex actor systems will be smoother than ever.

Testing with XState

Testing is a controversial topic across the computing engineering industry: ask ten developers how they are testing their software, and they will all have different answers.

The testing vocabulary conveys different meanings based on who you are talking to. Integration and end-to-end tests are the same for some people; they are different testing practices for others.

Across the web development industry, teams have differing views on what parts of their front-end code they should test. Some teams mock side effects and test logic code separately; some do snapshots of the HTML nodes returned by their components in a mocked browser; others test production code on real browsers.

Unsurprisingly, respondents to the survey disagreed on ways to test XState machines.

Unit tests

37% of the respondents are **unit testing** their machines.

For Marcin Cekiera, finding a quick and scalable way to unit test machines is challenging. He has struggled to avoid excessive boilerplate code required for testing machines.

Christian Grøngaard is also unit-testing his machines and has developed his own pattern to test machines.

Unit testing machines mean mocking actors, like invoked promises, invoked callbacks, and spawned machines, to focus tests on the transitions that are taken by a machine when it receives an event in a specific state.

Unit testing machines efficiently ensures machines do not get stuck or reach impossible states. Actors are tested independently.

Even though they are beneficial, unit tests should not be the only testing strategy a team uses, as these tests don't ensure separate blocks of code work properly together once assembled.

It's great to know that a window can be opened in isolation, but it's better to know it's openable in its real-world environment.

Integration and e2e tests

Integration and e2e tests become useful to ensure that the separate parts of a system work together: with them, the whole system is tested, or at least a large part of it. I'm using Kent C. Dodds' test classification, the Testing Trophy.

71% of the respondents use **integration and e2e testing** with XState.

Richard Seaman is testing his application's behavior through a user's eyes with integration and e2e tests. He runs more specific tests on actions and actors with unit tests. Both testing methods are complementary.

Alexandre Stahmer prefers to test the business logic of his state machines directly through e2e tests.

Testing is a complex topic in the industry, and that's also true about XState.

XState users disagree on a single pattern to test their state machines, and that's great! There are always many ways to pursue the same goal in our profession.

Beyond tests maintained by developers

11% of the respondents are **not responsible for testing**. Some of them delegate UI testing to QA testers.

Christian Grøngaard does not write e2e tests, as QA testers maintain them.

Javier Madueño delegates e2e tests to QA too, and he shares the state machines he built with them so that they have a visual representation of the system they are testing and can better find edge cases.

Building software with XState reduces the number of bugs and eases debugging. Making the code visual makes it easier to understand why a sequence of events led to an invalid state.

Aleksandra Desmurs-Linczewska uses XState Viz to test the machine separately from the application.

Model-Based Testing

Another interesting topic regarding testing and XState is Model-Based Testing.

Model-Based Testing is a way to test the software by modeling how it can be used by a user, for example, and generating all possible usage combinations.

These combinations can drive tests that should cover all the possible ways to use the software.

Depending on the model, these combinations may include all the possible sequences for a set of events.

The Stately team built a library for Model-Based Testing with XState: `@xstate/test`. With this library, we can model a system with an XState machine.

According to my survey results, Model-Based Testing is an advanced and controversial topic about XState.

26% of the respondents declared they **don't know what Model-Based Testing is**, and **51%** reported they **still need to give Model-Based Testing a try**.

`@xstate/test` is a powerful tool, but developers should only use it to test something that requires a heavy setup. **10%** of the respondents think **Model-Based Testing benefits don't outweigh the setup costs and the tests' complexity**.

In the end, **13%** of the respondents **used Model-Based Testing**.

And there is another segmentation here! Indeed, developers can run `@xstate/test` on implementation machines or specific machines built to represent the system from a user's point of view.

5% of the respondents do Model-Based Testing on their **implementation machines**.

Cédric Radicia is doing so. It allows him to write integration tests for scoped components, like a complex input, and ensure they work properly in whatever way a user interacts with them and in whatever sequence of events.

The other part of `@xstate/test` users (**8%** of the respondents) prefers to create a **separate machine** to model and run tests on the system.

Doing so is better to test a whole system independently of how it has been implemented: you can then use `@xstate/test` to test a codebase that wasn't coded with XState.

I usually use Model-Based Testing by creating a separate machine for the testing model. This pattern is useful to test complex flows such as an authentication flow, where validation and server errors are mixed with redirections.

The difficulty with `@xstate/test` is that the amount of generated paths quickly grows exponentially as you add states and events.

`@xstate/test` for XState v4 exposes two functions to generate paths: `getShortestPathPlans` and `getSimplePathPlans`. `getSimplePathPlans` generates all paths to all other reachable states; `getShortestPathPlans` generates only the shortest path to all other reachable states.

For XState v4, I recommend using `getSimplePathPlans` with a machine kept as small as possible to test all the combinations without leading to 500 tests being generated.

The Stately team is working on improvements to `@xstate/test` for XState v5; we'll see where this interesting testing method goes!

Wrap up

Thank you for having read my report about the State of XState. I hope you better understand the different ways the community uses XState.

I am Baptiste Devessier. I am a full-stack web and XState freelance consultant. When I am not consulting on freelance projects, you will find me writing articles, recording video tutorials, and working on my side projects.

I have been a passionate XState user since I first used it three years ago to implement the search bar of an e-commerce website, which included complex features to implement without XState, like debounced search-as-you-type.

XState and state machines are excellent tools to master, as they can ease solving complex problems. **Subscribe to my newsletter to learn how to use XState best.**

Credits

I thank again everyone who took the time to answer my survey and those who accepted to be interviewed.

The background image of the first page is a screenshot of Stately Editor.

I credit Tailwind Labs for the overall design of this document, for which I took inspiration from the Refactoring UI book.

This PDF document has been generated from an Astro website built with Svelte and Tailwind CSS with Typography plugin. I learned a lot about developing websites to be printed, and I wasn't expecting it to be so tricky at times.

I am grateful to Mia Combeau, Paul Rastoin, and Émilie Phe, who reviewed the report and showed support at every step of the journey. Without them, the report wouldn't be as good, and I would be less confident to present it to you today.