



# Focal

## Functional-reactive state management

Sergey Yavnyi / @glatteis

Nov, 2018

Hi, my name is Sergey 

- User-facing products @ [Grammarly](#)
- Functional programming + static typing = 

[github.com/blacktaxi](https://github.com/blacktaxi)



Le|

- A writing app (*grammar, spelling and more!*)
- Web (+ extensions!), desktop, mobile
- 10M+ active users



*(since 1.8!)*

# THIS TALK: FOCAL

- State management for React
- Immutable state store
- Easy to use
- Type safe

[github.com/grammarly/focal](https://github.com/grammarly/focal)

```
npm i -S @grammarly/focal
```

# REDUX?

# REDUX



- Predictable, consistent state
- Dev tools



- Conciseness (actions, reducers, middleware, async)
- Modularity (combineReducers)
- Type safety (you'll need a [guide](#))
- Performance (shouldComponentUpdate?)

PREDICTABLE, CONSISTENT STATE

Worth it.

*...but can we do better?*

# CONCISENESS

Atom – the "store"

```
// create the store
const state = Atom.create({ count: 0 })

// update state
state.modify(s => ({ ...s, count: s.count + 1 }))
```

*modify: functional update (similar to a reducer)*

# CONCISENESS

```
type AppState = number
```

## Redux

```
type AppAction = { type: 'INC' }

const increment = () => ({ type: 'INC' })

const reducer = (s: AppState, a: AppAction) => {
  switch (a.type) {
    case 'INC': return s + 1

    // ...
  }
}

//...
<button onClick={() => props.dispatch(increment())} />
```

# Focal

```
//...
<button onClick={() => props.state.modify(s => s + 1)} />
```

# MODULARITY

```
type AppState = { count: number }
```

# MODULARITY

## Redux

```
type AppAction = { type: 'INC' }

const increment = () => ({ type: 'INC' })

const reducer = (s: AppState, a: AppAction) => {
  switch (a.type) {
    case 'INC': return { ...s, count: s.count + 1 }

    // ...
  }
}

const mapStateToProps = (state: AppState) => ({
  count: state.count
})

// to be continued...
```

## ...Redux

```
// ...continued
const Counter = connect(mapStateToProps)(
  (props: {
    count: number; dispatch: (a: AppAction) => void
  }) =>
  <p>
    Value: {props.count}
    <button onClick={() => props.dispatch(increment())}></button>
  </p>
)

const App = () =>
  <div>
    <Counter />
  </div>

// ...flip over to the next slide ↩
```

## ...Redux

```
// ...continued

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
// DONE! 💪
```

# MODULARITY

## Focal

```
const Counter = (props: { count: Atom<number> }) =>
  <F.p>
    Value: {props.count}
    <button onClick={() => props.count.modify(x => x + 1)} />
  </F.p>

const App = (props: { state: Atom<AppState> }) =>
  <div>
    <Counter count={state.lens('count')} />
  </div>

render(
  <App state={state} />,
  document.getElementById('root')
)
// That's it! 🎉
```

# Atom.lens(...)?

```
// type: Atom<{
//   todos: { items: string[] };
//   counter: { count: number };
// }>
const state = Atom.create({
  todos: {
    items: ['banana']
  },
  counter: {
    count: 0
  }
})
```

Break state into pieces

```
// type: Atom<{ items: string[] }>
const todos = state.lens('todos')

// type: Atom<{ count: number }>
const counter = state.lens('counter')

// type: Atom<number>
const count = counter.lens('count')
```

```
// type: Atom<{ count: number }>
state.lens('counter') ——————  

// type: Atom<number>
counter.lens('count') ——————  

  
const state = Atom.create({  
  todos: {  
    items: ['banana']  
  },  
  counter: {  
    count: 0  
  }  
})
```



# LENS?

A way to read/write a *part*<sup>1</sup> of immutable data

```
interface Lens<T, U> {  
    get(source: T): U // a getter  
    set(newValue: U, source: T): T // immutable update  
}
```

<sup>1</sup> – abstractly speaking

```
state.lens('count')
```

...just a short form of

```
state.lens(Lens.create(  
    // getter  
    (s: AppState) => s.count,  
    // setter  
    (v, s) => ({ ...s, count: v })  
))
```

# LENSES ARE FUN

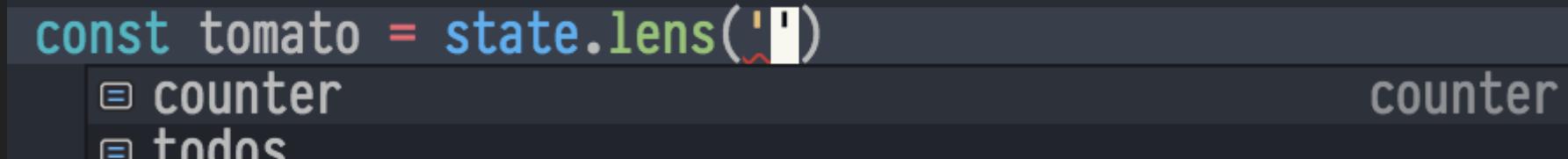


- Program imperatively using Haskell lenses
- partial.lenses, calmm-js

# TYPE SAFETY

```
const state = Atom.create(  
  { counter: { count: 0 }, todos: ['banana'] }  
)  
  
const count = state.lens('counter') // ✅ good to go  
  
// [ts] Argument of type '"potato"' is not assignable to  
// parameter of type '"counter" | "todos"'.  
const potato = state.lens('potato') // ❌ does NOT compile  
  
// [ts] Argument of type '"nope"' is not assignable to  
// parameter of type '{ count: number; }'.  
counter.set('nope') // ❌ no luck here
```

## Auto-completion



# REDUX + TS

[github.com/piotrwitek/react-redux-typescript-guide](https://github.com/piotrwitek/react-redux-typescript-guide)

## Table of Contents

---

- [Type Definitions & Complementary Libraries](#)
- [React Types Cheatsheet !\[\]\(0cf1de0a013c12b724e3acde3b2f780a\_img.jpg\) NEW](#)
- [Component Typing Patterns](#)
  - [Stateless Components - SFC](#)
  - [!\[\]\(1c108ad8fe9e61d7178ebe04ab721b96\_img.jpg\) Stateful Components - Class !\[\]\(7dcf48fddd351e3ddacf30e5bc93fcd2\_img.jpg\) UPDATED](#)
  - [Generic Components](#)
  - [Render Props !\[\]\(27098a06ea2a054d76bd80b8907db857\_img.jpg\) NEW](#)
  - [!\[\]\(9e2a99a7e5be25d92c34927621c21f02\_img.jpg\) Higher-Order Components !\[\]\(dd85859b09ced4f79be824dff1294950\_img.jpg\) UPDATED](#)
  - [Redux Connected Components](#)
- [Redux](#)
  - [!\[\]\(82aa5612a0e71a3811f519ee40b41375\_img.jpg\) Action Creators !\[\]\(b73eac04980eb13fc2a388f9e2e97f00\_img.jpg\) UPDATED](#)
  - [!\[\]\(a8dcea7e223d78c77088223afc00a210\_img.jpg\) Reducers !\[\]\(49576c177f6b68780da1bcfa9513d91b\_img.jpg\) UPDATED](#)

- State with Type-level Immutability
- Typing reducer
- Testing reducer
- Store Configuration  UPDATED
- Async Flow  UPDATED
- Selectors
- Typing connect  NEW
- Tools
  - TSLint
  - Jest
  - Enzyme
  - Living Style Guide  NEW
  - Common Npm Scripts

# <F.p>...?

F-components: put Atoms in JSX

Subscribe to Atoms on mount

```
<F.p onClick={() => state.modify(x => x + 1)}>  
  Count: {state}.  
</F.p>
```

Atom.subscribe(...)

```
state.subscribe(s => {  
  // print the app state as it's changing  
  console.log('new app state is:', s)  
})
```

# SUBSCRIBE?

Atom<T> extends Observable<T>



# Atom IS Observable

Can use RxJS operators on Atoms

```
state
  .debounceTime(500)
  .subscribe(s => {
    window.localStorage.setItem(
      'appState',
      JSON.stringify(s)
  })
})
```

# RXJS + REACT

<F.\* /> render Observables

```
<F.div>
  The time is {
    Observable.interval(1000)
      .startWith(0)
      .map(_ => new Date().toString())
  }.
</F.div>
```

Not just Atoms!

# OBSERVABLES

```
// simplified
interface Observable<T> {
    subscribe(
        onNext: (value: T) => void
    ): { unsubscribe(): void }
}
```

- First class event: *event as value*
- A stream of data: *collection of values over time*

# OBSERVABLES



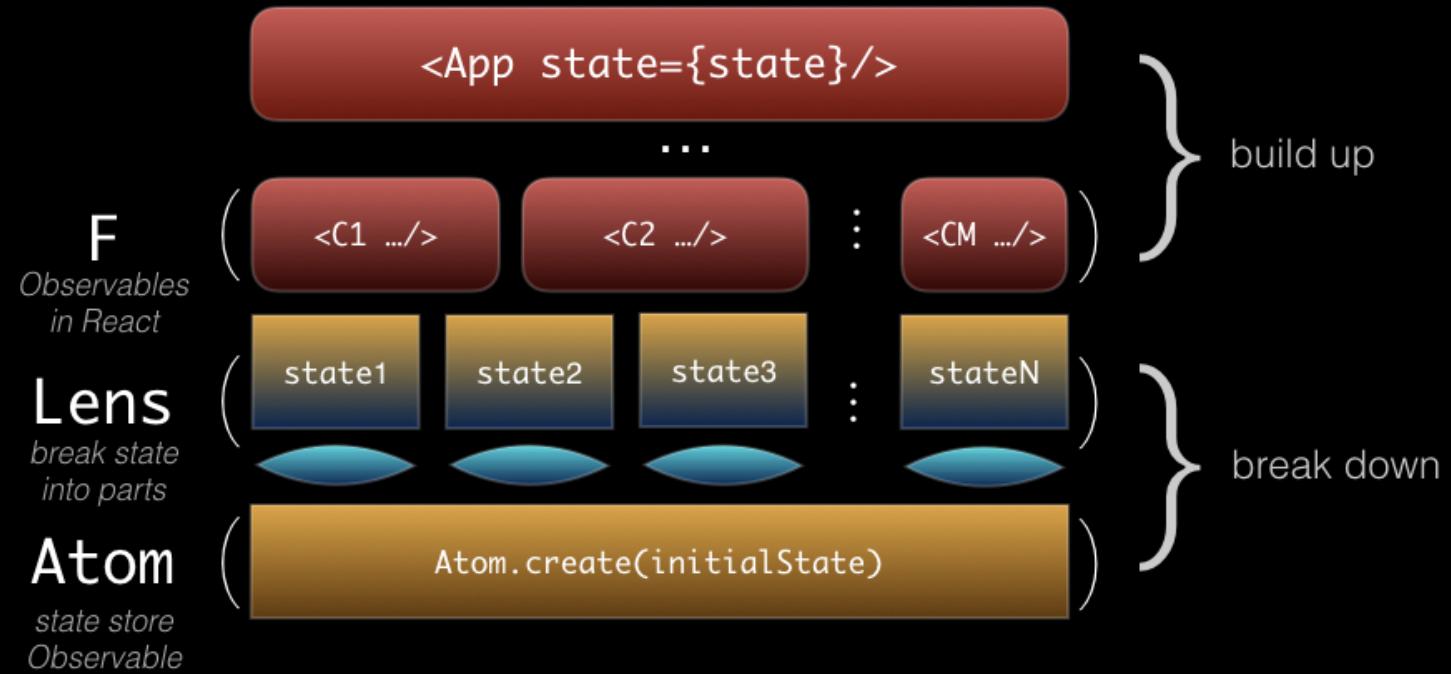
- <http://queue.acm.org/detail.cfm?id=2169076>
  - Your Mouse is a Database
- <https://goo.gl/fQvnzc>
  - Learning Observable By Building Observable
- <http://rxmarbles.com>

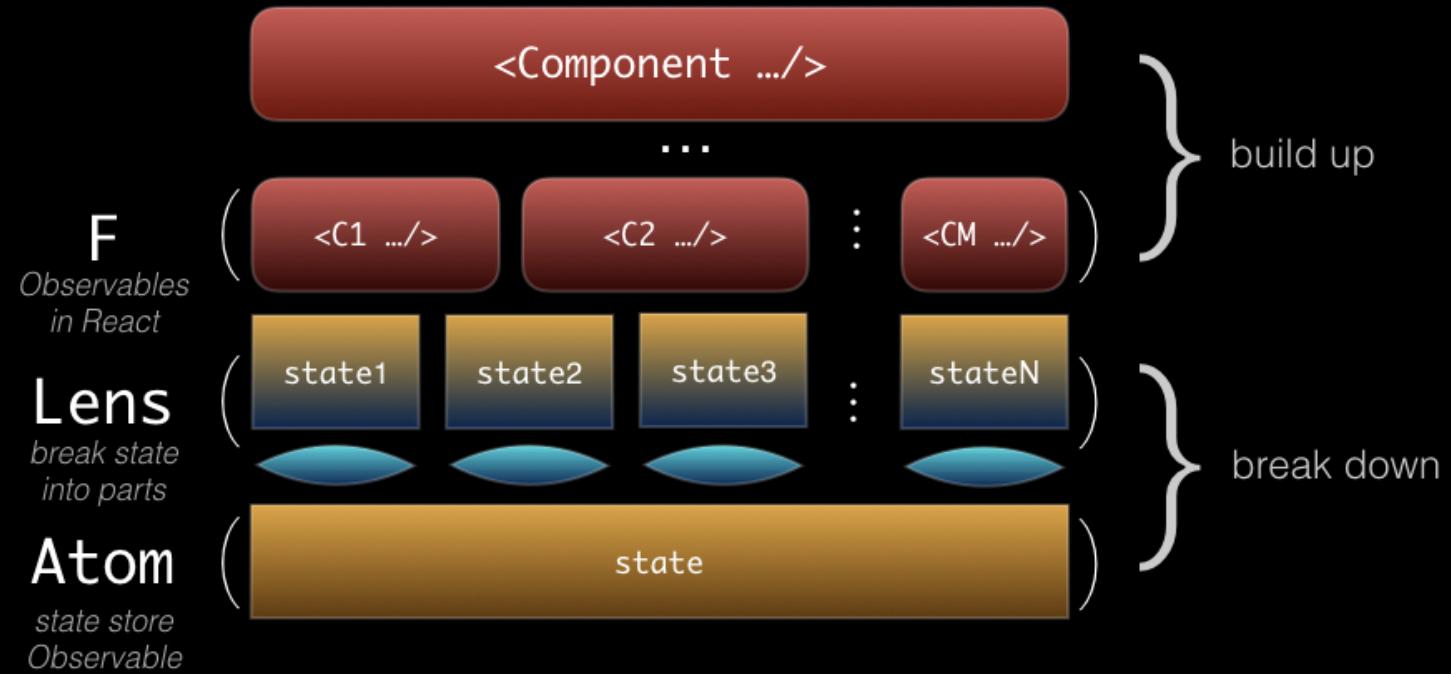
# PERFORMANCE

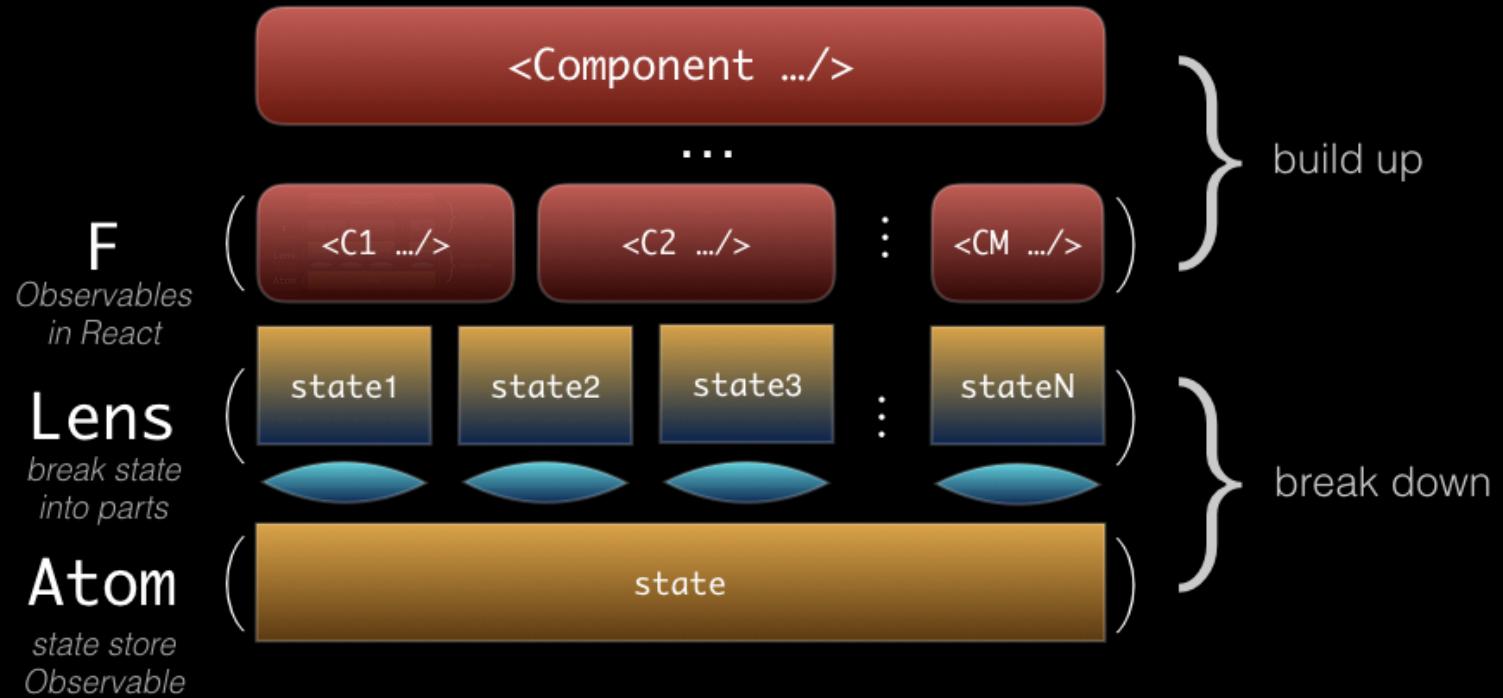
~~shouldComponentUpdate~~

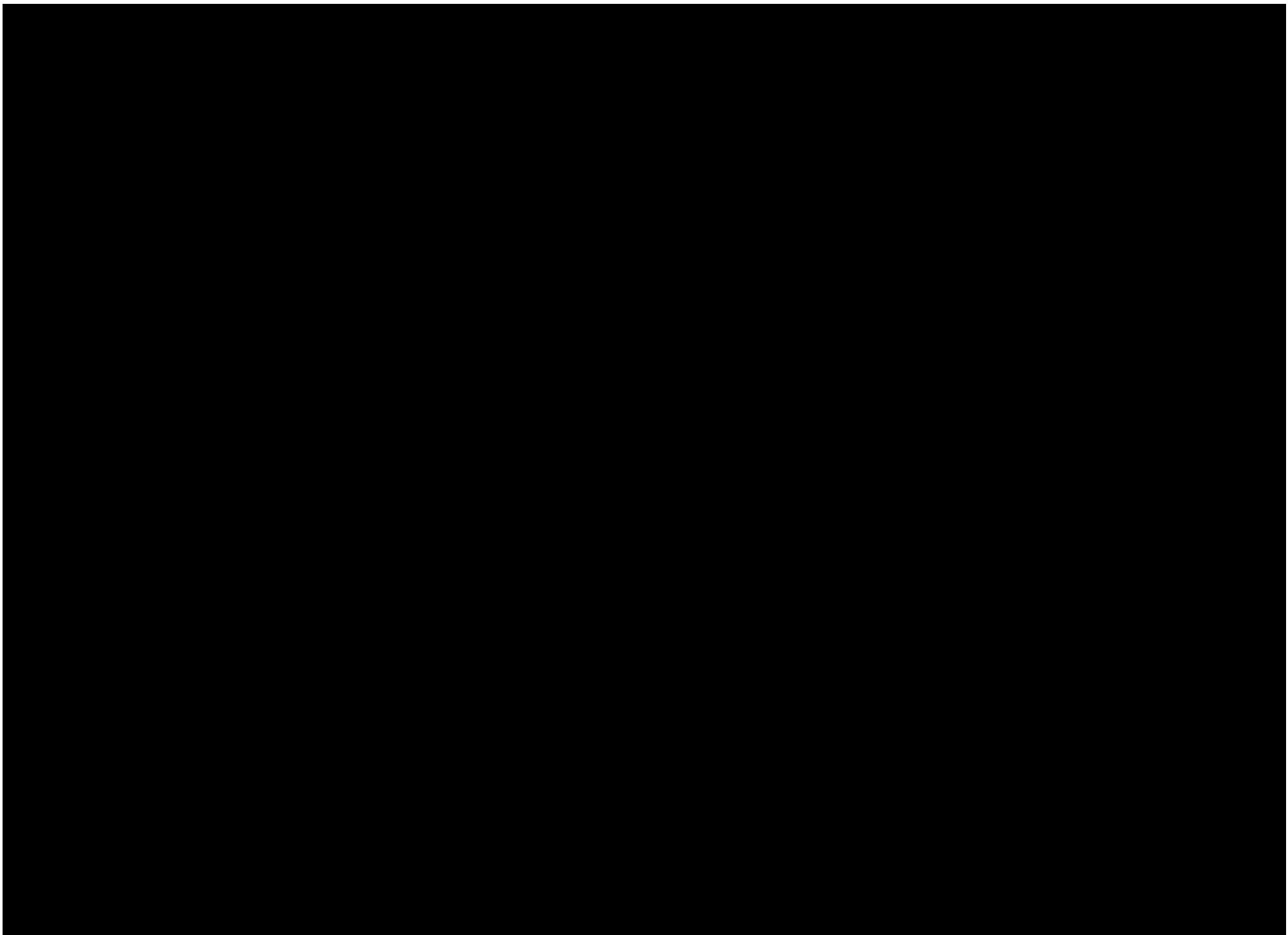
# BONUS

Can use time-travel debugging!











*It's hacking time!*

[source](#)

[more examples](#)

# Q & A

[github.com/grammarly/focal](https://github.com/grammarly/focal)



npm i -S @grammarly/focal

A photograph of a night sky over a city. In the foreground, a tall utility pole stands vertically, with multiple wires radiating outwards towards the horizon. The sky is a deep, dark blue at the top, transitioning into a vibrant orange and yellow glow near the horizon. In the distance, a large suspension bridge, similar to the Golden Gate Bridge, is visible across a body of water. The city below is lit up with numerous streetlights and building lights, creating a glowing urban landscape.

WE'RE HIRING!

grammarly.com/jobs