# Week 09: State management in React, Auth

Filip Kaštovský

## Outline

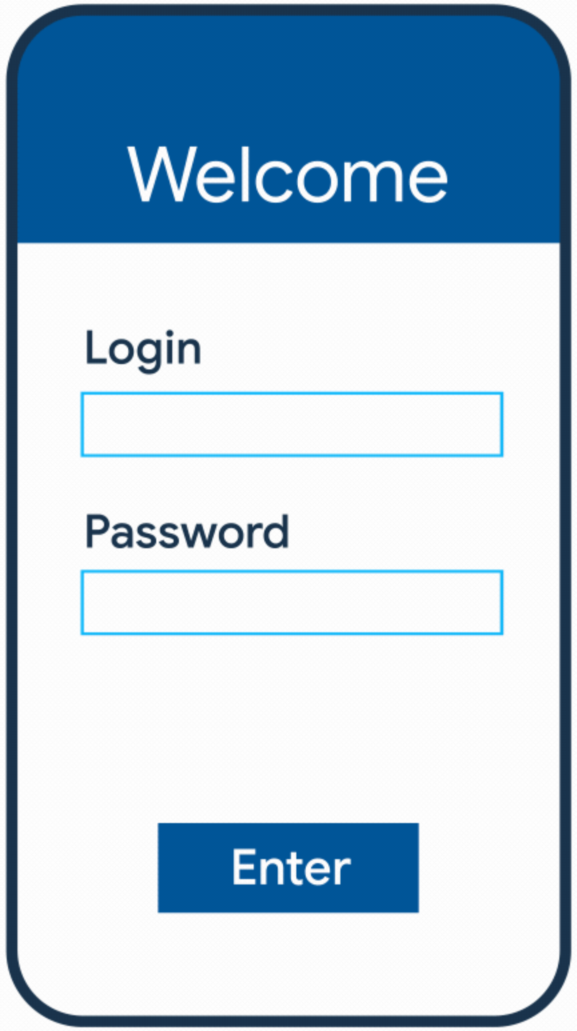
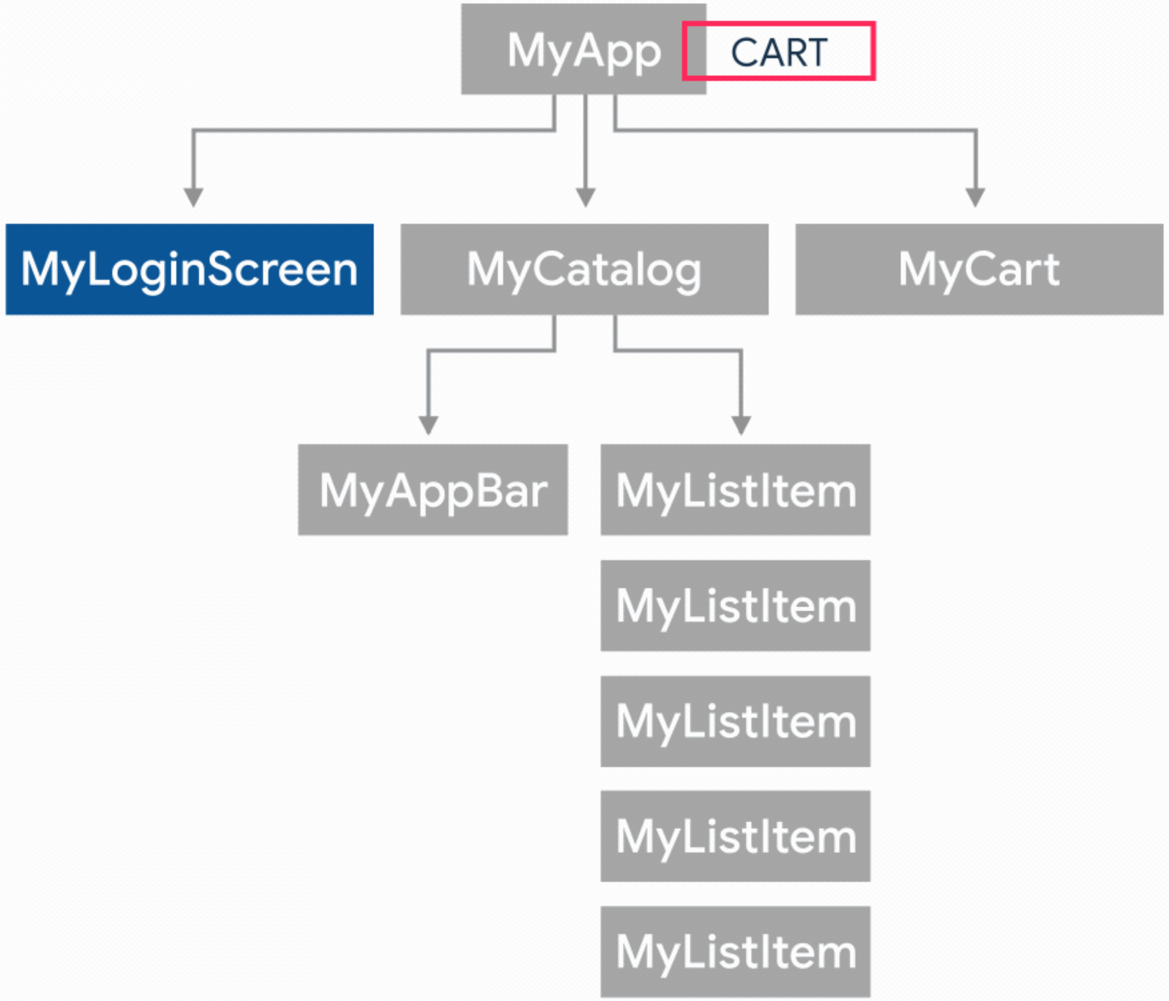
- What is state management? Why do we need it?
- History of state management in React
- State segregation
- Modern state management

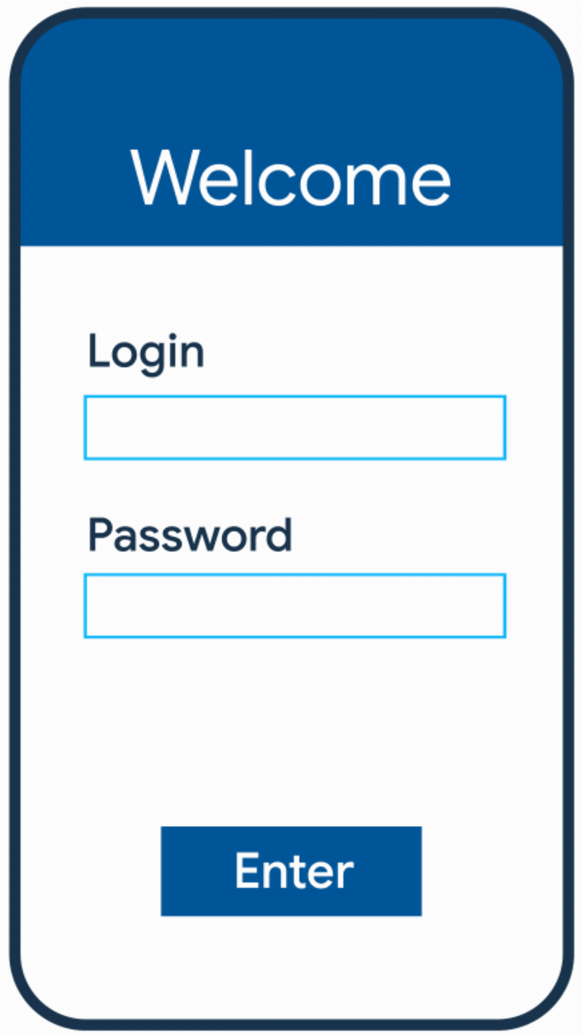
## What is state management? Why do we need it?

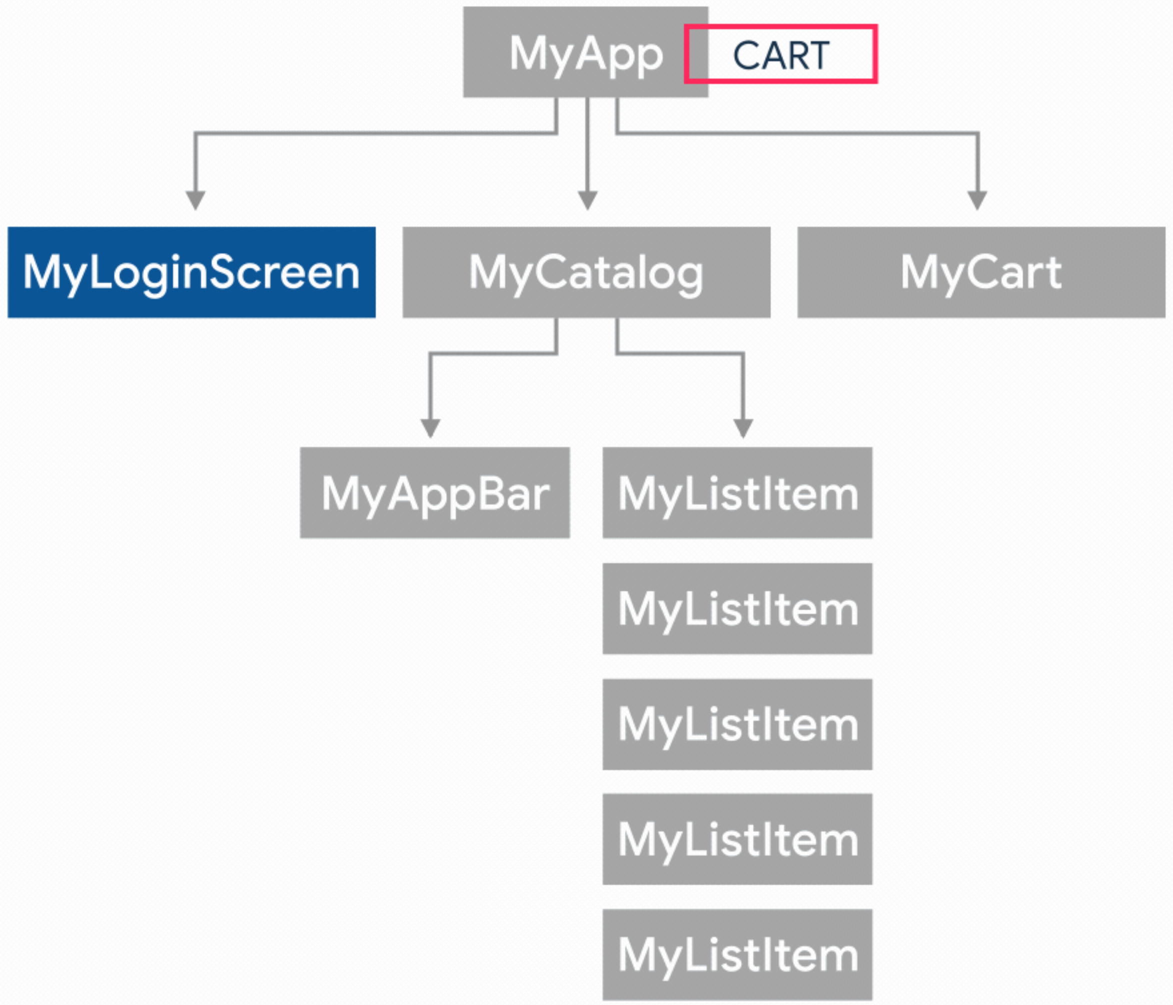
## We don't!*

## What is state management? Why do we need it?

- A way to manage the state of your application (duh)
- State is data that is used by your application
- State management is needed because:
    - State is shared between components
    - State is updated by multiple components
    - State is updated by external sources (API, user input)


Welcome
Login
Password
Enter

MyApp
CART
MyLoginScreen
MyCatalog
MyCart
MyAppBar
MyListItem
MyListItem
MyListItem
MyListItem
MyListItem

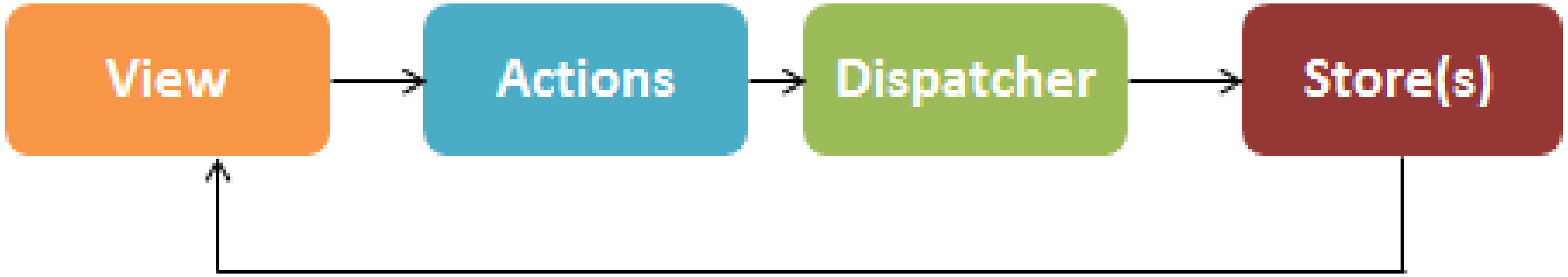

Welcome
Login
Password
Enter


MyApp
CART
MyLoginScreen
MyCatalog
MyCart
MyAppBar
MyListItem
MyListItem
MyListItem
MyListItem
MyListItem

## History of state management in React

## Pre-redux era (< 2016)

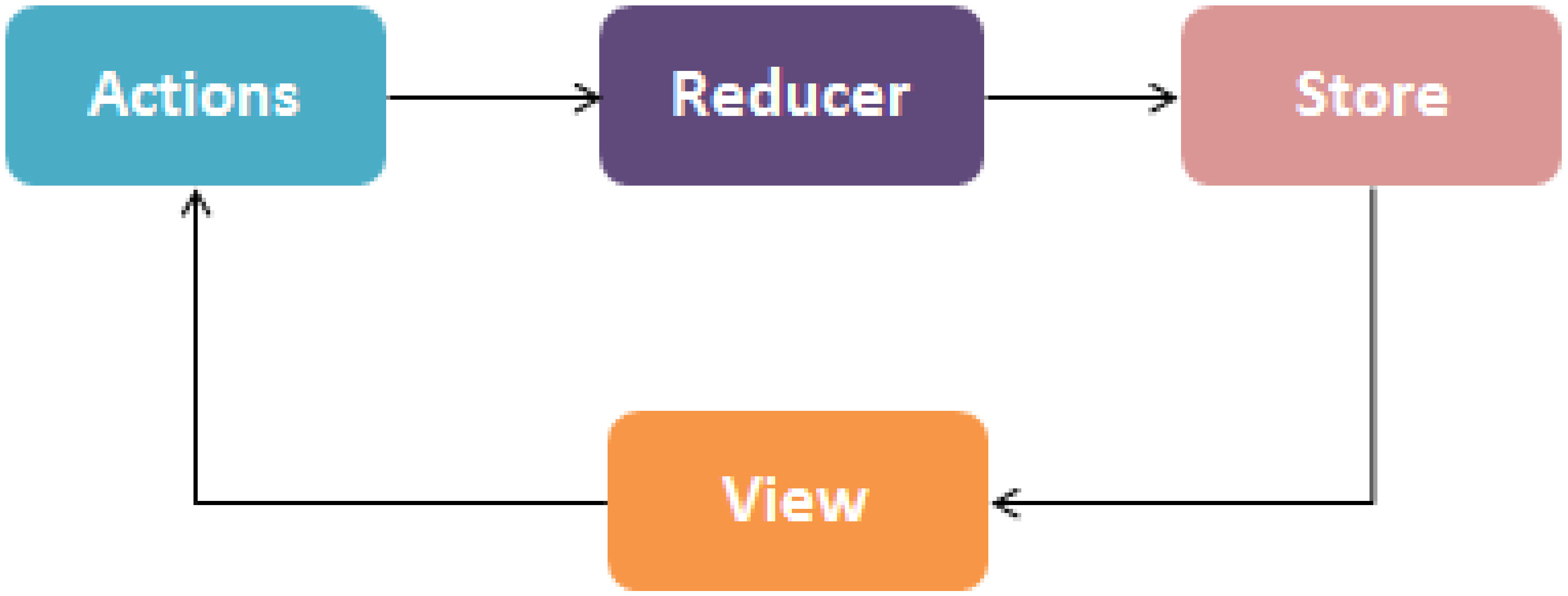
- React's self contained components were an unexplored concept
- Nobody knew what they were doing
- `Lifting state up` was a common way to share state between components
  - `God` components
  - Prop drilling
- Emerging patterns:
  - `Flux` architecture



Flux Architecture

## Redux (2016 - 2020)

- Created by Dan Abramov and Andrew Clark
- A simplification of the `Flux` architecture, Redux introduces a single store that is the source of truth for the entire application
- State is immutable and can only be updated by dispatching actions
- Actions are processed by reducers, which update the state



Redux Architecture

## Redux (2016 - 2020)

```
type Action =
  | { type: "increment" }
  | { type: "decrement" }
  | { type: "set"; payload: number };

type State = { count: number };

function counterReducer(state = State, action: Action) {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    case "set":
      return { count: action.payload };
    default:
      return state;
  }
}
```

## Redux (2016 - 2020)

- Reducers cannot be async -> a new layer before reducers is introduced: redux middleware
  - in an era without async/await, async stuff was painful
  - `redux-thunk`, `redux-saga`, `redux-observable`
- A lot of boilerplate
- Everything is in one place (store)
- Immutable updates for nested objects
- Heavy!
- Fundamentally changes how you write your app
- Large ecosystem of libraries and tools for almost everything

## React Context API (> 2019)

- React's built-in state management solution
- Grew to popularity with hooks
- Declare a context and insert it into the component tree:

```
const MyContext = React.createContext(defaultValue);
```

```
<MyContext.Provider value={value}>
  <MyComponent />
</MyContext.Provider>
```

- Anything inside of `value` can be accessed via a `useContext(MyContext)` hook

## React Context API (> 2019)

- You can put `useState` values and functions in the context, sharing them between components

```
const [state, setState] = useState(initialState);
const value = useMemo(() => ({ state, setState }), [state]);
```

```
<MyContext.Provider value={value}>
  <MyComponent />
</MyContext.Provider>
```

> But, remember how `react` re-renders components?

> **The above is a very bad idea for global state management**

## Sidetrack: `useReducer`

- `useReducer` is a hook that is similar to `useState`, but it allows you to manage more complex state
- uses the same reducer pattern as `redux`

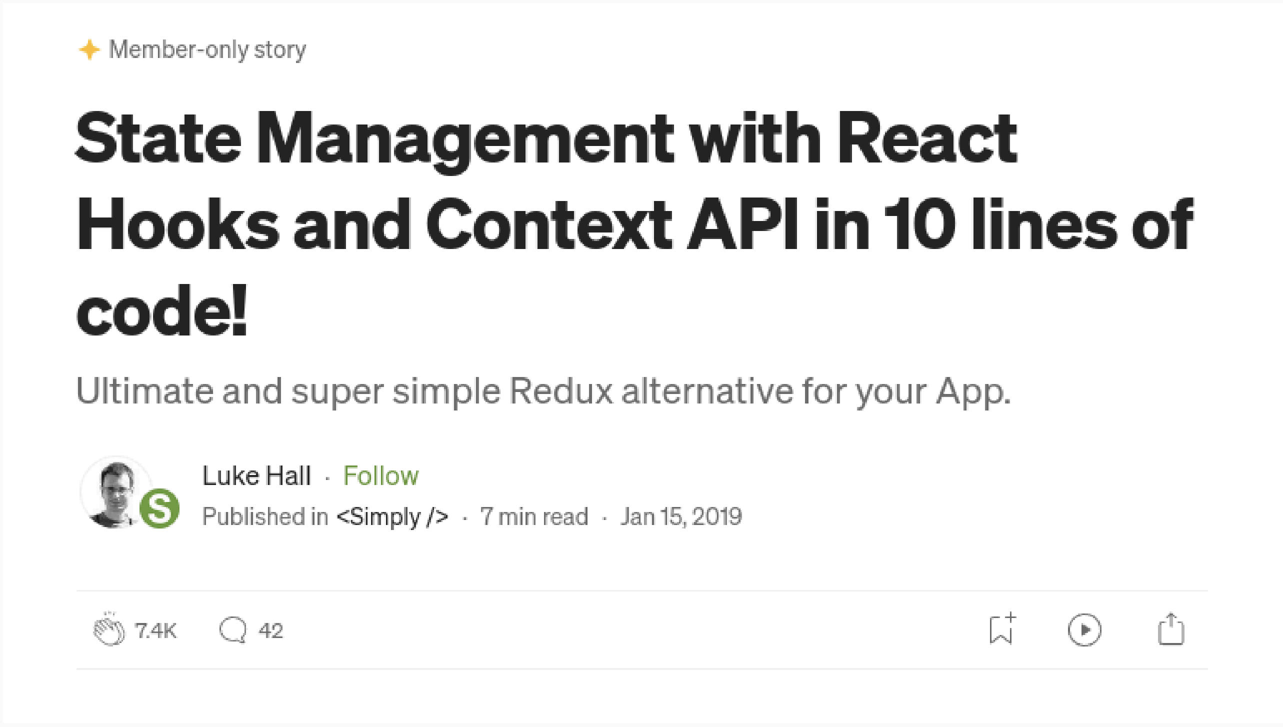
```
const [state, dispatch] = useReducer(reducer, initialState);
```

```
const reducer = (state: State, action: Action) => {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    case "set":
      return { count: action.payload };
    default:
      return state;
  }
};
```

## Antipattern: Poor mans redux

- You can combine `useReducer` and `useContext` to create a poor mans redux in like 10 lines of code

> **DO not do this**, this is not a replacement for redux (or any other state management solution)



Member-only story

State Management with React Hooks and Context API in 10 lines of code!

Ultimate and super simple Redux alternative for your App.

Luke Hall · Follow

Published in <Simply /> · 7 min read · Jan 15, 2019

7.4K 42

## Change of perspective: State segregation

- Rather then treating all of our application state as `global` and one big pile of `data` handled by a generic manager, use specialized tools for specialized tasks
- There is no need to reinvent the wheel!
- Form data? Use `react-hook-form`
- API? Use `Tanstack Query`
- Routing? Use `react-router`
- Local state? Use `useState` / `useReducer`
- Local state across multiple components? Use `useContext`
- ???

> What else do we need to track of in an app?

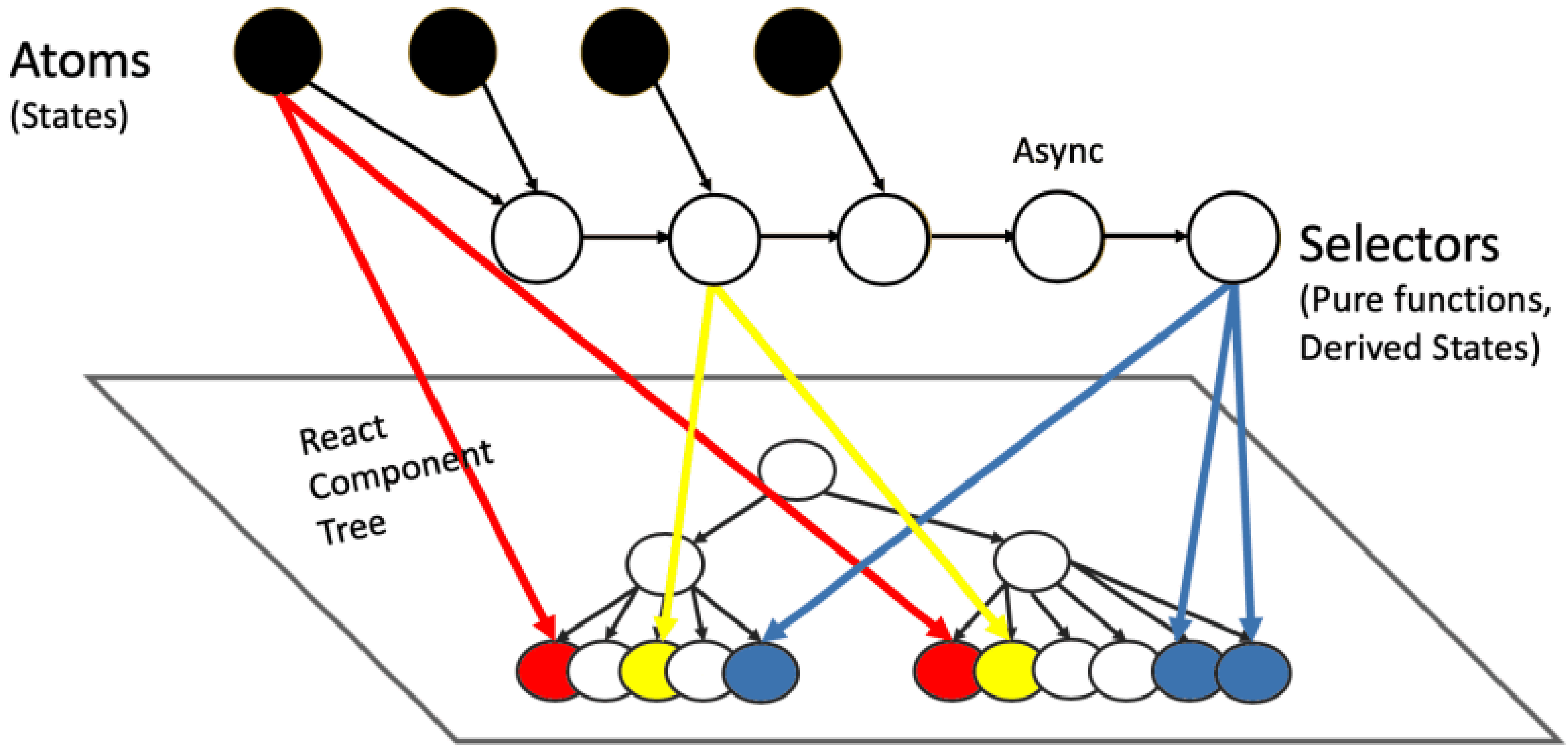
## State segregation

- There is nothing left! We have covered all of the state management needs of our application.
- Almost... (theme, current user, etc.)
- Most of the time, you will not need a global state management solution anymore

## Modern state management

- Ideal state is:
    - handle all state updates outside of React (it is bad at it, wants to re-render everything)
    - only notify and update components that are interested in the state change

## Composable state (atoms)


Atoms
(States)
Async
Selectors
(Pure functions,
Derived States)
React
Component
Tree

## Composable state (atoms)

- Originally a react core team's idea, now implemented as `Recoil`

```
const countState = atom({
  key: "countState",
  default: 0,
});

function Counter() {
  const [count, setCount] = useRecoilState(countState);
  return <div>{count}</div>;
}
```

- Recoil is a bloated library for what it provides, `jotai` is a much better implementation

## Signals

- Observer pattern for state management, introduced by `solid-js`
- By far the most performant solution for state management!
- Backported to react as `@preact/signals-react`

## Auth

## Outline

- Access control
- AuthN vs AuthZ
- Common auth patterns
- Auth in Express
- Auth on the frontend
- Security considerations

## Access control

- Only allow access to resources to authorized users
- Different users have different permissions

> User level access: only the resource owner can access the resource
> Role-based access control: users are assigned roles, roles have permissions
> Rules, policies, etc...

## AuthN vs AuthZ

- Authentication (AuthN) is the process of verifying the identity of a user
- Authorization (AuthZ) is the process of verifying that the user has the necessary permissions to access a resource
  - Easy to mix up, but they are different things
- AuthN and AuthZ are often handled together (Auth / AA) by an application

## Common auth patterns

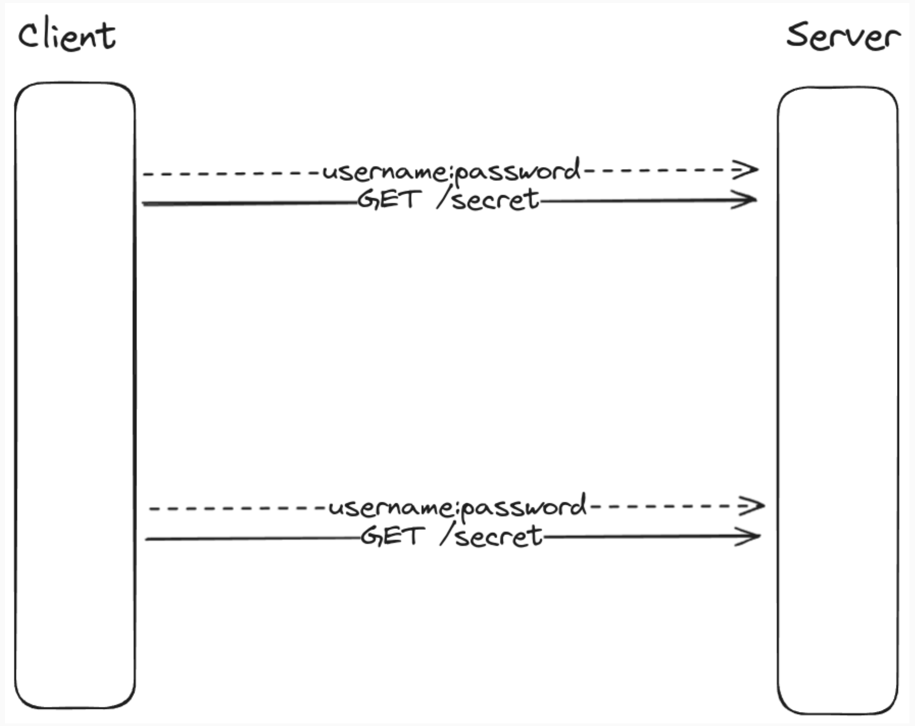
## Basic auth

- Username and password
- Sent in the `Authorization` header
- Base64 encoded

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

- Not secure, use encrypted connections
- Some clients allow encoding the password in the URL

```
smtp://username:password@server:587
```



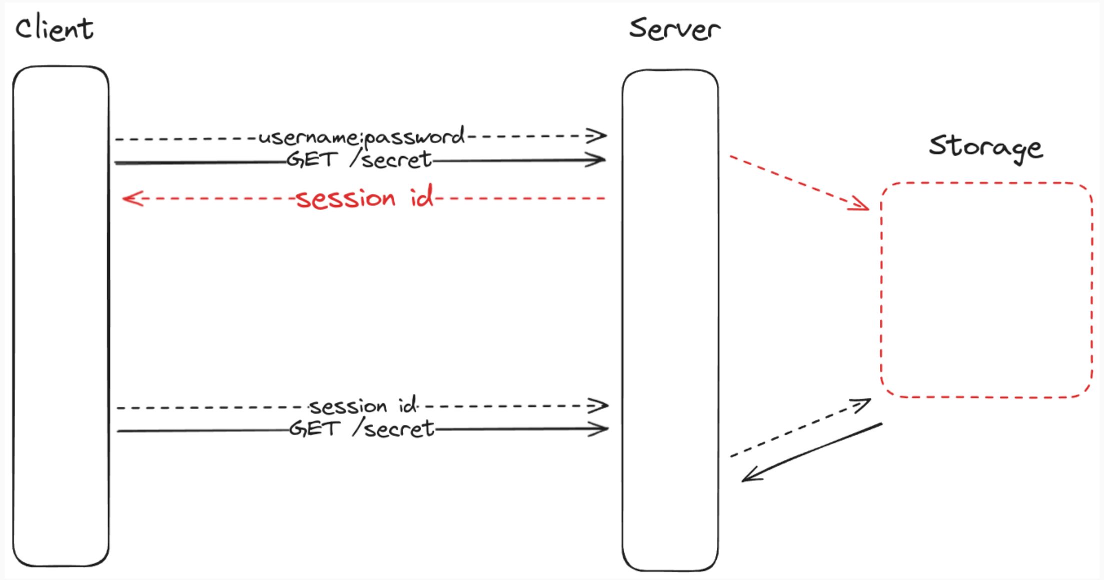
## Token based auth

- A client authenticates with a server and receives a token
- The token is then used for subsequent requests

## SessionID

- Create a unique session ID for each successful authentication
- Store it a database
- Send it to the client
- Client sends it back with each request

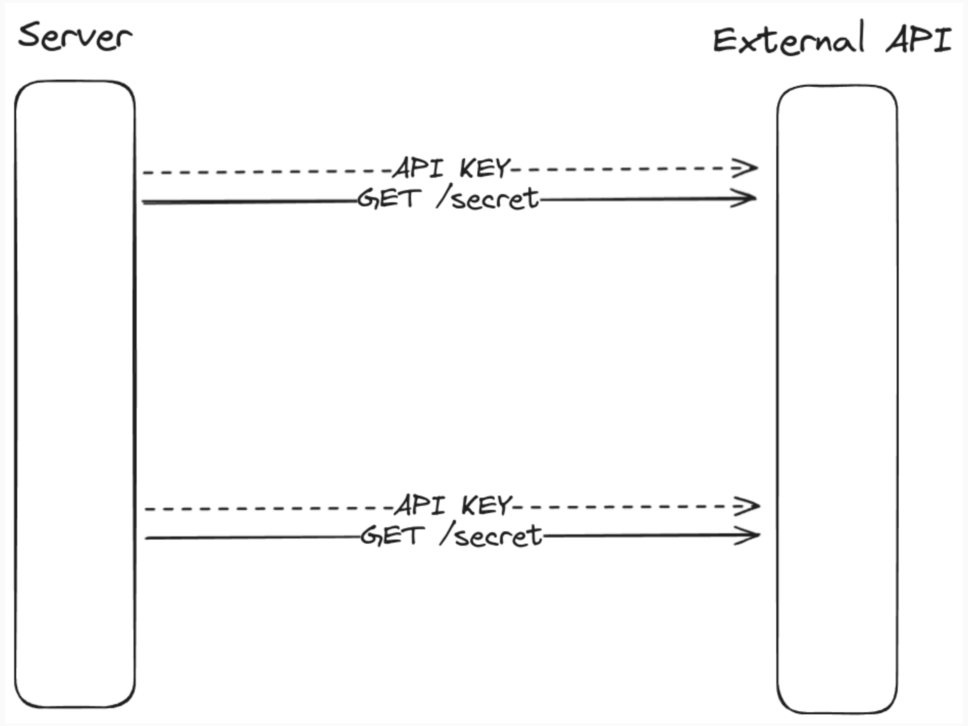
> Overhead of storing sessions 'somewhere', stateful



## API keys

- Used for authenticating against an external API
- Can be of any format

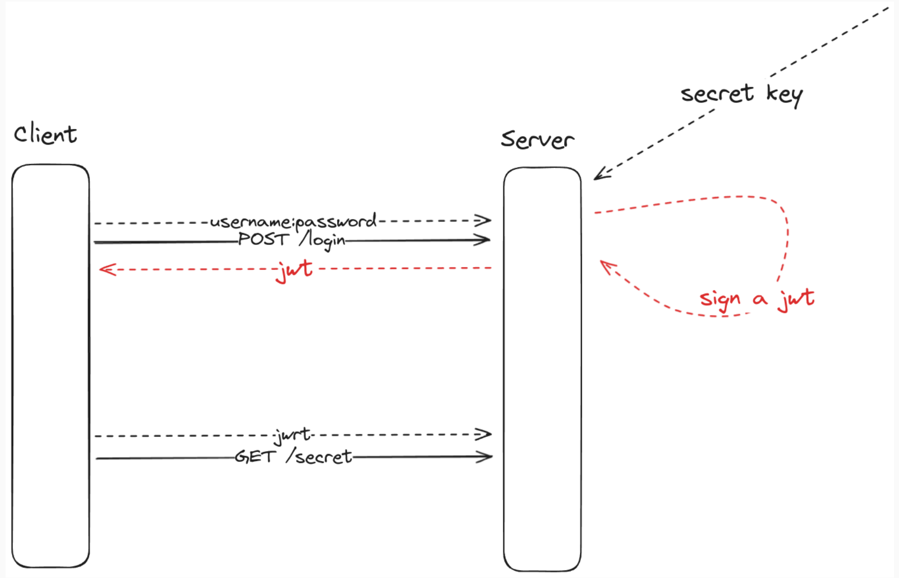
> Generally considered insecure
> ? Principle of least privilege
> Rotate keys often



## JWT

- JSON Web Token
- Self-contained, signed token
- Contains claims (data) about the user
- Cryptographically signed with a given expiry
- [jwt.io](jwt.io)

> Invalidation/revoking



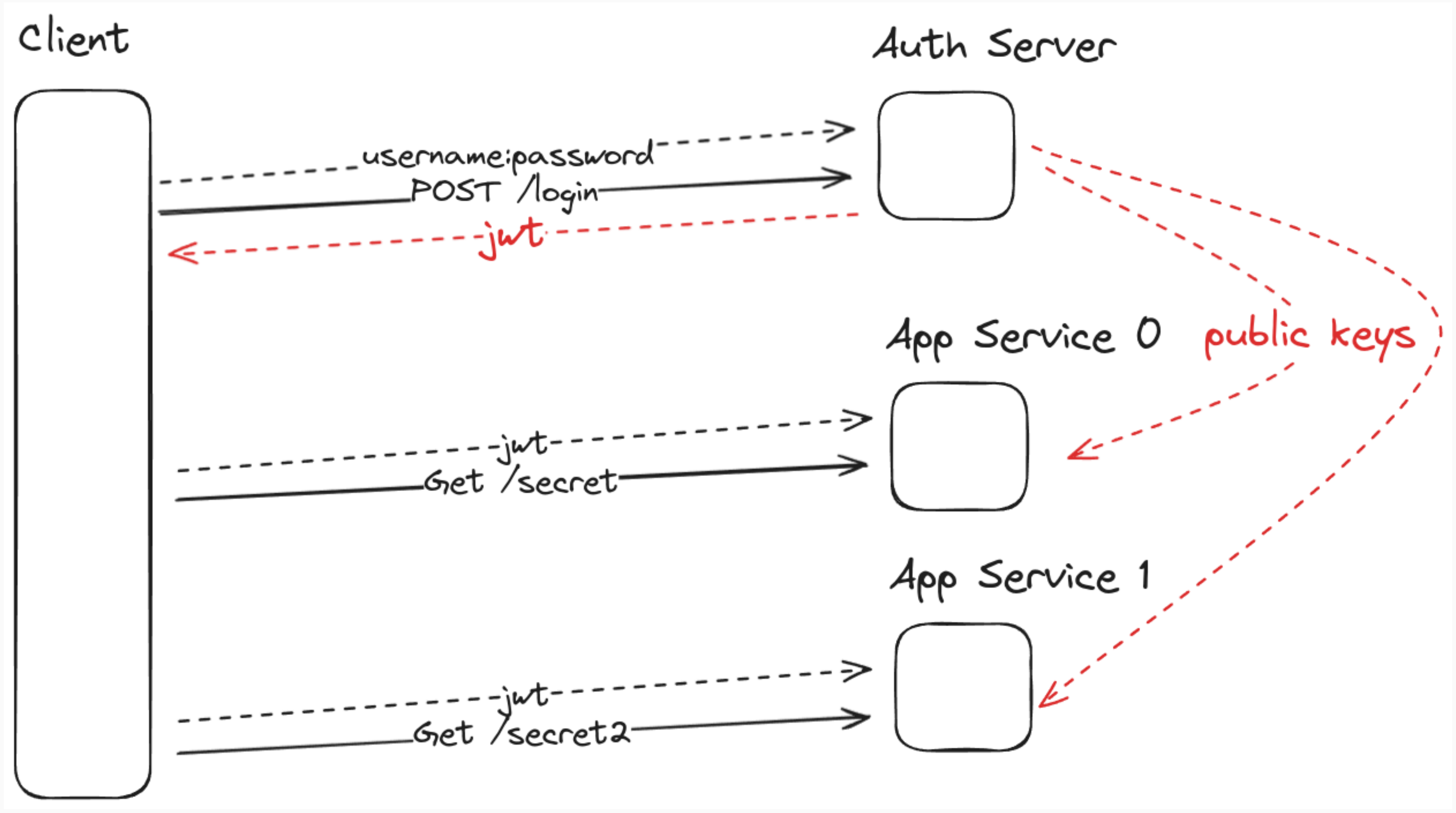
## OAuth2/OpenID Connect

- OAuth2 is an authorization framework
- OpenID Connect is an identity layer on top of OAuth2
- Used for single sign-on (SSO), "social login"
- Allows third-party applications to access resources on behalf of a user

> Requires multiple token exchanges for security
> Complex, but very powerful

## Pattern: Federated auth

- Dedicating a separate service for authentication and authorization
- Allows for more flexibility and scalability
- Allows multiple services to authenticate against the same service!
- In case of a breach, it is more difficult to access all the users' data
- Allows you to use an off the shelf solution, reducing the risk of introducing vulnerabilities


Client
Auth Server
username:password
POST /login
jwt
App Service 0
public keys
jwt
Get /secret
App Service 1
jwt
Get /secret2

## Auth in Express

`passport.js` is a middleware for Express that handles session management

```
app.post("/login/password", passport.authenticate("local"));
```

- `middleware`
- `strategies`
- `sessions`

## Strategies

- A strategy is a way to authenticate a user
- Over 500 strategies available!

For local username/password authentication:

```
npm install passport-local
```

OpenID Connect:

```
 npm install passport-openidconnect
```

## Sessions

Passport also contains connectors for session management

- `express-session`

```
app.use(
  session({
    secret,
  })
);
app.use(passport.session());
```

- internally uses cookies

## But wait? Where do I store the token for token-based auth?

## HTTP cookies

- both sides (client and server) are allowed to read and write them (most of the time)
- Usually the server sets a cookie with the token
- Logout can be done by deleting the cookie

> setting the cookie as `httpOnly` prevents client-side JS from reading it

## Headers

- The token can be sent in the `Authorization` header (just as with basic auth)

```
Authorization: Bearer <token>
```

- This header is then read by the server, but has to be sent by the client.
- Here, the client has to manage the token

> Local storage, session storage...

## Security considerations

## XSS

- Cross-Site Scripting
- Attacker injects malicious scripts into a website and can get access to cookies, session tokens, etc.

## CSRF

- Cross-Site Request Forgery
  - Forces authenticated users to submit a request to a Web application against which they are currently authenticated
- Mitigation: CSRF tokens

## Password storage and validation

- **Never store passwords in plaintext!**
- Always use a secure hashing algorithm (argon2, scrypt, bcrypt)
- Salting

Hashing:

```
const hash = await argon2.hash(..);
```

```
try {
  if (await argon2.verify("<big long hash>", "password")) {
    // password match
  } else {
    // password did not match
  }
} catch (err) {
  // internal failure
}
```

## Rate limiting

- Prevents brute force attacks
- Limits the number of requests a user can make in a given time frame
- Protects underlying infrastructure
- For sensitive endpoints, require the user to solve a challenge (captcha) to prevent automated attacks
- The above doesn't work well anymore (AI)

## Web Application Firewalls

Generally a paid service. Filters out malicious traffic.

WAF uses a set of heuristics and rules to determine if a request is malicious, can prompt the user to solve a challenge if the target is a website.

## OWASP

- Open Web Application Security Project
- A community that produces freely-available articles, methodologies, documentation, tools, and technologies in the field of web application security
- OWASP Top 10

## Thanks for listening!

- Questions?