React 4

CS571: Building User Interfaces

Cole Nelson

Today's Warmup

Clone this repository.

Run `npm install` on starter and solution projects.

Import Week6 and HW6 Postman collections.

Other Announcements

- Cole's office hours on Thursday, 11a-2p this week.
- You will have until Wednesday, March 8th for HW6.
  - Wednesday, March 15th late deadline.
- When it comes to homeworks...
  - Started > !Started
  - Partially Done > Started
  - Done > Partially Done
  - Perfect 🦄
- Midterm exam is *next week*.

Midterm Exam

- Thursday, March 9th 5:45-7:15pm in the Chemistry Building Room S429. You will have **75 minutes**.
  - 24 MC (8 pts)
  - 5 SA (5 pts)
  - 1 LR (2 pts)
- F22 midterm and solution on Canvas.
- Contact me **today** about conflicts!

Last Time...

We covered routing and state sharing.

Routing

```
<BrowserRouter>
  <Routes>
    <Route path="/" element={<Layout />}>
      <Route index element={<Home />} />
      <Route path="about-us" element={<AboutUs />} />
      <Route path="other-info" element={<OtherInfo />} />
      <Route path="*" element={<Home />} />
    </Route>
  </Routes>
</BrowserRouter>
```

Browser `outlet`

`<Outlet/>` shows the component returned by the child route! e.g. in `Layout` we may see...

```
function Layout() {
  return (
    <>
      <Navbar bg="dark" variant="dark">
        { /* Some navigation links...*/ }
      </Navbar>
      <Outlet />
    </>
);
}
```

State Management

How do we talk back to our parent? How do siblings talk to each other?

- Passing callbacks
- `useContext`
- `cookie`, `sessionStorage`, and `localStorage`
- Third-party libraries
  - Redux, Recoil, MobX, XState

How do we persist data...

...permanently? and share with others?

What will we learn today?

- How can we store data more persistently?
- How do we work with "complex" APIs?
- How to keep a secret? 🤫
- What is "memoization"?

Working with Complex APIs

Beyond GETting data...

Scenario

You are building a database system. What operations should you allow a developer to perform?

Scenario

You are building a database system. What operations should you allow a developer to perform?

1. **C**reate data.
2. **R**ead data.
3. **U**pdate data.
4. **D**elete data.

CRUD Operations via HTTP

CRUD Operation	HTTP Operation
Create	POST
Read	GET
Update	PUT
Delete	DELETE

HTTP Recap

Data is transmitted by requests and responses that allow us to create (POST), read (GET), update (PUT), and delete (DELETE) data!

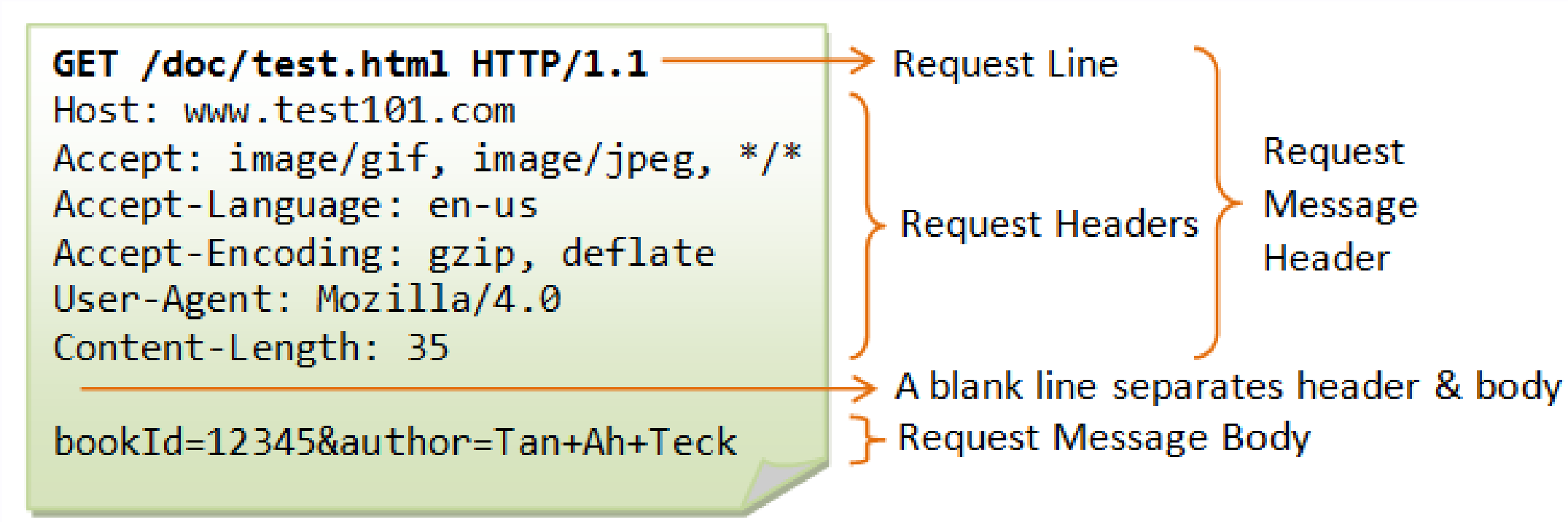


Image Source

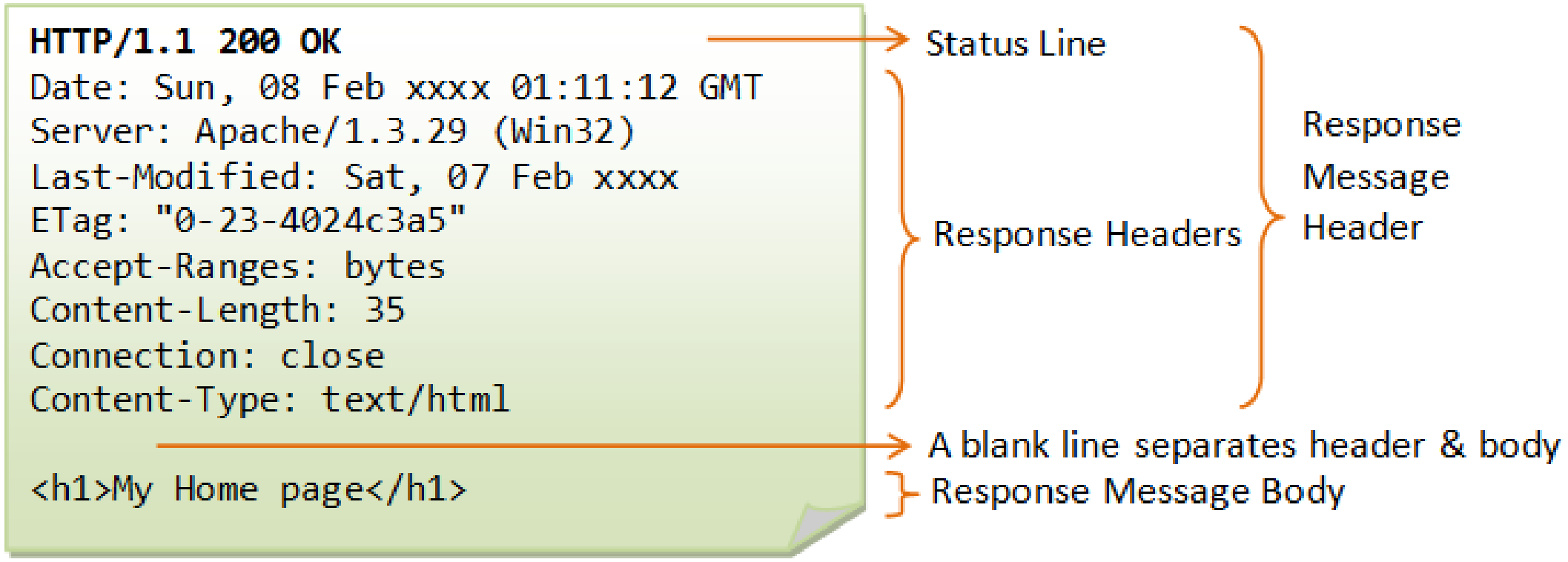


Image Source

HTTP Recap

An HTTP request may have *path* and *query* parameters

```
https://www.example.com/packers/AaronRodgers/stats?all=true&since=2010
```

Here, `AaronRodgers` is a *path* parameter while `all` and `since` are *query* parameters.

HTTP Recap

HTTP requests (specifically `PUT` and `POST`) may also have a *request body*. This is located below the headers.

```
{
    "title": "Hello world!",
    "content": "abc123",
}
```

For a JSON body, we need an additional HTTP header.

`Content-Type: application/json`

HTTP Status Codes

HTTP Code	Response Type
100s	Informational
200s	Successful
300s	Redirection
400s	Client Error
500s	Server Error

HTTP Specific Status Codes

HTTP Code	Response
200	OK
304	Not Modified
400	Bad Request
401	Unauthorized
404	Not Found
409	Conflict
413	Request Entity Too Large
500	Internal Server Error

What is this "HTTPS" I hear about?

The "secure" version of HTTP.

Same thing as the HTTP protocol with end-to-end encryption. We use HTTPS for our API.

Week 6 API

Use Postman to `POST` a ticket.

Fetching w/ `POST`, `PUT`, and `DELETE`

`fetch` can do a lot more than just retrieving data.

- specify request method
- specify request headers
- specify request body
- inspect response status
- inspect response headers
- inspect response body
- ...and so much more!

```
fetch("https://example.com/create-content", {
    method: "POST",
    headers: {
        "Content-Type": "application/json" // must include this header
    },
    body: JSON.stringify({ // must stringify
        content: "Hello World!"
    })
}).then(res => {
    if (res.status === 409) {
        alert("This content already exists!")
    }
    return res.json();
}).then(json => {
    if (json.msg) {
        alert(json.msg)
    }
});
```

Your Turn!

Create *controlled* or *uncontrolled* components for a user to type in their title and content.

When the user clicks "Create Ticket", `POST` the ticket to the API and reload the tickets on the screen.

Credentialed Requests

Secrets! Secrets!

Is there anything **special** about requests for logging in? Kind of! We must include credentials.

It varies from system to system, but typically we POST a username and password in the request body to an authentication endpoint and recieve a cookie.

e.g. `POST /register` or `POST /login`

Secrets! Secrets!

These endpoints return a session for the user, in HW6 this is in the form of a cookie containing a JavaScript Web Token (JWT).

This is a temporary, all-access token for authenticating with the API. It is used in lieu of a username and password. Why might we do this?

Secrets! Secrets!

A session is typically stored in an http-only cookie.

An HTTP-Only cookie is not accessible by JavaScript.

Why might we want this?

Secrets! Secrets!

Because `cs571.org` is a different domain than `localhost`, we must also *explictly* include credentials to every endpoint affecting authorization...

This includes logging in, logging out, and posting!

```
fetch("https://example.com/api/submit", {
  method: "POST",
  credentials: "include",
  // ...
```

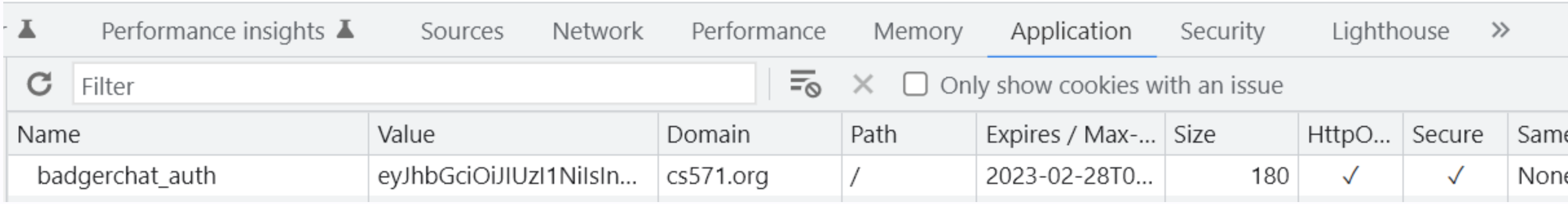
```
fetch("https://example.com/create-content", {
    method: "POST",
    credentials: "include", // add this to requests related to cookies!
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify({
        content: "Hello World!"
    })
}).then(res => {
    if (res.status === 409) {
        alert("This content already exists!")
    }
    return res.json();
}).then(json => {
    if (json.msg) {
        alert(json.msg)
    }
});
```

Reading the Docs

Go over the HW6 API documentation.

Secrets! Secrets!

What's the benefit? The browser handles all things authentication! 🥳🎉



HW6 Demo

Go over the HW6 API documentation and use Postman.

Memoization

Not memorization!

Memoization

Storing the result so you can use it next time instead of calculating the same thing again and again

what the frik is: memoization

`useCallback` to memoize functions

`useMemo` to memoize calculated values

`memo` to memoize components

`useCallback` Hook

Consider the following functional component...

```
function MyApp() {
  const myComplicatedFunction = () => {
    // ...
  }

  return <>
    <button onClick={myComplicatedFunction}>Click Me</button>
  </>
}
```

How many times do we *create* the function `myComplicatedFunction`? We do on *every render*!

`useCallback` Hook

`useCallback` is used to 'memoize' a callback function.

```
function MyApp() {
  const myComplicatedFunction = useCallback(() => {
    // ...
  }, []);

  return <>
    <button onClick={myComplicatedFunction}>Click Me</button>
  </>
}
```

Takes a callback function to 'memoize' and an optional list of dependencies (e.g. when to re-'memoize').

`useMemo` Hook

Same thing as `useCallback`, except memoizes the *value* of a *callback* rather than the *callback* itself.

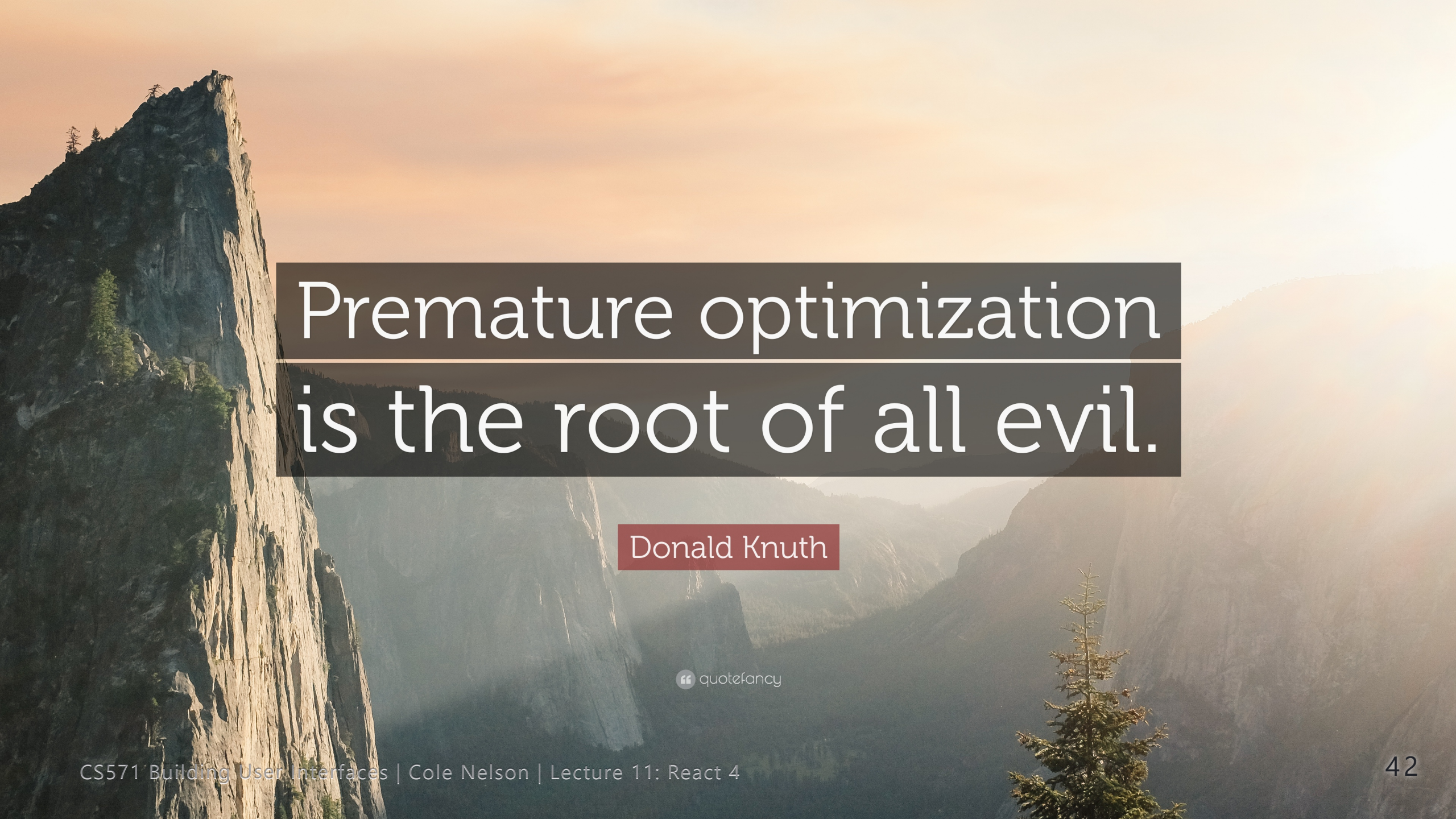
```
function MyApp() {
  const myComplicatedValue = useMemo(() => { /* Some complex call */}, []);

  return <>
    <p>{myComplicatedValue}</p>
  </>
}
```

`memo` -ized Components

Used for creating *purely functional* components. Given the same props, the function renders the same output.

```
//                      v--- Name of functional component!
export default memo(GroceryList, (prevProps, nextProps) => {
  return prevProps.apples === nextProps.apples &&
    prevProps.bananas === nextProps.bananas &&
    prevProps.coconuts === nextProps.coconuts;
})
```

See StackBlitz for `useCallback`, `useMemo`, and `memo`


Premature optimization is the root of all evil.
Donald Knuth
quotefancy

A Plea for Lean Software

Niklaus Wirth
ETH Zürich

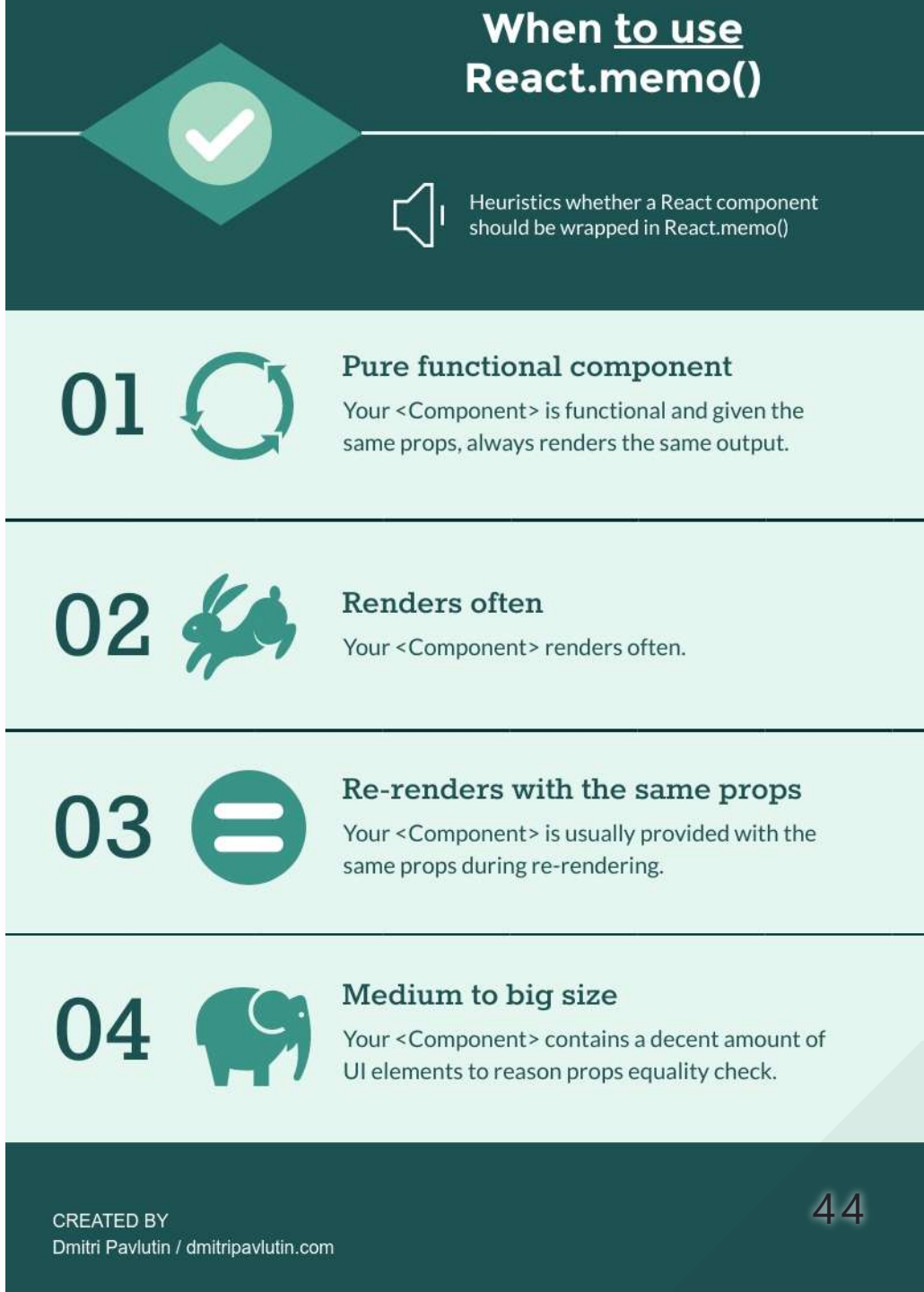
Memory requirements of today's workstations typically jump substantially—from several to many megabytes—whenever there's a new software release. When demand surpasses capacity, it's time to buy add-on memory. When the system has no more extensibility, it's time to buy a new, more powerful workstation. Do increased performance and functionality keep pace with the increased demand for resources? Mostly the answer is no.

About 25 years ago, an interactive text editor could be designed with as little as 8,000 bytes of storage. (Modern program editors request 100 times that much!) An operating system had to manage with 8,000 bytes, and a compiler had to fit into 32 Kbytes, whereas their modern descendants require megabytes. Has all this inflated software become any faster? On the contrary. Were it not for a thousand times faster hardware, modern software would be utterly unusable.

Finding a Balance

1. Given the same input, renders the same output.
2. Is rendered often.
3. Does not change often.
4. Is of substantial size.

Dmitri Pavlutin Blog Post



Badger Bingo

Cumulative example, see StackBlitz.

What did we learn today?

- How can we store data more persistently?
- How do we work with "complex" APIs?
- How to keep a secret? 🤫
- What is "memoization"?

Questions?