

Middleware in Express.js

Route function:

```
(req, res, next) => {  
    // ...  
}
```

Can handle request fully, or call 'next()'

Multiple handlers on the route

```
app.get(  
  '/user/:id',  
  (req, res, next) => {  
    console.log('Request URL:', req.originalUrl);  
    next();  
  },  
  (req, res, next) => {  
    res.send('Hello world');  
  }  
);
```

Multiple routes

```
app.get(  
  '/user/:id',  
  (req, res, next) => {  
    console.log('ID:', req.params.id);  
    next();  
  },  
  (req, res) => {  
    res.send('User Info');  
  }  
);
```

// handler for the /user/:id path, which prints the user ID

```
app.get('/user/:id', (req, res) => {  
  res.send(req.params.id);  
});
```

First route never passes control to the second

Multiple routes with skip

```
app.get(
  '/user/:id',
  (req, res, next) => {
    // if the user ID is 0, skip to the next route
    if (req.params.id === '0') next('route');
  },
  (req, res) => {
    res.send('regular');
  }
);
```

```
// handler for the /user/:id path, which sends a special response
app.get('/user/:id', (req, res) => {
  res.send('special');
});
```

Handler array

```
app.get(  
  '/user/:id',  
  [(req, res, next) => {  
    console.log('ID:', req.params.id);  
    next();  
  },  
  (req, res) => {  
    res.send('User Info');  
  }  
]  
);
```

Can be array variable

Adding middleware

```
app.use((req, res, next) => {  
    console.log('Time:', Date.now());  
    next();  
});
```

Handles all requests

Adding subtree middleware

```
app.use("/path", (req, res, next) => {  
    console.log('Time:', Date.now());  
    next();  
});
```

Handles requests to paths starting with /path

Error middleware

```
app.use((err, req, res, next) => {  
  console.error(err.stack);  
  res.status(500).send('Something broke!');  
});
```

Note 4 parameters: cannot skip `next` here

Express.js middleware

`express.static` serves static assets such as HTML files, images, and so on.

`express.json` parses incoming requests with JSON payloads.

`express.text` parses incoming requests with text payloads.

`express.urlencoded` parses incoming requests with URL-encoded payloads.

Third party: cookies, favicon, virtual host, etc.

Cookies: set

```
var cookie = require("cookie");

app.get("/", (req, res) => {

  res.setHeader(
    "Set-Cookie",
    cookie.stringifySetCookie({
      name: "name",
      value: String(query.name),
      httpOnly: true,
      maxAge: 60 * 60 * 24 * 7, // 1 week
    }));
})
```

Cookies: get

```
var cookieParser = require('cookie-parser');  
  
app.use(cookieParser());  
  
app.get('/', function (req, res) {  
  console.log('Cookies: ', req.cookies);  
});
```

Web Programming

Lecture 12

Authentication, authorization and security

Josue Obregon



jobregon@seoultech.ac.kr

Seoul National University of Science and Technology
Information Technology Management

► [Lecture slides index](#)

May 17, 2026

Security in web applications and APIs

- APIs concentrate sensitive functionality:
 - Expose business logic through endpoints
 - Transmit user data, transactions, and authorization decisions
 - Attackers target APIs because they are often less protected than user interfaces:
 - Weak input validation
 - Missing authentication or authorization checks
 - Predictable resource identifiers
 - The Open Web Application Security Project (OWASP) publishes reference lists of the most critical vulnerabilities:
 - **Web Applications Top 10**
 - **API Security Top 10**
-

Authentication and authorization

- **Authentication** verifies *who* a user is.
 - Establishes identity through credentials such as username/password, tokens, or biometrics
 - May include Multi-Factor Authentication (MFA) for stronger assurance
- **Authorization** determines *what* an authenticated user is allowed to do.
 - Based on roles, permissions, or scopes
 - Applied independently to each protected resource or action
- Key distinction:
 - Authentication: “You are user12345.”
 - Authorization: “user12345 can view flights but cannot create new flights.”
- Both must be enforced on the server. Client-side checks improve UX but never guarantee security.

Authentication factors

- Authentication relies on the user presenting an identifier together with a credential.
 - The server verifies the binding between identifier and credential, then decides whether to grant access.
- Credentials fall into three categories (**factors**):
 - Something the user **knows**: password, PIN, security question
 - Something the user **has**: phone, hardware token, smart card
 - Something the user **is**: fingerprint, face, voice (biometrics)
- **Multi-Factor Authentication (MFA)** combines two or more categories.
 - Example: password (knows) plus a one-time code sent to a phone (has)
 - Significantly raises the cost of credential theft
- Common authentication mechanisms in web applications:
 - Username and password
 - Session-based authentication (server-side state)
 - Token-based authentication (e.g., JWT), common in RESTful APIs

Authorization and Role-Based Access Control

- Authorization decisions answer questions such as:
 - Can this user read this resource?
 - Can this user modify or delete this resource?
 - Can this user reach this route at all?
- **Role-Based Access Control (RBAC)** is the most common authorization model:
 - Permissions are grouped into roles (e.g., **admin**, **editor**, **viewer**)
 - Users are assigned one or more roles
 - Each protected action is allowed or denied based on the user's roles
- Implementation principles:
 - Apply checks on the server for every protected route and action
 - Deny by default, allow only what is explicitly granted
 - Validate resource ownership when a user accesses their own data (e.g., a user can only delete their own posts)
- Missing authorization checks are the root cause of *Broken Access Control* vulnerabilities.

Password hashing

- Passwords must never be stored in plain text.
 - A database breach would immediately expose all user credentials
 - Users often reuse passwords across services, amplifying the impact
- **Hashing** transforms a password into a fixed-length value using a one-way function.
 - The original password cannot be recovered from the hash
 - On login, the server hashes the submitted password and compares it to the stored hash
- A good password hash function for storage is **slow and adaptive**:
 - **bcrypt**, **Argon2**, **scrypt** add a configurable cost factor
 - Slowness limits brute-force attempts
- General-purpose hashes (MD5, SHA1, SHA256) are **not suitable** for password storage: they are fast by design.

More on Password hashing

- **Salting:** a unique random value added to each password before hashing.
 - Prevents precomputed (rainbow table) attacks
 - Ensures identical passwords produce different hashes
 - bcrypt embeds the salt in the resulting string
- The code below shows an example using the npm package **bcrypt**, a popular package used for password hashing

```
1 const bcrypt = require('bcrypt');
2 const hash = await bcrypt.hash('Hello World', 10);
3 console.log(hash);
4 // $2b$10$9Nub0LuSmYa5v0Po.h.9D0h09Cnor/nXSPpakMK2N6D/ZKZdAw1yq
5
6 const ok = await bcrypt.compare('Hello World', hash);
7 // true
```

JSON Web Tokens (JWT)

- A JWT is a compact, URL-safe string that encodes information (**claims**) and is **signed** with a secret or private key.
- Purpose in authentication:
 - After successful login, the server issues a JWT to the client
 - The client includes the JWT in subsequent requests
 - The server verifies the signature to trust the claims, without storing session state
- Properties relevant to API design:
 - **Stateless**: the server does not keep session records, simplifying horizontal scaling
 - **Self-contained**: claims (user id, roles, expiration) travel with the request
 - **Verifiable**: any modification invalidates the signature and is rejected
- Trade-offs:
 - Tokens cannot be revoked instantly without additional mechanisms
 - Sensitive data should not be placed in the payload — it is encoded, not encrypted



JWT structure

- A JWT is composed of three parts separated by dots, each base64url-encoded:
 - **Header**: token type and signing algorithm
 - **Payload**: claims describing the subject and metadata
 - Common claims: **iss** (issuer), **sub** (subject), **exp** (expiration), **nbf** (not before), **iat** (issued at), **jti** (token id)
 - Custom claims: user id, roles, scopes
 - **Signature**: produced by signing the encoded header and payload with the secret or private key
- Verification:
 - The server recomputes the signature and compares it to the one in the token
 - A valid signature proves the token was issued by a trusted party and has not been altered

- The payload is readable by anyone holding the token. It must not contain secrets.

Header

```
1 {  
2   "alg": "HS256",  
3   "typ": "JWT"  
4 }
```

Payload

```
1 {  
2   "sub": "1234567890",  
3   "name": "John Doe",  
4   "admin": true,  
5   "iat": 1516239022  
6 }
```

Encoded token

```
1 eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9yJzdv
```

Inspect tokens with the **JWT Debugger**.

JWT authentication flow

- Login phase:
 - The client submits credentials over HTTPS
 - The server validates them and signs a JWT containing the user's identity and roles
 - The token is returned to the client
- Authenticated requests:
 - The client sends the token in the **Authorization: Bearer <token>** header
 - The server verifies the signature and expiration
 - On success, the request proceeds; on failure, the server responds with **401 Unauthorized**
- The server never stores the token; trust comes from the signature.

OAuth 2.0 — delegated authorization

- OAuth 2.0 is an **authorization framework** that lets a user grant a third-party application limited access to their data **without sharing credentials**.
- Used by major providers (Google, Meta, GitHub) to implement “Sign in with...” and API access flows.
- Roles in OAuth 2.0:
 - **Resource Owner**: the user who owns the data
 - **Client**: the application requesting access on the user’s behalf
 - **Authorization Server**: authenticates the user and issues access tokens
 - **Resource Server**: hosts the protected data and validates access tokens
- The client receives an **access token** with a defined scope and expiration; the user’s password is never exposed to the client.
- OAuth is often paired with **OpenID Connect** to add an authentication layer on top of authorization.

Protected routes and middleware

- A **protected route** requires the requester to be authenticated (and possibly authorized) before the route handler executes.
- In Express, this is implemented with **middleware** placed before the route handler:
 - The middleware inspects the request (headers, cookies)
 - It validates the credential (e.g., verifies a JWT)
 - On success, it attaches user information to the request and calls **next()**
 - On failure, it short-circuits the pipeline with **401 Unauthorized** or **403 Forbidden**
- Authentication vs authorization in the pipeline:
 - **Authentication middleware**: confirms identity (valid token)
 - **Authorization middleware**: confirms permission for the specific action (e.g., role check, ownership check)
- Status code convention:
 - **401 Unauthorized**: missing or invalid credentials
 - **403 Forbidden**: authenticated, but not allowed to perform the action

Installing dependencies

```
1 npm install bcrypt jsonwebtoken dotenv
```

- Package roles:
 - **bcrypt**: password hashing and verification
 - **jsonwebtoken**: signing and verifying JWTs
 - **dotenv**: loads configuration values from a **.env** file into **process.env**
- Why **dotenv**:
 - The JWT signing secret must never live inside the source code
 - Anyone with the secret can forge valid tokens for any user
 - Hard-coded secrets get committed to Git, leak through code reviews, end up in public repositories
 - **.env** keeps the secret in a local file that is excluded from version control

Configuration with .env

- Create a **.env** file in the project root:

```
1 JWT_SECRET=replace-with-a-long-random-string
2 JWT_EXPIRES_IN=1h
```

- Add **.env** to **.gitignore** so it is never committed:

```
1 node_modules/
2 .env
```

- Load the file as early as possible in **app.js** (before anything that reads **process.env**):

```
1 require("dotenv").config();
```

- After this call, the values are available as **process.env.JWT_SECRET** and **process.env.JWT_EXPIRES_IN** anywhere in the project.
- Good practice: commit a **.env.example** file with the variable names but no real values, so other developers know what to provide

Auth types: session

After login, the server creates a session and stores it (memory, DB, or cache).

The client receives a session ID, typically in a cookie.

Each request includes the cookie; the server looks up the session.

Characteristics:

- State lives on the server

- Progressive-enhancement-friendly

- Easy to invalidate (delete the session)

- Scales poorly across many servers without shared session storage

- Cookies require CSRF protection

Auth types: token

After login, the server issues a signed token containing user claims.

The client stores the token and sends it in the Authorization header on each request.

The server verifies the signature; no session lookup is needed.

Characteristics:

- Stateless: scales horizontally without shared storage

- Token revocation requires extra mechanisms (blocklists, short expirations)

- Storage in the browser must be chosen carefully (XSS or CSRF exposure)

software studio

model view controller

Daniel Jackson

what does a web app do?

routing

- › translates URLs to actions, and vice versa

presentation

- › uses HTML, CSS to display data

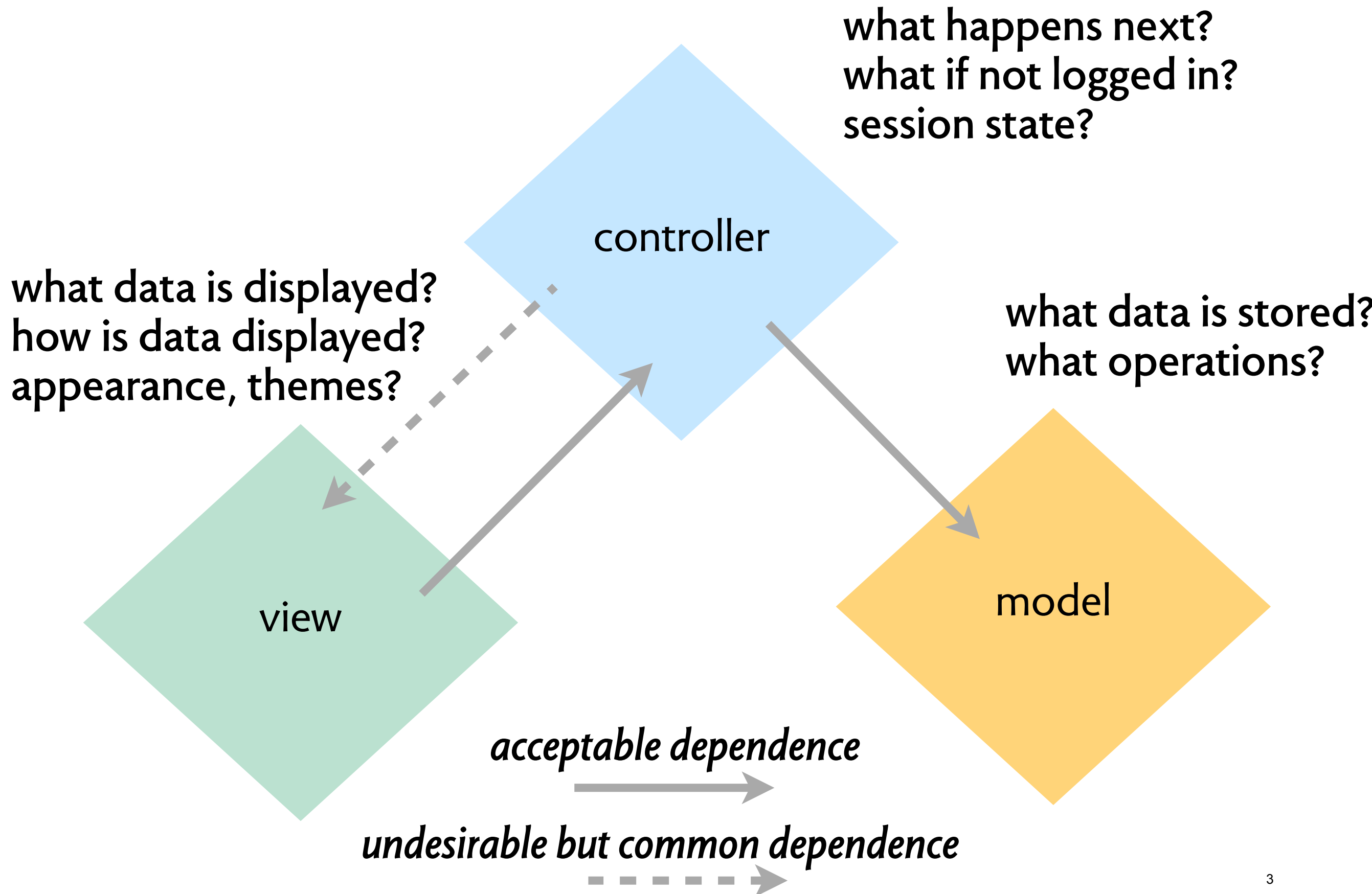
interaction & control

- › determines what user can do next

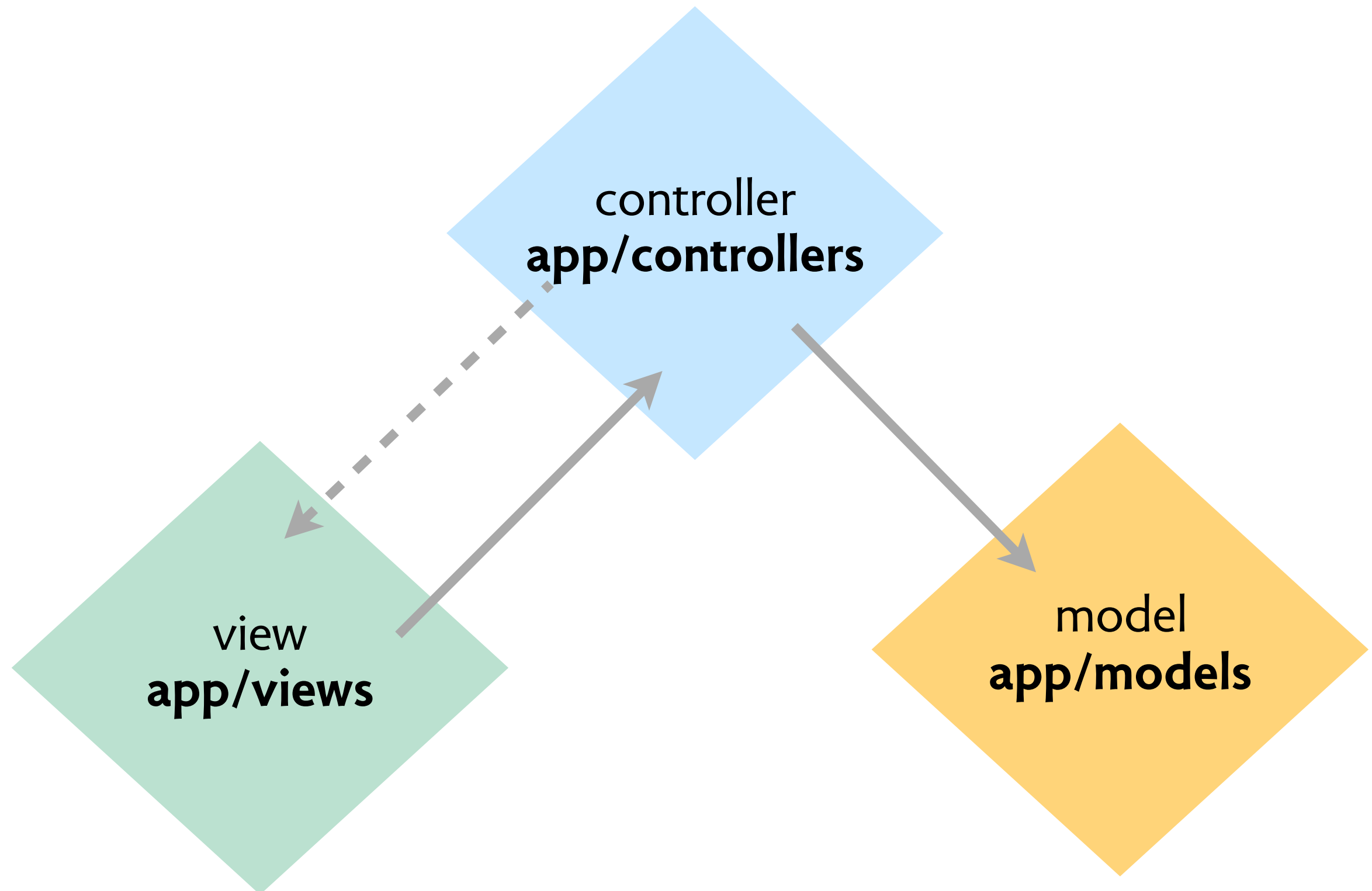
data access & storage

- › defines effect of operations on persistent state

MVC structure



MVC in Rails



example: restaurant review app

in MVC, where to make these changes?

- › increase font size of reviews
- › allow editing if logged in as author
- › change rating of zero to mean no rating

JWT creation

```
import jwt from 'jsonwebtoken';

const payload = { sub: '12345', role: 'user' };
const secret = process.env.JWT_SECRET;

const token = jwt.sign(payload, secret, {
  expiresIn: '15m'
});

console.log('Access Token:', token);
```

JWT creation — another library

```
import { SignJWT } from 'jose';

const secret = new TextEncoder().encode(process.env.JWT_SECRET);

const token = await new SignJWT({ sub: '12345', role: 'user' })
  .setProtectedHeader({ alg: 'HS256' })
  .setIssuedAt()
  .setExpirationTime('15m')
  .sign(secret);

console.log('Access Token:', token);
```

Loading secrets

```
import dotenv from 'dotenv';  
dotenv.config();  
  
const secret = process.env.JWT_SECRET;
```

Checking tokens

```
function authenticateToken(req, res, next) {  
  const authHeader = req.headers['authorization'];  
  const token = authHeader && authHeader.split(' ')[1];  
  if (!token) {  
    return res.status(401).json({message: 'No token'});  
  }  
  jwt.verify(token, process.env.JWT_SECRET,  
    (err, decoded) => {  
    if (err) {  
      return res.status(403).json({  
        message: 'Invalid or expired token' });  
    }  
    req.user = decoded;  
    next();  
  });  
}
```

Introduction to React

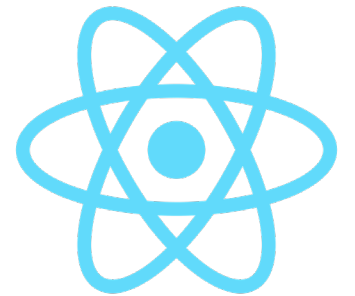
A workshop for COMP 523

Aaron Smith

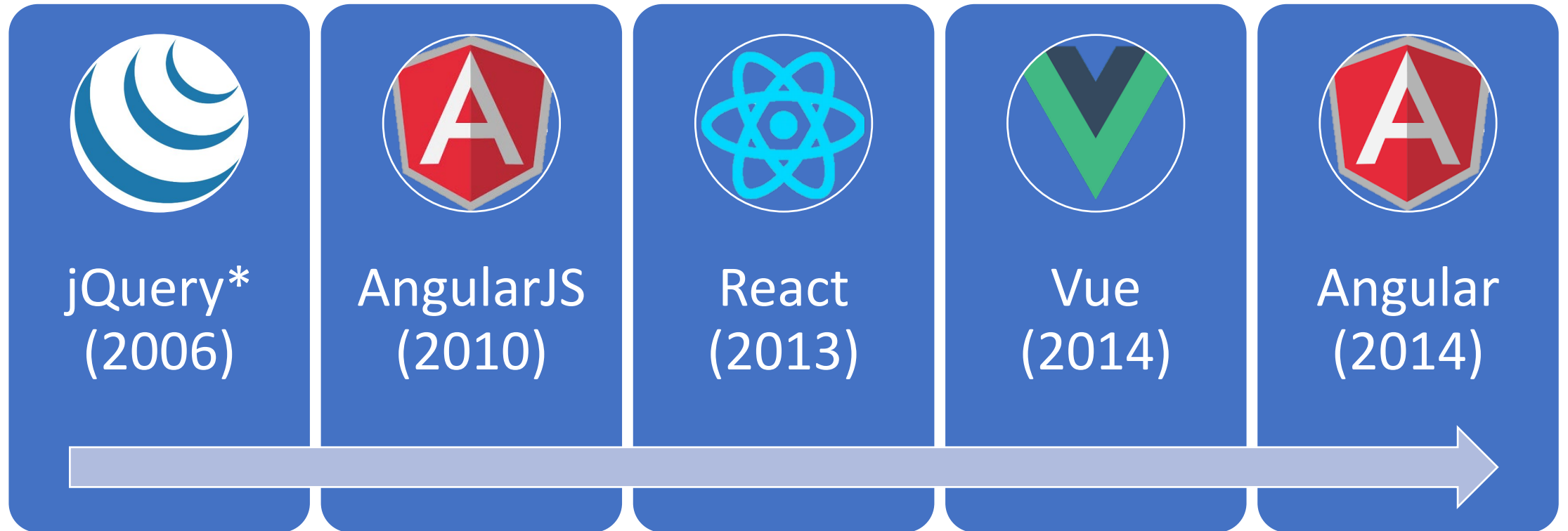
Monday, Feb. 10, 2020

What is React?

- React is a **JavaScript framework**
- Used for **front end web development**
- Think of jQuery, but more structured
- Created and used by **Facebook**
- Famous for implementing a **virtual dom**



Timeline of front-end JavaScript frameworks



* jQuery is more often considered a **library** than a **framework**

Common tasks in front-end development

App state

Data definition, organization, and storage

User actions

Event handlers respond to user actions

Templates

Design and render HTML templates

Routing

Resolve URLs

Data fetching

Interact with server(s) through APIs and AJAX

Fundamentals of React

1. JavaScript and HTML in the same file (JSX)
2. Embrace functional programming
3. Components everywhere

JavaScript and HTML *in the same file*



Traditional
approach



React
approach

JSX: the React programming language

```
const first = "Aaron";  
const last  = "Smith";  
  
const name = <span>{first} {last}</span>;
```

```
const list = (  
  <ul>  
    <li>Dr. David Stotts</li>  
    <li>{name}</li>  
  </ul>  
>);
```

```
const listWithTitle = (  
  <>  
    <h1>COMP 523</h1>  
    <ul>  
      <li>Dr. David Stotts</li>  
      <li>{name}</li>  
    </ul>  
  </>  
>);
```

“React is just JavaScript”

Functional programming

1. Functions are “first class citizens”
2. Variables are immutable
3. Functions have no side effects

Functional programming

Functions are “**first class citizens**”

This means functions can be...

1. Saved as **variables**
2. Passed as **arguments**
3. **Returned** from functions

```
let add = function() {  
  console.log('Now adding numbers');  
  const five = 3 + 2;  
};
```

```
function performTask(task) {  
  task();  
  console.log('Task performed!');  
}  
  
performTask(add);
```

```
function foo() {  
  return function() {  
    console.log('What gets printed?');  
  };  
}  
  
foo  
foo();  
foo()();
```

Functional programming

Variables are **immutable**

Tip: Use **const** instead of **let** to declare variables!

```
let a = 4;  
a = 2; // Mutates `a`
```

```
let b = [1, 2, 3];  
b.push(4); // Mutates `b`  
let c = [...b, 4]; // Does not mutate `b`
```

Functional programming

Functions have **no side effects**

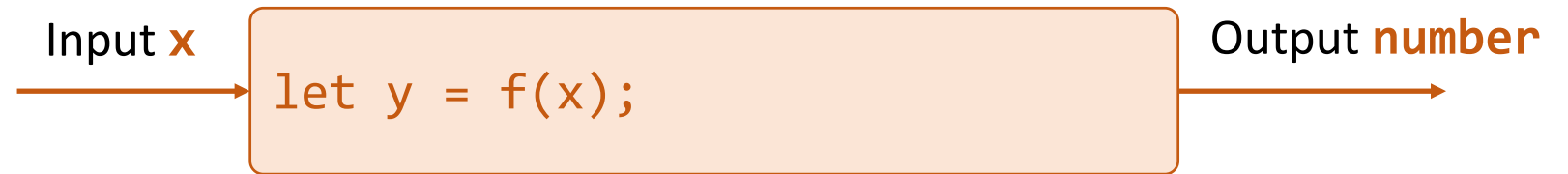
```
const b = [];  
  
function hasSideEffects() {  
  b = [0];  
}
```

Components

Functions help break your code into small, reusable pieces

Components are **functions** for **user interfaces**

Math function:



Component function:



Anatomy of a React **component**

The component is just a function

Inputs are passed through a single argument called "props"

```
export default function MyComponent(props) {  
  return <div>Hello, world! My name is {props.name}</div>;  
}
```

The function outputs HTML

```
const html = <MyComponent name="aaron" />;
```

The function is **executed** as if it was an HTML tag

Parameters are passed in as HTML attributes

Component **rendering**

- When a component function **executes**, we say it “**renders**”
- Assume components may re-render at any time

Our job is to ensure that
every time the component re-renders,
the correct output is produced

“In React, everything is a component”

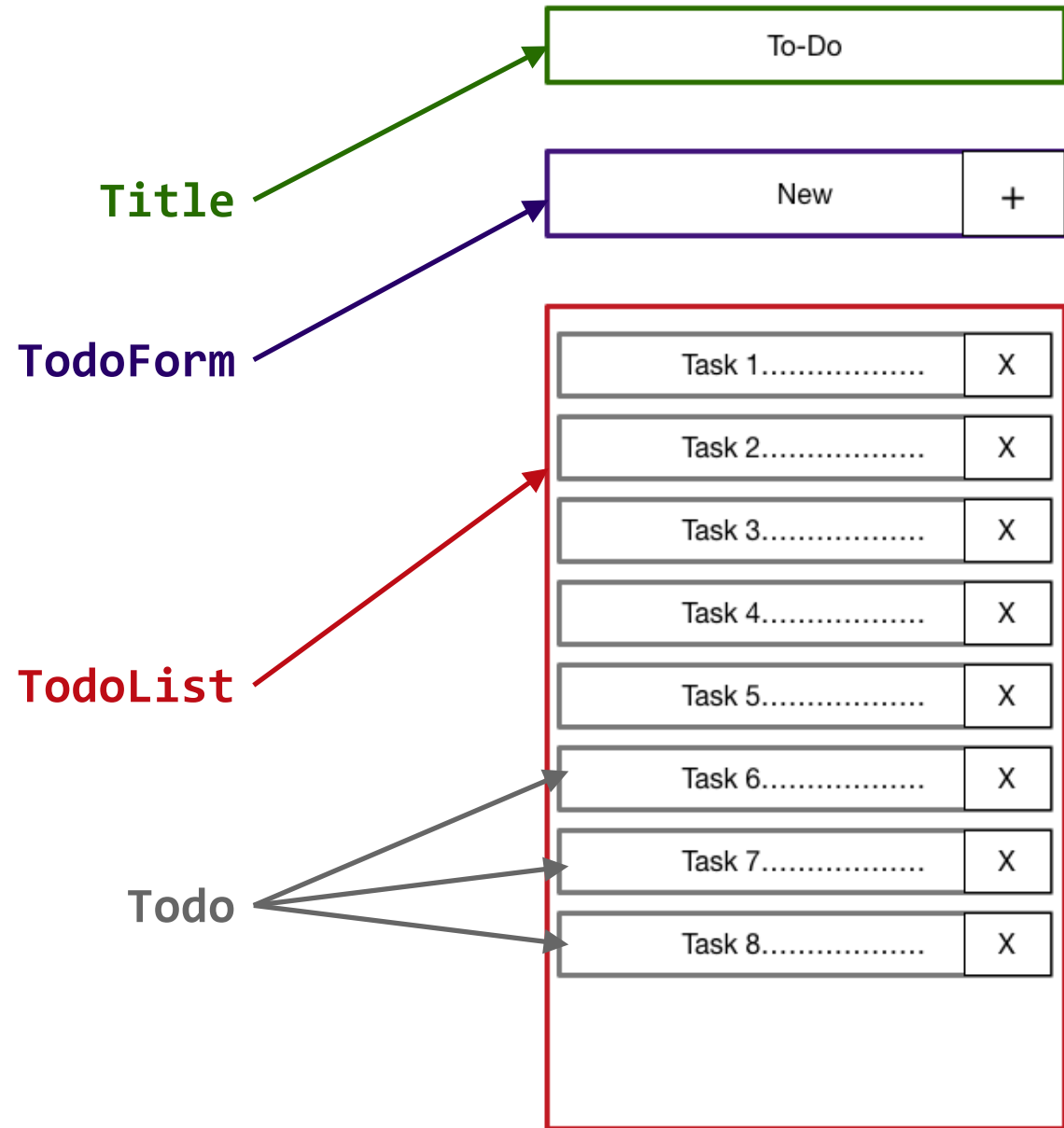
Todo application

Big idea:

- A digital to-do list

First step:

- mockup / wireframe



Creating a new React app

Creating a new React app is simple!

1. Install Node.js
2. Run: **npx create-react-app app-name**
3. New app created in folder: **./app-name**

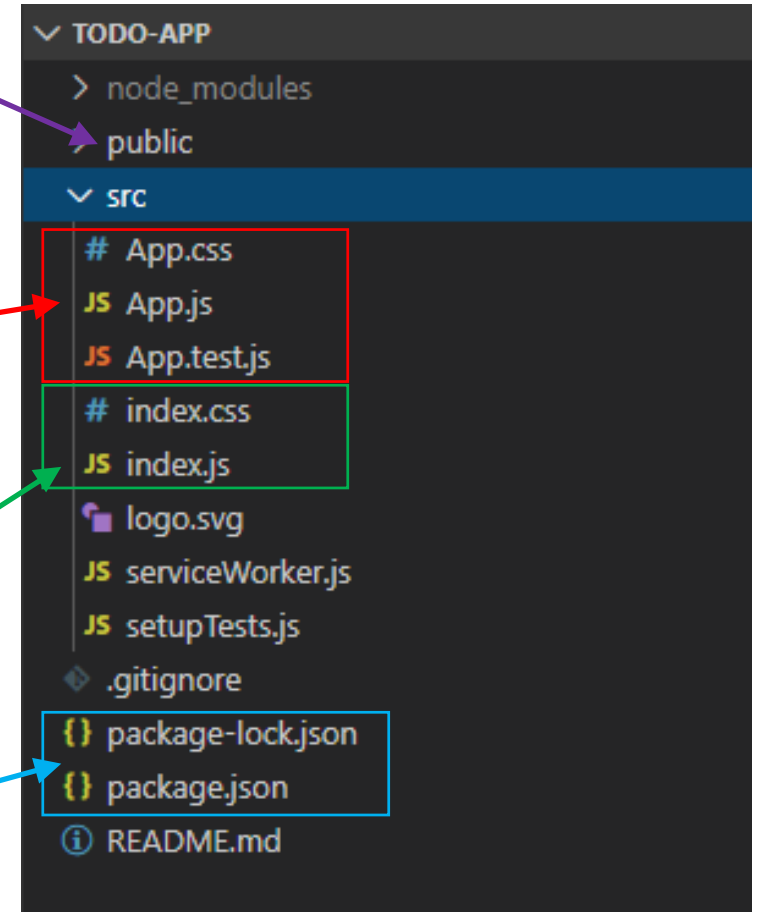
Anatomy of a new React app

public holds the initial html document and other static assets

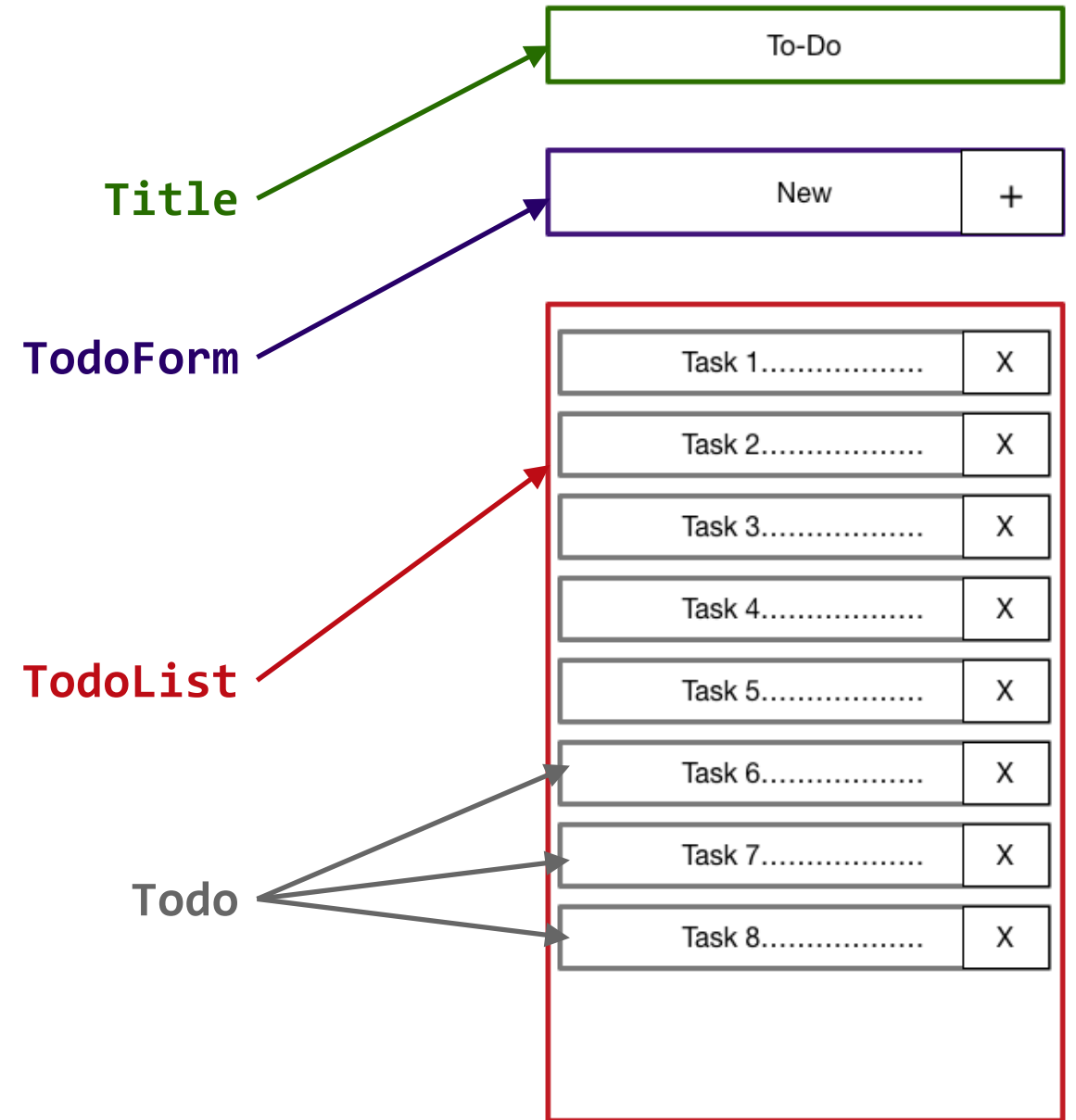
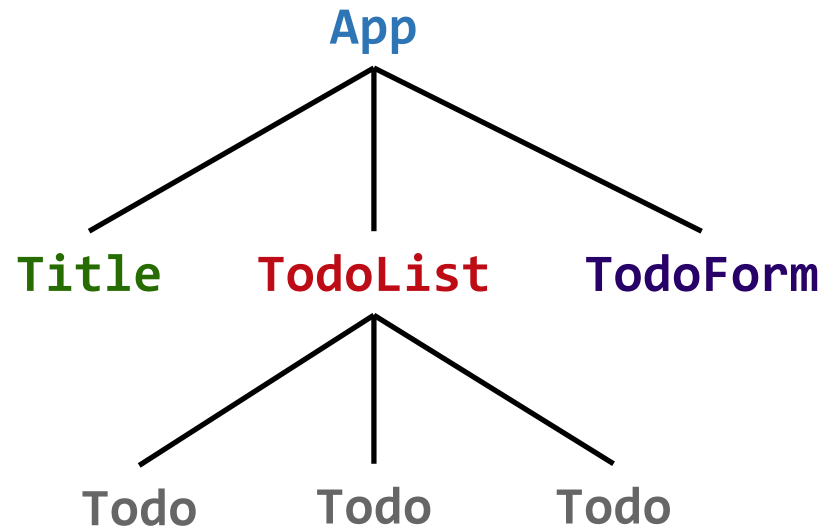
App is a boilerplate starter component

index.js binds React to the DOM

package.json configures npm dependencies



Component Hierarchy



Special list **key** property

- **Situation:** Display a **dynamic array of elements**
- Must specify a special “**key**” property for each element
- The key of an item **uniquely identifies it**
- Used by React internally for **render optimization**
- Can be any unique value (string or number)

What are **hooks**?

Hooks: Special functions that allow developers to hook into **state** and **lifecycle** of React components.

State: One or more data values associated with a React component instance.

Lifecycle: The events associated with a React component instance (create, render, destroy, etc).

Built-in hooks:

We will cover
these today

{ useState
useEffect

We will **not** cover
these today

{ useReducer
useMemo
useRef
useCallback

First React hook: `useState`

Purpose:

1. Remember values internally when the component re-renders
2. Tell React to re-render the component when the value changes

Syntax:

```
const [val, setVal] = useState(100);
```

The current value

A setter function to
change the value

The initial
value to use

Predicting component re-rendering

A component will only re-render when...

1. A value inside **props** changes

– or –

2. A **useState** setter is called

This means all data values displayed in the HTML should depend on either **props** or **useState**

Second React hook: **useEffect**

Purpose:

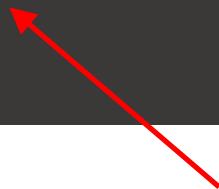
Act as an **observer**, running code in response to value changes

Syntax:

```
useEffect(() => {  
  console.log(`myValue was changed! New value: ${myValue}`);  
}, [myValue]);
```



A list of values such that changes
should trigger this code to run



The code to run when
values change

Building a React project

- When you're ready to launch your app, run this command:

npm run build

- This bundles your app into CSS/JS/HTML files and puts them in the **/build** folder
- These files can be served from an AWS S3 bucket

3rd party components and libraries

- React-Router
- Redux
- Material-UI
- Bootstrap
- Font-Awesome
- SWR

React props

```
import React from 'react';

function Greet(props) {
  return <h1>Hello, {props.name}!</h1>;
}

function App() {
  return <Greet name="John" />;
}

export default App;
```

React props: multiple

```
import React from 'react';

function Greet(props) {
  return <h1>Hello, {props.name}! Ready for {props.age}
  birthday?</h1>;
}

function App() {
  return <Greet name="John" age={25} />;
}

export default App;
```

React useState

```
import { useState } from 'react';
import './App.css';

function App() {
  const [value, setValue] = useState(100);
  return (
    <div className="App">
      <span onClick={function(){setValue(value+1)}}>
        {value * 3}</span> </div>
    );
}

export default App;
```

React useEffect

```
import { useState, useEffect } from 'react';
import './App.css';

function App() {
  const [value, setValue] = useState(100);
  useEffect(() => {console.log(value)}, [value]);
  return (
    <div className="App">
      <span onClick={function(){setValue(value+1)}}>
        {value * 3}</span> </div>
    );
}

export default App;
```

Usability and accessibility of HTML elements

ARIA (Accessibility) button role:

- clickable element

- trigger response when activated

`role="button"` — tells screen reader it's a button

Expected behaviours:

- click events

- keyboard handling

- ...

Without screen reader should *look* clickable

Everything inside is presentational not semantic

`button / input type="button"` do things reasonably by default

Usability and accessibility of HTML elements

ARIA (Accessibility) button role:

- clickable element

- trigger response when activated

`role="button"` — tells screen reader it's a button

Expected behaviours:

- click events

- keyboard handling

- ...

Without screen reader should *look* clickable

Everything inside is presentational not semantic

`button / input type="button"` do things reasonably by default

Usability and accessibility of HTML elements

ARIA (Accessibility) button role:

- clickable element

- trigger response when activated

`role="button"` — tells screen reader it's a button

Expected behaviours:

- click events

- keyboard handling

- ...

Without screen reader should *look* clickable

Everything inside is presentational not semantic

`button / input type="button"` do things reasonably by default

Usability and accessibility of HTML elements

ARIA (Accessibility) button role:

- clickable element

- trigger response when activated

`role="button"` — tells screen reader it's a button

Expected behaviours:

- click events

- keyboard handling

- ...

Without screen reader should *look* clickable

Everything inside is presentational not semantic

`button / input type="button"` do things reasonably by default

Usability and accessibility of HTML elements

ARIA (Accessibility) button role:

- clickable element

- trigger response when activated

`role="button"` — tells screen reader it's a button

Expected behaviours:

- click events

- keyboard handling

- ...

Without screen reader should *look* clickable

Everything inside is presentational not semantic

`button` / `input type="button"` do things reasonably by default

Artificial button

`role="button"`

CSS to look clickable

Form relations

Possibly with validation

`tabindex=` — navigation/focus

`keydown` — Space/Enter also activate

`touch/pointer` (if mouse not click)

Maybe disabling support

`click` — as for normal button

Maybe `aria-label` (if content not screen-readable) — as for normal buttons

https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Reference/Roles/button_role

[https:](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

[//madcampos.dev/blog/2026/07/accessibility-from-scratch/](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

Artificial button

`role="button"`

CSS to look clickable

Form relations

Possibly with validation

`tabindex=` — navigation/focus

`keydown` — Space/Enter also activate

`touch/pointer` (if mouse not click)

Maybe disabling support

`click` — as for normal button

Maybe `aria-label` (if content not screen-readable) — as for normal buttons

https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Reference/Roles/button_role

[https:](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

[//madcampos.dev/blog/2026/07/accessibility-from-scratch/](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

Artificial button

`role="button"`

CSS to look clickable

Form relations

Possibly with validation

`tabindex=` — navigation/focus

`keydown` — Space/Enter also activate

`touch/pointer` (if mouse not click)

Maybe disabling support

`click` — as for normal button

Maybe `aria-label` (if content not screen-readable) — as for normal buttons

https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Reference/Roles/button_role

[https:](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

[//madcampos.dev/blog/2026/07/accessibility-from-scratch/](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

Artificial button

`role="button"`

CSS to look clickable

Form relations

Possibly with validation

`tabindex=` — navigation/focus

`keydown` — Space/Enter also activate

`touch/pointer` (if mouse not click)

Maybe disabling support

`click` — as for normal button

Maybe `aria-label` (if content not screen-readable) — as for normal buttons

https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Reference/Roles/button_role

[https:](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

[//madcampos.dev/blog/2026/07/accessibility-from-scratch/](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

Artificial button

`role="button"`

CSS to look clickable

Form relations

Possibly with validation

`tabindex=` — navigation/focus

`keydown` — Space/Enter also activate

`touch/pointer` (if mouse not click)

Maybe disabling support

`click` — as for normal button

Maybe `aria-label` (if content not screen-readable) — as for normal buttons

https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Reference/Roles/button_role

[https:](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

[//madcampos.dev/blog/2026/07/accessibility-from-scratch/](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

Artificial button

`role="button"`

CSS to look clickable

Form relations

Possibly with validation

`tabindex=` — navigation/focus

`keydown` — Space/Enter also activate

`touch/pointer` (if mouse not click)

Maybe disabling support

`click` — as for normal button

Maybe `aria-label` (if content not screen-readable) — as for normal buttons

https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Reference/Roles/button_role

[https:](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

[//madcampos.dev/blog/2026/07/accessibility-from-scratch/](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

Artificial button

`role="button"`

CSS to look clickable

Form relations

Possibly with validation

`tabindex=` — navigation/focus

`keydown` — Space/Enter also activate

`touch/pointer` (if mouse not click)

Maybe disabling support

`click` — as for normal button

Maybe `aria-label` (if content not screen-readable) — as for normal buttons

https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Reference/Roles/button_role

[https:](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

[//madcampos.dev/blog/2026/07/accessibility-from-scratch/](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

Artificial button

`role="button"`

CSS to look clickable

Form relations

Possibly with validation

`tabindex=` — navigation/focus

`keydown` — Space/Enter also activate

`touch/pointer` (if mouse not click)

Maybe disabling support

`click` — as for normal button

Maybe `aria-label` (if content not screen-readable) — as for normal buttons

https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Reference/Roles/button_role

[https:](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

[//madcampos.dev/blog/2026/07/accessibility-from-scratch/](https://madcampos.dev/blog/2026/07/accessibility-from-scratch/)

Semantic labels

```
<div class="preference">  
  <label for="cheese">I like cheese.</label>  
  <input type="checkbox" name="cheese" id="cheese" />  
</div>
```

Input-label association

Screen-readable

Label becomes clickable