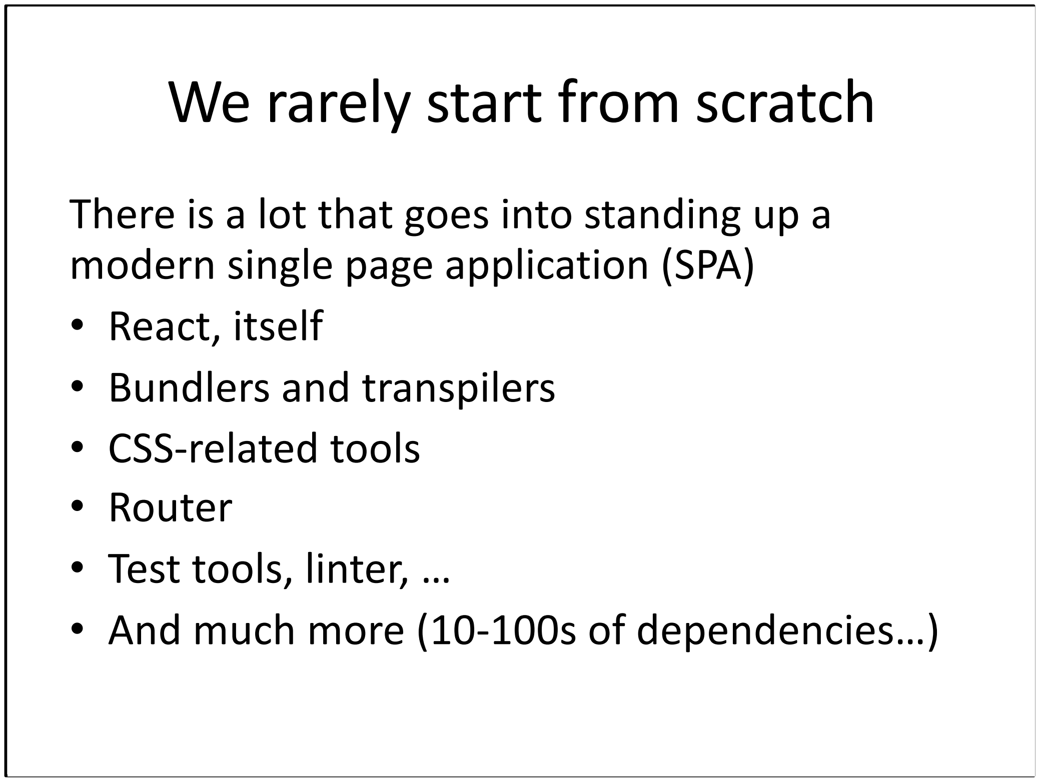
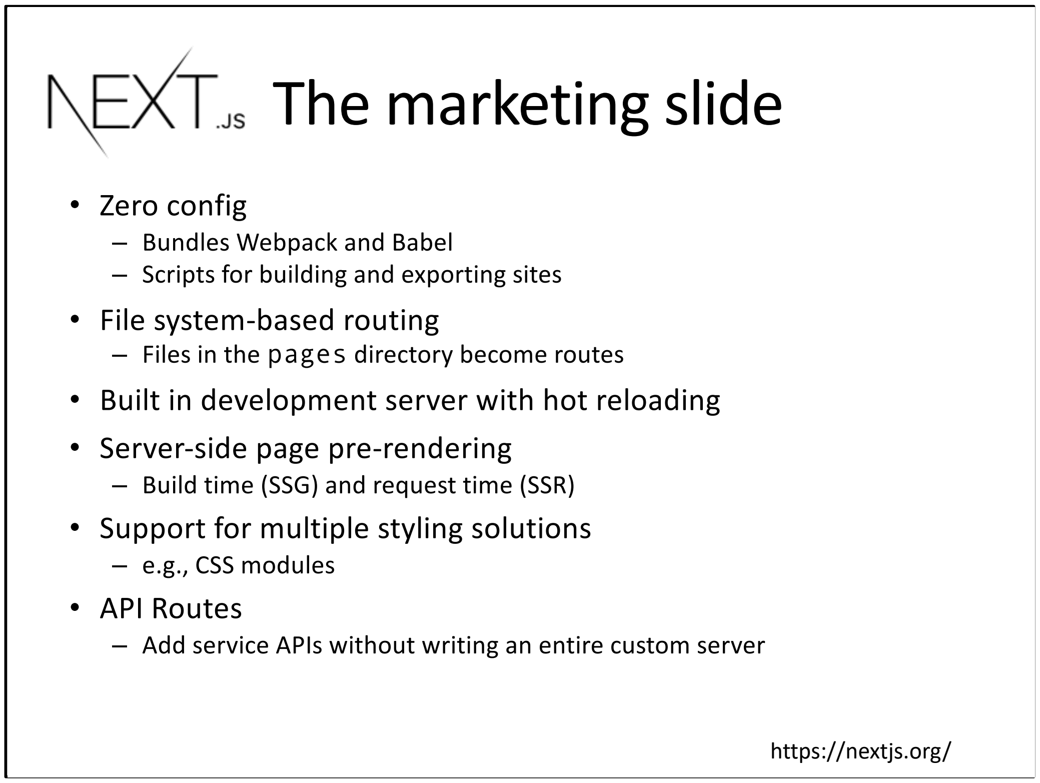


We rarely start from scratch

There is a lot that goes into standing up a modern single page application (SPA)

- React, itself
- Bundlers and transpilers
- CSS-related tools
- Router
- Test tools, linter, …
- And much more (10-100s of dependencies…)

We could absolutely assemble all of these technologies ourselves, but that is a lot of work (and a lot decisions for which we many not have enough information to have an informed opinion). Instead, we are going to use frameworks that wrap around React that have already integrated many (most?) of the tools we need to get started.



NEXT.JS The marketing slide

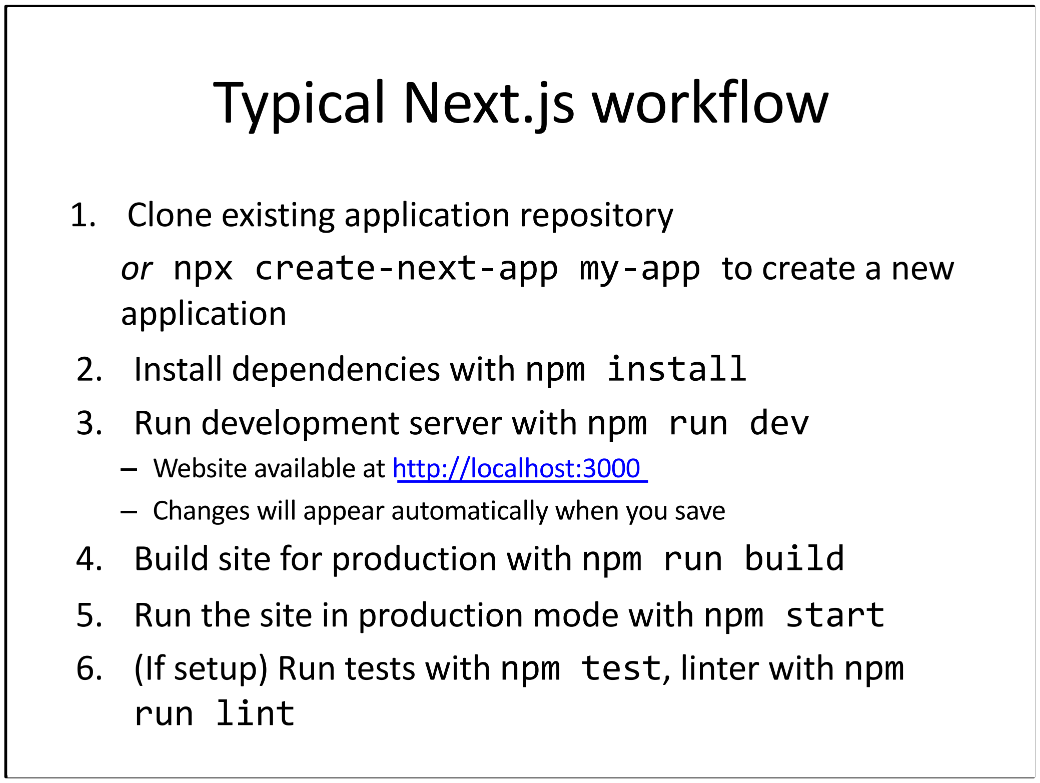
- Zero config
  - Bundles Webpack and Babel
  - Scripts for building and exporting sites
- File system-based routing
  - Files in the `pages` directory become routes
- Built in development server with hot reloading
- Server-side page pre-rendering
  - Build time (SSG) and request time (SSR)
- Support for multiple styling solutions
  - e.g., CSS modules
- API Routes
  - Add service APIs without writing an entire custom server

https://nextjs.org/

React just provides the component system. We will be using Next.js to manage development and some of our server functionality. We can think of Next.js as providing a more full featured (and opinionated) setup around React. Instead of use choosing among and integrating this functionality on our own, Next.js has done so already. Next.js is not the only way to setup a React application (we previously used CreateReactApp), and as with some of our other tools you/we may not agree with some of the choices made by the Next.js developers. However, it provides a robust starting point with many best practices built in.

This is a partial list of the features listed on the Next.js front page (primarily the ones that are interesting to us). While we won’t necessarily care about the speed boost from the server-side pre-rendering, it is a major feature of the framework, so it merits notice.

You have already been using Next for some of the initial practicals and assignments. We are going to start to dig into its features a bit more…



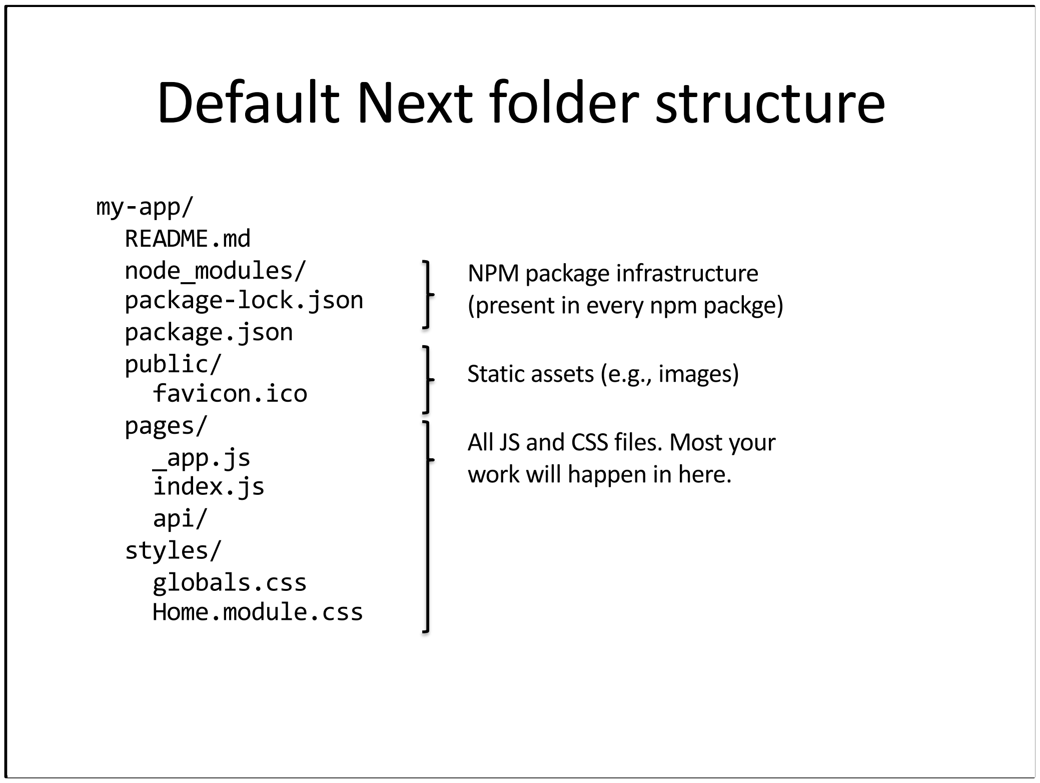
Typical Next.js workflow

1. Clone existing application repository

   or `npx create-next-app my-app` to create a new application
2. Install dependencies with `npm install`
3. Run development server with `npm run dev`
   – Website available at http://localhost:3000
   – Changes will appear automatically when you save
4. Build site for production with `npm run build`
5. Run the site in production mode with `npm start`
6. (If setup) Run tests with `npm test`, linter with `npm run lint`

Will will typically be cloning an existing repository so that I can provide starter code. That is I already did step 1…

We will typically not do steps 4 and 5 locally, but instead those steps will be integrated into our deployment workflow.



Default Next folder structure

```
my-app/
  README.md
  node_modules/
  package-lock.json
  package.json
  public/
    favicon.ico
  pages/
    _app.js
    index.js
    api/
  styles/
    globals.css
    Home.module.css
```

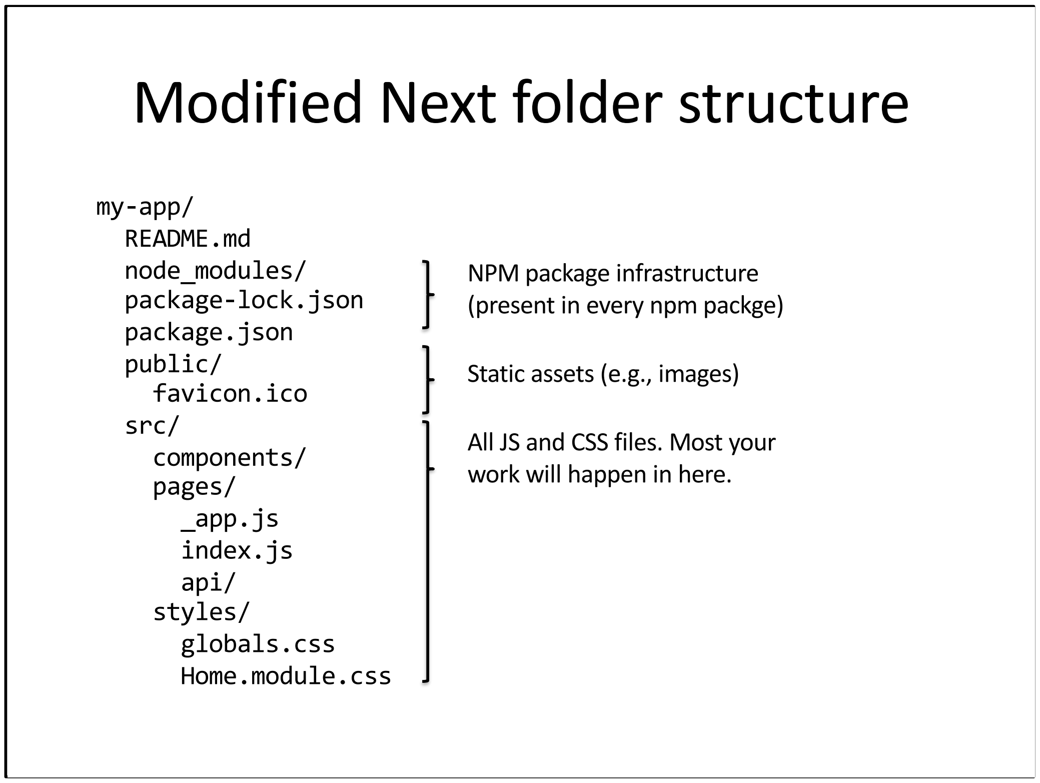
NPM package infrastructure (present in every npm packge)

Static assets (e.g., images)

All JS and CSS files. Most your work will happen in here.

Next uses a very specific directory structure, with functionality dependent on file location/naming. This is an example of "convention vs. configuration". That is by sticking to certain conventions we don't have to explicitly configure our application. Only atypical situations require explicit configuration (to override the convention). This is a common approach for web frameworks where the goal is to make the typical case easy.

One caveat for Next, is that the directory structure is evolving quite a bit and so you might see different setups. Specifically we will use the following structure [click].



Modified Next folder structure

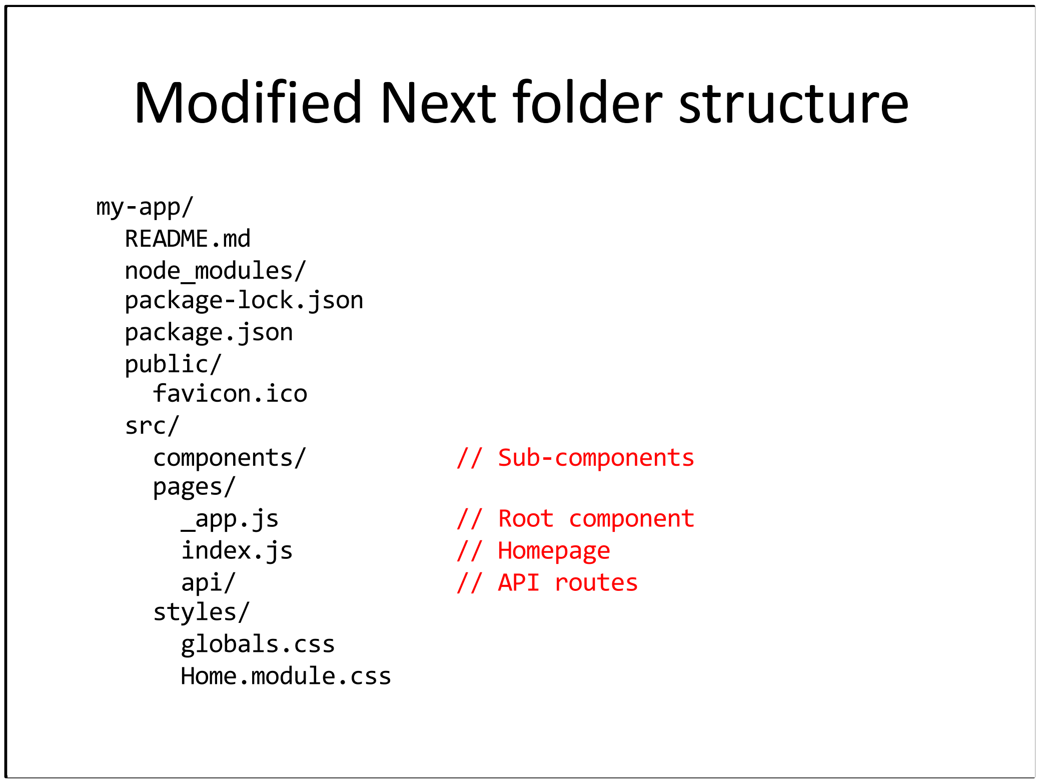
```
my-app/
  README.md
  node_modules/
  package-lock.json
  package.json
  public/
    favicon.ico
  src/
    components/
    pages/
      _app.js
      index.js
      api/
    styles/
      globals.css
      Home.module.css
```

NPM package infrastructure (present in every npm packge)

Static assets (e.g., images)

All JS and CSS files. Most your work will happen in here.

This is the structure that you will see for our applications, where all of the “code” is contained in the “src” directory. Let’s dig into that “src” directory a bit more…



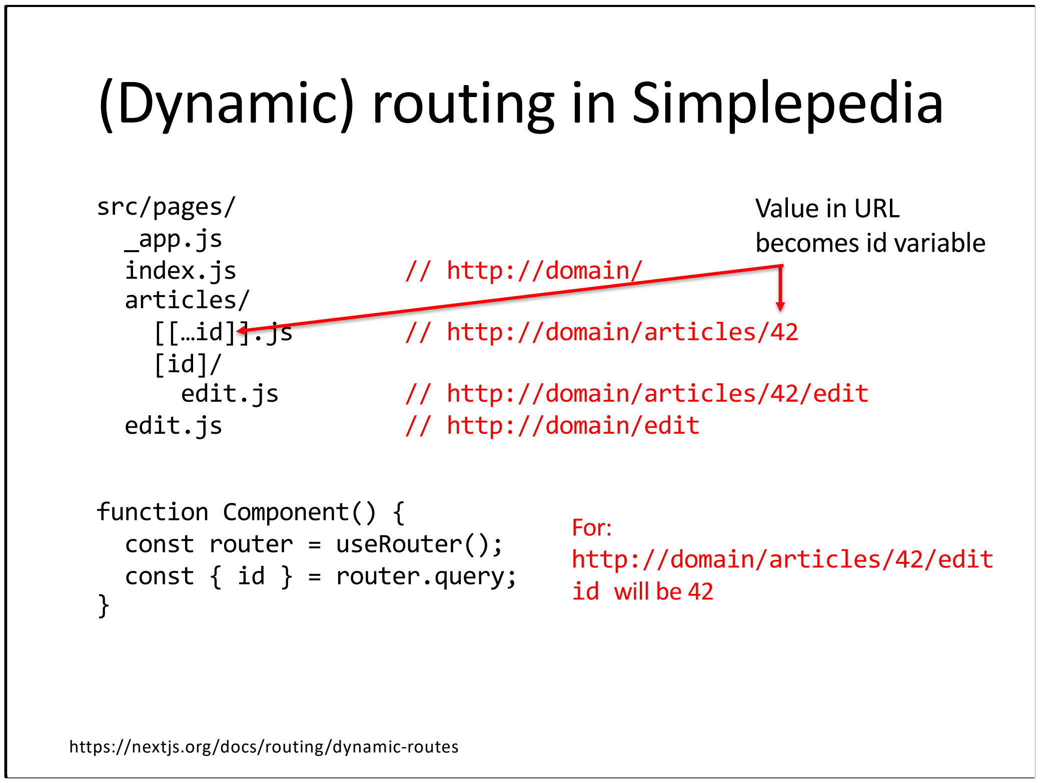
Modified Next folder structure

```
my-app/
  README.md
  node_modules/
  package-lock.json
  package.json
  public/
    favicon.ico
  src/
    components/          // Sub-components
    pages/
      _app.js            // Root component
      index.js           // Homepage
      api/               // API routes
    styles/
      globals.css
      Home.module.css
```

Components contains React components used in (potentially many) pages. These components may be used anywhere in our application. We separate them out to facilitate that reuse. For example, our “in-class” tool is a Next application. The same poll component is re-used in all the different interfaces (participant, instructor, and the view layered on the screen…).

The pages directory describes separate “pages” in our application, with the “api” sub-directory implementing code that runs exclusively on the server not the browser. We will talk about that aspect later in the semester.

The notion of multiple pages seems to conflict with the idea of the single page application (SPA). It is a recognition that most SPAs still have distinct views and that the URLs (which we can link to, move between with the browser’s back/forward button) are a helpful tool for managing those views. So, we use those tools, specifically the browser history to maintain state for us. We can use the URL to determine which component we want to show on the screen at any one time , effectively treating the URL like other forms of application state! This enables the interactivity (and statefulness) of an SPA, with the familiar interaction mechanics of a multiple page application (e.g., we can link to specific views in our application).



(Dynamic) routing in Simplepedia

```
src/pages/
  _app.js
  index.js            // http://domain/
  articles/
    [[…id]].js        // http://domain/articles/42
    [id]/
      edit.js         // http://domain/articles/42/edit
  edit.js             // http://domain/edit
```

Value in URL becomes id variable

```
function Component() {
  const router = useRouter();
  const { id } = router.query;
}
```

For: http://domain/articles/42/edit
id will be 42

https://nextjs.org/docs/routing/dynamic-routes

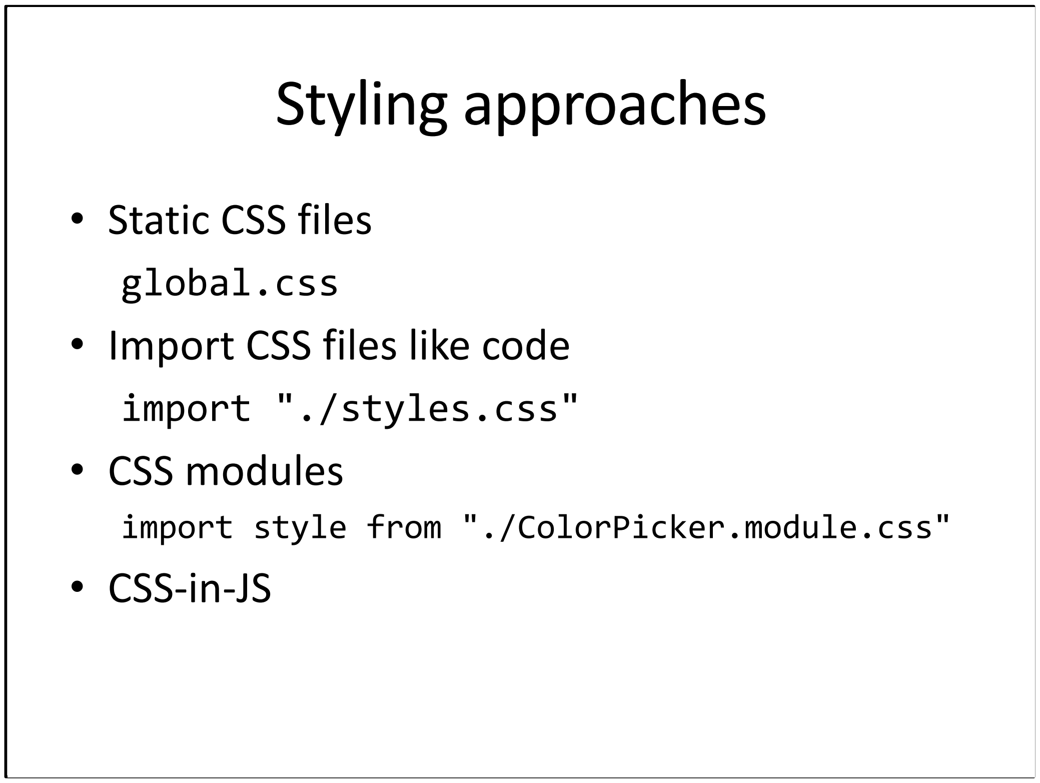
The pages directory is an example of convention over configuration. When we navigate to a URL, e.g., http://domain/articles/42/edit, Next.js will render the components in the associated file based on the directory structure in pages (i.e., the router is determined by the directory convention not a configuration file somewhere that maps routes to components). In some cases, the files define static routes, e.g., a fixed mapping between names and files. In others the routing is dynamic, that is multiple routes map to the same file, with part that varies extract as a variable. These are indicated by the square brackets of various kinds. [click]. In this example, that variable is named id.

Specific examples shown here (see documentation for me).
[[…id]], Optional catch all route, e.g., will match /, /a, /a/b, …, assigning an array to id
[id], Match route and assign to id

Note more specific URLs take precedence, that is why /articles/42/edit matches as shown instead of the catch all.

With the component rendered by Next, we can access the router (the functionality that selected the component) to obtain the variable and other information, e.g., the extract the id variable for use in the application. We can also use the router to switch between components, e.g., to switch articles by "navigating" to say /articles/26 (article with id 26). We say "navigate", but that is not really what is happening. In practice, we are adding an entry to the browser history and updating the router state to (re)render the correct component, with the new URL and associated variables.

Check out documentation: https://nextjs.org/docs/routing/dynamic-routes



Styling approaches

- Static CSS files

  `global.css`
- Import CSS files like code

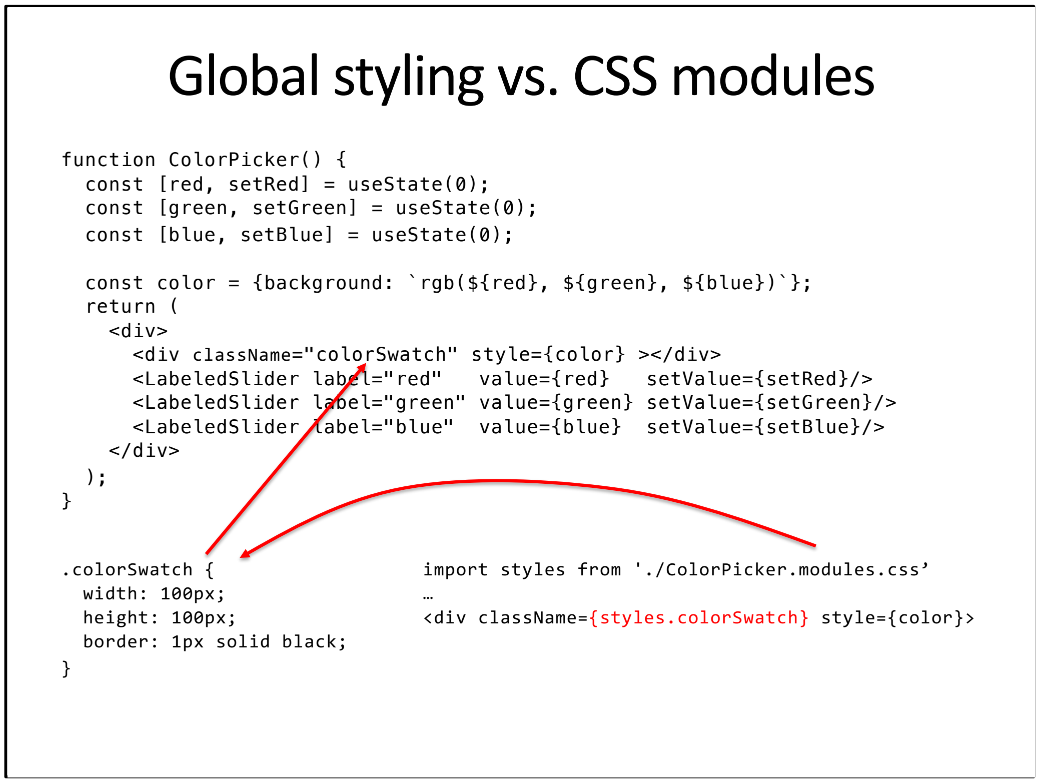
  `import "./styles.css"`
- CSS modules

  `import style from "./ColorPicker.module.css"`
- CSS-in-JS

We can include a static CSS file as an asset, i.e., the traditional approach. But this approach is not very modular and doesn't necessarily work well with a component-based design. CSS has a single global name space. We would have to merge the styles for all components into the global file, even components we didn't write ourselves, and hope there no conflicts.

We can "import" CSS files (using features of Webpack to bundle that CSS into the JavaScript file) for each component. This eliminates the need to combine our CSS files but doesn't resolve issues with having all classname is a single, global, namespace. There are still many opportunities for naming collisions. So not a great fit for truly modular components. CSS modules are essentially a scoped version of importing css into the module that address thar issue (by automatically extending the class names with unique identifiers, checkout your applications in the browser developer tools).

CSS-in-JS integrates styling into the components as JavaScript code (similar to our previous example in which we created the styles as JavaScript objects but with many more features, like dynamically changing styles, using swappable themes, etc).



Global styling vs. CSS modules

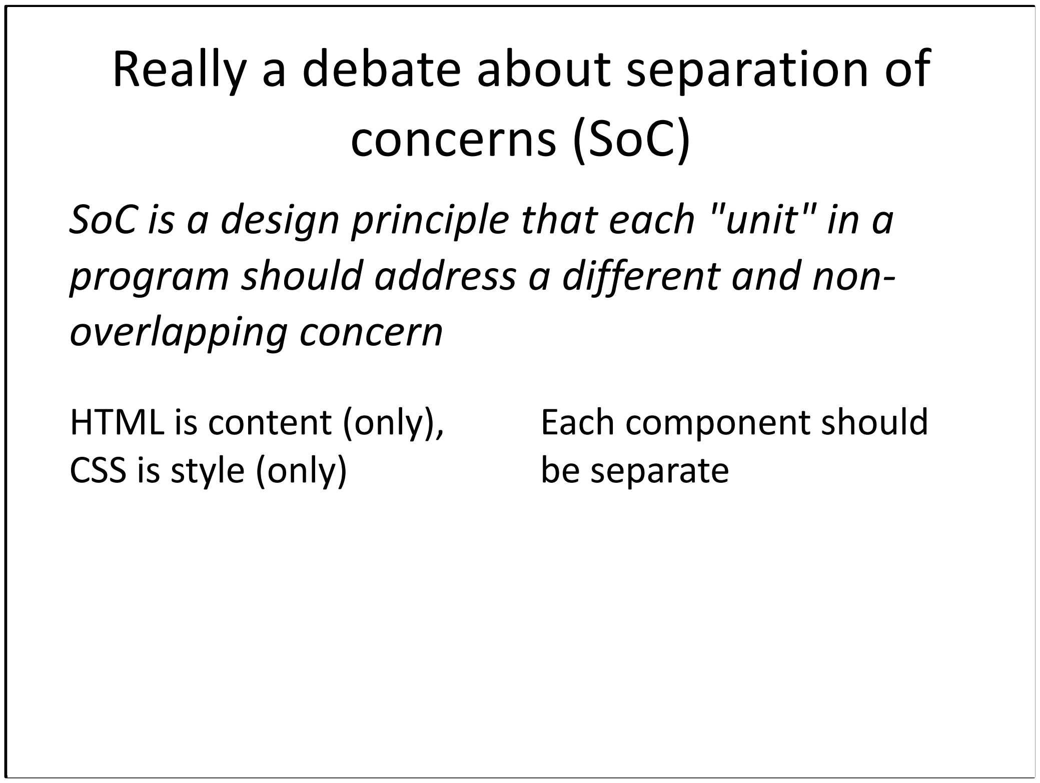
```
function ColorPicker() {
  const [red, setRed] = useState(0);
  const [green, setGreen] = useState(0);
  const [blue, setBlue] = useState(0);

  const color = {background: `rgb(${red}, ${green}, ${blue})`};
  return (
    <div>
      <div className="colorSwatch" style={color} ></div>
      <LabeledSlider label="red"   value={red}   setValue={setRed}/>
      <LabeledSlider label="green" value={green} setValue={setGreen}/>
      <LabeledSlider label="blue"  value={blue}  setValue={setBlue}/>
    </div>
  );
}
```

```
.colorSwatch {
  width: 100px;
  height: 100px;
  border: 1px solid black;
}
```

```
import styles from './ColorPicker.modules.css’
…
<div className={styles.colorSwatch} style={color}>
```

The difference as noted is that it will create a class name like: ColorPicker_colorSwatch__Lu74p



Really a debate about separation of concerns (SoC)

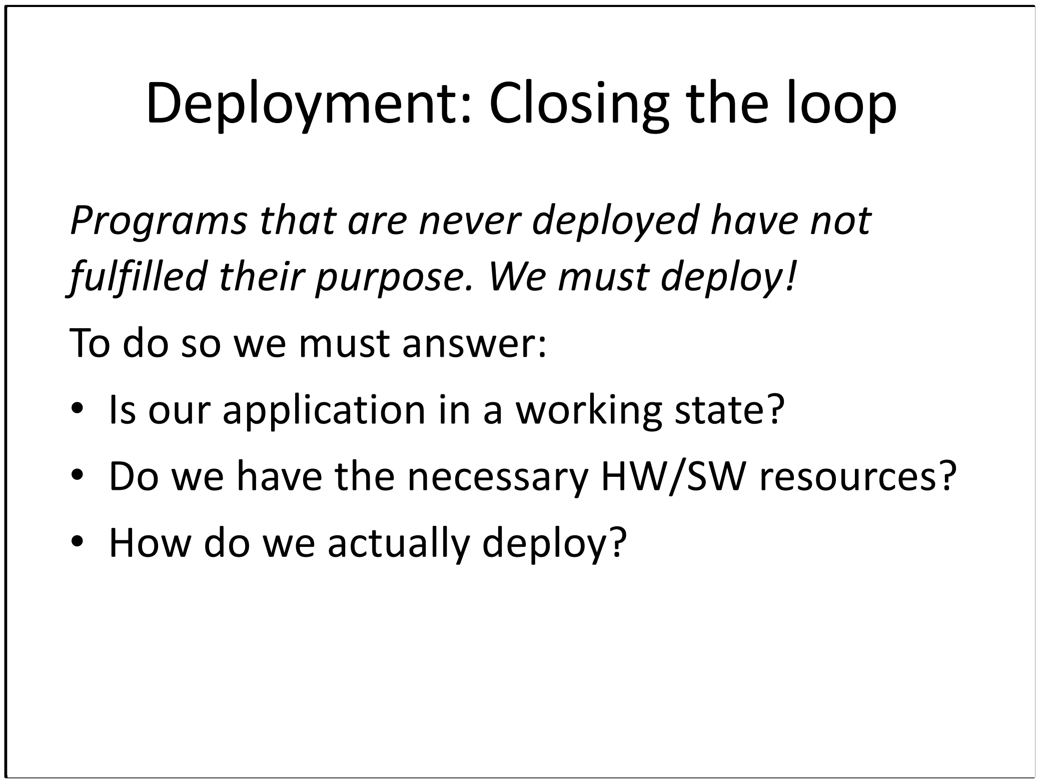
SoC is a design principle that each "unit" in a program should address a different and non-overlapping concern

HTML is content (only), CSS is style (only)

Each component should be separate

Separation of Concerns (SoC) will be a recurring topic this semester, but in short, SoC is a design principle that each “unit” in a program should address a different and non-overlapping concern.

In this context, a common SoC argument around HTML/CSS is that HTML should specify content (only) and CSS should specify the style (only), i.e., separate style from content. Proponents of CSS-in-JS also make a SoC argument, but that one component should be entirely separate from the others.

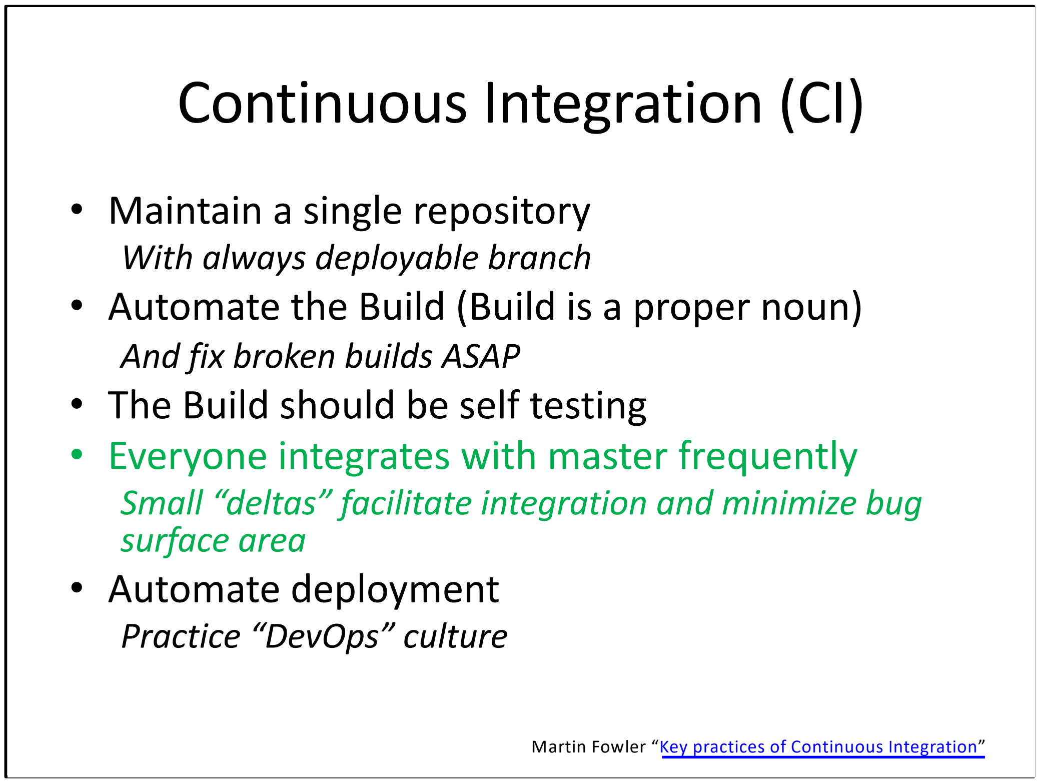


Deployment: Closing the loop

Programs that are never deployed have not fulfilled their purpose. We must deploy!

To do so we must answer:

- Is our application in a working state?
- Do we have the necessary HW/SW resources?
- How do we actually deploy?



Continuous Integration (CI)

- Maintain a single repository
  With always deployable branch
- Automate the Build (Build is a proper noun)
  And fix broken builds ASAP
- The Build should be self testing
- Everyone integrates with master frequently
  Small "deltas" facilitate integration and minimize bug surface area
- Automate deployment
  Practice "DevOps" culture

Martin Fowler "Key practices of Continuous Integration"

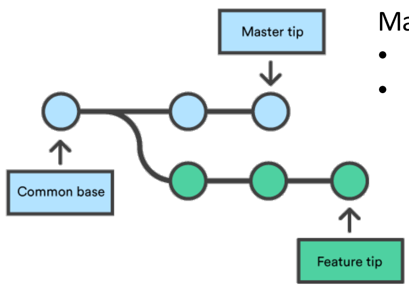
CI emphasizes frequent small integrations (hence the name)

There are two related concepts:
* *Continuous Deployment*: Every change automatically gets put into production, and thus there are many production deployments each day.
* *Continuous Delivery*: An extension of CI in which SW is deployable throughout its lifecycle, the team prioritizes keeping SW deployable, and it is possible to automatically deploy SW on demand.

https://martinfowler.com/bliki/ContinuousDelivery.html

We will be aiming for a Continuous Delivery-like workflow in which our applications start and stay deployable throughout the development process. As with CI, this reduces the complexity (and risk) of deployment by enabling us to do so in small increments. And Continuous Delivery facilitates getting user feedback by frequently getting working SW in front of real users. Although to mitigate risk companies will often first deploy for a small subset of users.

Git workflow for CI

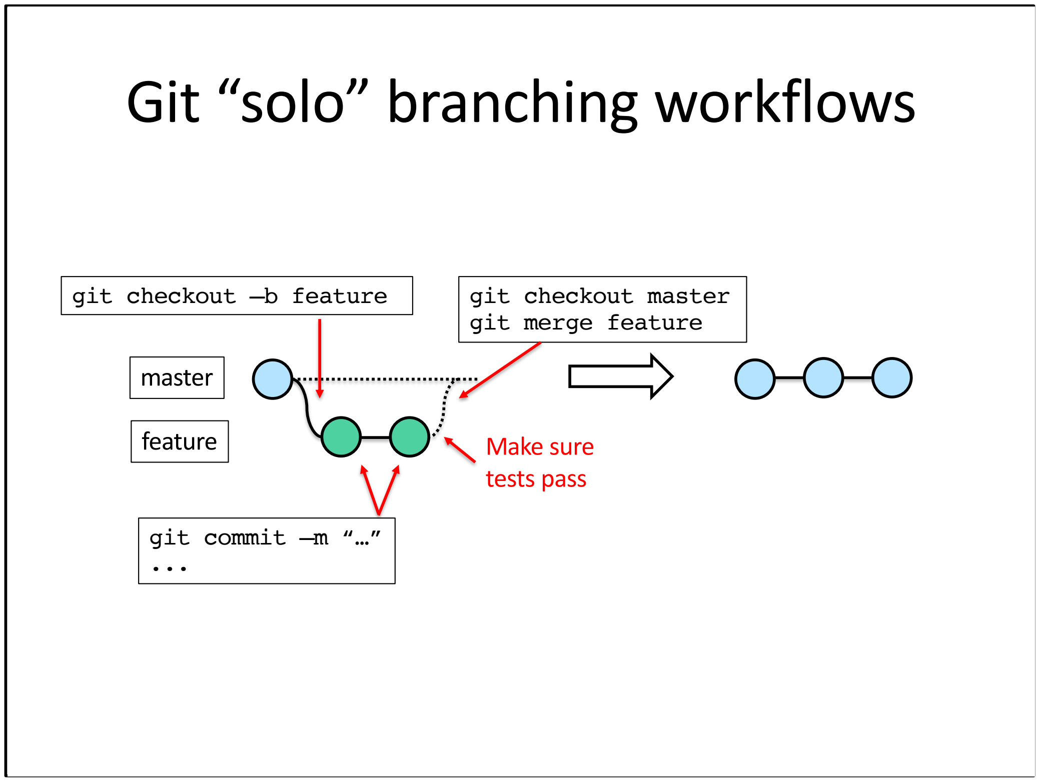


Master is always "deployable"
- Tests pass
- No incomplete features

Short-lived branch for single feature

- Branching is cheap in Git
- We will use features branches to segregate changes until integration
- The "master" branch remains deployable

https://www.atlassian.com/git/tutorials/using-branches



Git “solo” branching workflows

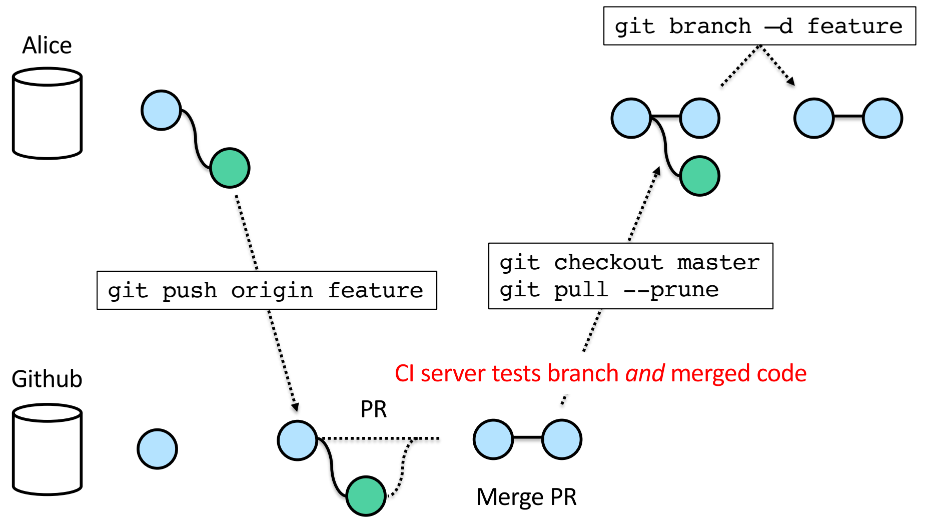
```
git checkout –b feature
```

```
git checkout master
git merge feature
```

```
git commit –m “…”
...
```

But we are rarely working alone. On a team we need to make sure we stay in sync and create opportunities to get a second pair of eyes on our code (i.e. create opportunities for code review).

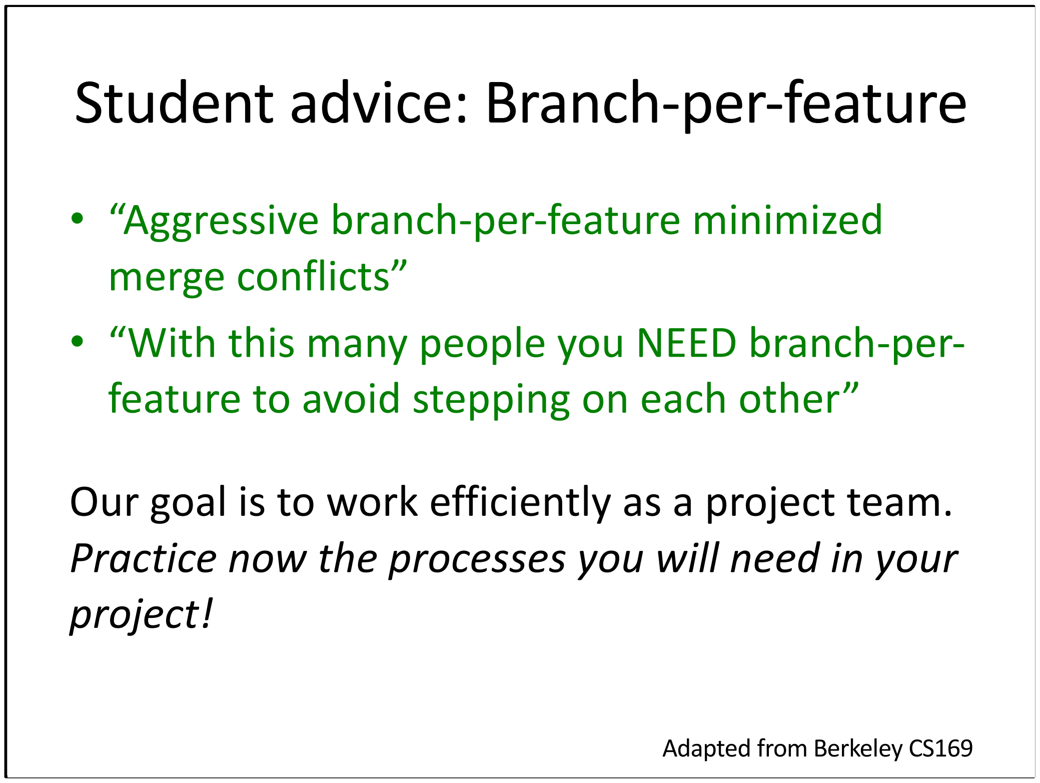
Git/GitHub workflow with CI



```
git push origin feature
```

```
git checkout master
git pull --prune
```

```
git branch –d feature
```



Student advice: Branch-per-feature

- "Aggressive branch-per-feature minimized merge conflicts"
- "With this many people you NEED branch-per-feature to avoid stepping on each other"

Our goal is to work efficiently as a project team. *Practice now the processes you will need in your project!*

Adapted from Berkeley CS169

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.