Layout

UNC COMP 523

Mon Sep 14, 2020

Prof. Jeff Terrell

Announcements

- music: Gymnopedie no.1 by Erik Satie, because it's beautiful IMO
- Q2 grades uploaded to gradebook on Sakai
- currently behind on mentor reports

Outline

- ~~announcements~~
- assignment checkin
- tradeoffs of dependencies
- web and mobile app choices
- layout

Outline

- ~~announcements~~
- **assignment checkin**
- tradeoffs of dependencies
- web and mobile app choices
- layout

Assignment Checkin

- Note: A3: User Stories ☕☕ is your high-level task list for the project
- Your Trello board should have (at least) assignments and user stories in it
- A4: Clickable Prototype ☕☕☕☕ due this week
- A5: APPLES reflection 1 ☕ due next week
- A6: Application Architecture ☕☕☕☕ and A7: Architecture Diagram ☕ due week after next
- Will finish equipping for A6 and A7 today
- Equipping for A8 starts today and will last for several lectures

Outline

- ~~announcements~~
- ~~assignment checkin~~
- **tradeoffs of dependencies**
- web and mobile app choices
- layout

general tradeoffs of dependencies

Speaking generally:

- you depend on something because of some benefit
- but you cede control, so
  - you might not get it exactly your way
  - it might not be 100% reliable

general examples of dependencies

- you delegate work to an employee
  - benefit: you have to do less
  - cost: it might not get done, or not on time or not to your satisfaction
- you have a company and partner with another company for sales and marketing help
- you deploy your app in the cloud instead of running servers yourself
- you use a library instead of writing the code yourself

trust

- Key idea: **when you depend on something else, you're trusting it**
- Not just that it will work or won't break, but also for security
- You brought code that somebody else wrote into your application
- This is always fine, *until it isn't*
- How can you predict what somebody out there might do to violate your trust?
- This can inform your technology choices and justifications for A6 (application architecture)

case study: left-pad

- Azer Koçulu published an open-source library as the `kik` package on npm
- Kik (company) wanted to use the `kik` package, and they approached Azer about it
- Azer didn't want to give up the `kik` package name
- Kik approaches NPM through their conflict resolution policy about a solution
- NPM transfers ownership of `kik` to Kik (company)
- Azer retaliates by unpublishing all of his packages from NPM
- One of those packages, `left-pad`, was a popular dependency, underlying many other packages in the ecosystem
- `npm install` started failing across the board for every package that has a dependency on `left-pad@0.0.3` (even very indirect ones)
- Result: widespread failures across the NPM ecosystem

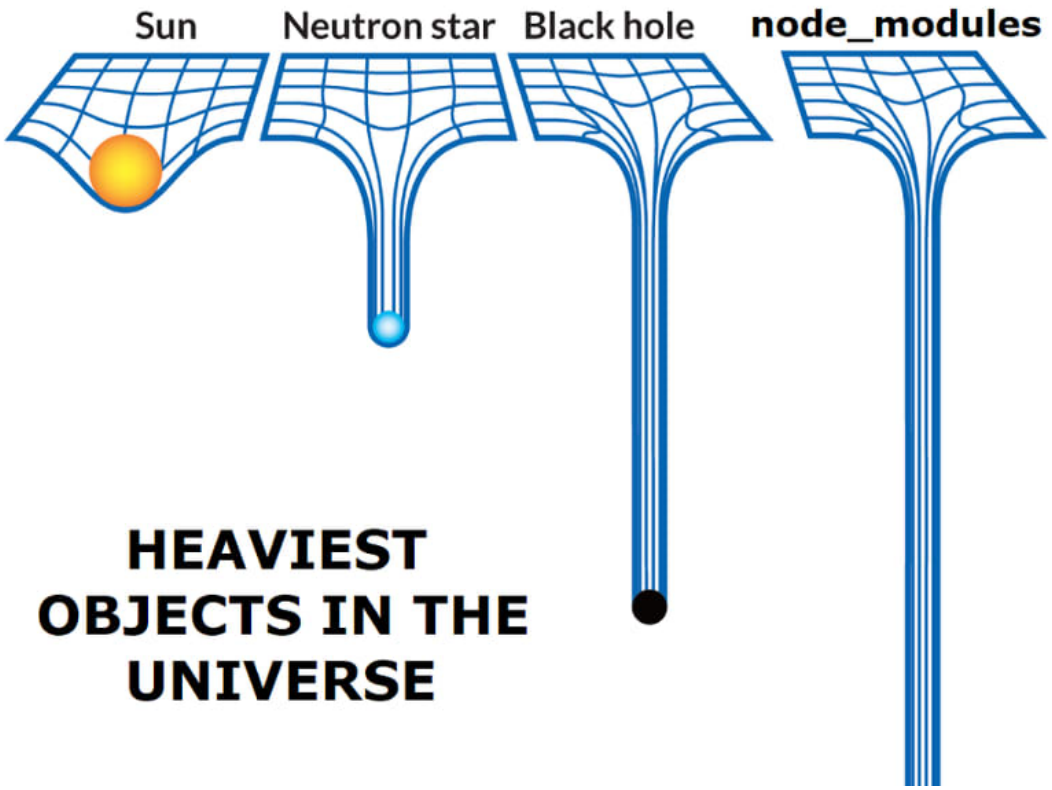
case study: event-stream

- Dominic Tarr, owner of several popular open source libraries, transferred ownership of a library he didn't want to maintain anymore to a "good Samaritan" who offered to take over
- New owner creates a separate module with a virus, then publishes a new version of `event-stream` that depends on the infected module, then *changes the tagged version in GitHub* to remove the malicious dependency.
- So even a code inspection wouldn't have found any problems.
- Now lots of production backend systems have a virus.

lessons

- Be careful what you depend on.
- Sometimes you get what you pay for.
- Remember that open source maintainers are people too. Check your sense of entitlement.
- You tend to get a lot more dependencies with an *easy* framework-based approach than a *simple*, library-based one.

humor


Sun
Neutron star
Black hole
node_modules
HEAVIEST OBJECTS IN THE UNIVERSE

Outline

- ~~announcements~~
- ~~assignment checkin~~
- ~~tradeoffs of dependencies~~
- **web and mobile app choices**
- layout

Web and mobile app choices

- native mobile apps (for iOS or Android)
- cross-platform mobile app frameworks (e.g. React Native and Flutter)
- progressive web apps
- responsive web apps

Native mobile apps

- con: locked in to platform-specific choices:
  - programming language (Java or Kotlin for Android, Swift or ObjC for iOS)
  - APIs for accessing hardware and displaying things on-screen
  - for iOS, must use MacOS hardware (VMs might be possible)
- con: if you want to reach both Android and iOS devices, must learn both platforms!
- pro: official choices can mean better documentation, support, and user communities
- pro: better access to full capabilities of hardware (e.g. sensors)
- pro: better performance
- pro: often "feels" better for users of that platform

Cross-platform mobile app frameworks

- examples:
  - React Native (Javascript/Typescript; my preference)
  - Flutter (Dart)
  - Cordova (web app packaged into a mobile app)
- con: might be difficult to access device hardware
- con: tends not to feel as authentic compared to native apps
- con: typically worse performance (not a problem for many apps)
- pro: only learn one platform and reach both Android and iOS devices
- pro: some platforms designed to be familiar to devs with certain kinds of experience (e.g. in React Native and Cordova, you're basically doing web app dev)

Progressive Web Apps (PWAs)

- this is a web app that you can "install" as an app on your mobile device
- e.g. mobile Safari on iOS has an "add to home screen" option for a page
- adding certain things to a web app (e.g. a manifest) can make this fairly capable
- pro: no app store membership needed
- con: limited hardware accessibility (limited to web APIs only)
- con: anecdotally, I've had challenges with expired cookies in PWAs on iOS

Responsive Web Apps

- this is a web app that is designed to look good at small screen sizes
- con: no home screen icon available; mobile users must discover your app without the app store and have to bookmark your site or find it some other way
- pro: fairly easy to do
- pro: wide reach: works on any device
- we'll talk more about this later

Web and mobile app choices

- native mobile apps (for iOS or Android)
- cross-platform mobile app frameworks (e.g. React Native and Flutter)
- progressive web apps
- responsive web apps

Outline

- ~~announcements~~
- ~~assignment checkin~~
- ~~tradeoffs of dependencies~~
- ~~web and mobile app choices~~
- **layout**

Why layout?

- you've designed the UI in Figma, and you'll need to implement those designs (starting with A8: the walking skeleton)
- this can be really frustrating if you don't know what you're doing
- doing this using modern approaches is very helpful to those with accessibility concerns
- let's learn some modern tools along the way, to save you time

Layout foundations

- HTML: web document *content*
  - tree of *elements*, e.g. `<div>`, `<body>`, `<ul>`
  - each element can have *attributes*, e.g. `href="http://example.com"`
- CSS: web document *style*
  - a *rule set* includes a selector and one or more rules
  - a *selector* selects which elements are affected by the rules, e.g.
    - `#wrapper` selects the element with an `id` attribute of `wrapper`
    - `.box` selects any element with a `class` attribute of `box`
    - `input[type=text]` selects input elements with a `type` attribute of `text`
  - a *rule* contains a property and a value, e.g. `background-color: green;`

For those not creating web apps

- I'm focusing on layout for web apps today
- Cordova, PWAs, and responsive web apps all use these technologies directly
- Some concepts relate strongly to other platforms
  - React Native requires the use of flexbox layout only
- Native mobile apps and custom frameworks like Flutter might not use them
- In such cases, I suggest focusing on:
  - general paradigms for how layout can be controlled
  - general techniques for using modern tooling to explore layout options

Layout outline

- Display types
- Positioning elements
- Flexbox layout
- Grid layout
- Responsive design

Layout outline

- **Display types (e.g. block, inline, float)**
- Positioning elements
- Flexbox layout
- Grid layout
- Responsive design

Normal flow

- Browsers have a default location for each element
- Depends on window dimensions
- Called *normal flow*
- Two types: block and inline

Block elements

- *Block elements* stack vertically
- They take up all available horizontal space
- Examples: paragraphs, headers, lists
- CSS: `display: block;` to force displaying as a block

Inline elements

- *Inline elements* flow left-to-right and wrap to new lines
- They affect *how* stuff is displayed, but not so much *where*
- Examples: italic, bold, code, links
- CSS: `display: inline;`

Demo

Demo 1: Normal Flow

- bring up dev tools with F12 or right-click -> inspect element
- "inspector" tab shows HTML markup and CSS style rules
- (note: I'm using Firefox, but Chrome and others are similar)
- block elements take up all horizontal space; their width is determined by the window width
- inline elements affect style but not position
- inline elements can even be broken over lines
- change an inline element to `display: block;`

Floating elements

- Some elements should not be block or inline
- Example: a graphic in a magazine article
- Text or other inline elements flow around it
- You can float an element to the left or right
- CSS: `float: left;` or `float: right;`
- Floats don't contribute directly to size of parents

Dependence on window dimensions

- Different window sizes can be surprising
- Or different amounts of text
- You can set an element to "clear" the float
- CSS: `clear: left;`
- Can clear left, right, or both (typically use `both`)

Demo

Demo 2: Floats

- `body` element has a border, and the 2nd float overflows the body's area
- how many lines of text wrap around the float depend on float size
- try `float: right`
- swap paragraph order to see ugly stacked floats
- add `clear: both` to make 2nd float clear the previous one (but text doesn't clear)

Table layout

- Anything can be displayed as a table
- CSS properties:
  - `display: table;`
  - `display: table-row;`
  - `display: table-cell;`
  - `display: table-caption;`
  - `caption-side: bottom;`

Table Layout Example

Demo 3: Table Layout (from MDN)

Layout outline

- ~~Display types (e.g. block, inline, float)~~
- **Positioning elements (e.g. relative, absolute, fixed)**
- Flexbox layout
- Grid layout
- Responsive design

Static positioning

- This is the default
- Just where the browser would place things normally
- Statically positioned elements are considered unpositioned

Relative positioning

- *Relative positioning* offsets an element from its static position
- CSS: `position: relative;`
- Use `top` and `left` CSS properties to specify the offset
- (Can also/instead specify `bottom` and `right`)
- These values can be negative
- Space is still reserved for the element's static position
- Demo 4
  - the special words are positioned relatively
  - space is still reserved for them: note where the subsequent period is

Absolute positioning

- *Absolute positioning* positions an element relative to an ancestor
- Reference frame is nearest positioned (i.e. non-static) ancestor
- CSS: `position: absolute;` and `top` and `left` properties
- Space is not reserved for element's hypothetical static position
- Demo 5
  - red box overlays the other content
  - can use z-index to change the relative ordering
  - no space in normal flow is reserved for the red box
  - can change reference frame to div#wrapper by adding `position: relative` to it

Fixed positioning

- *Fixed position* is when an element stays in a single spot on a screen
- Demo 6

Sticky positioning

- *Sticky positioning* is a combination of static and fixed positioning
- Easiest way to describe this is by showing it
- Demo 7
- Compatibility table (from caniuse.com)

Layout outline

- ~~Display types (e.g. block, inline, float)~~
- ~~Positioning elements (e.g. relative, absolute, fixed)~~
- **Flexbox layout**
- Grid layout
- Responsive design

Flex container and flex items

- Flex layout is opt-in
- CSS: `display: flex;`
- Element with flex display property is *flex container*
- Direct children are *flex items*
- Enables flexible dimensions *in one primary dimension*

Flex direction

- Top-level property on flex container: flex direction
- CSS: `flex-direction: row`
- Valid values:
  - `row`
  - `row-reverse`
  - `column`
  - `column-reverse`
- `row` is default for web (for English); `column` is default for React Native

Main axis and cross axis

- Flex direction determines the *main axis*
- The perpendicular direction is the *cross axis*
- CSS properties affect either the main or cross axis, so are dependent on the flex direction

Demo

Demo 8: Flex Direction

- try all 4 flex-direction values: `{row,column}{,-reverse}`

Justifying on the main axis

- Use `justify-content` CSS property to justify items on the main axis
- Common values:
  - `flex-start` (default)
  - `flex-end`
  - `center`
  - `space-between`
  - `space-around`
  - `space-evenly`

Aligning on the cross axis

- Use `align-items` CSS property to align items on the cross axis
- Common values:
  - `flex-start` (default)
  - `flex-end`
  - `center`
  - `stretch`
  - `baseline`

Flex start and end

- The *flex start* depends on the writing direction
- Default: left and top for English, but also depends on flex direction
- The *flex end* is the opposite side

Demo

Demo 9: Flex Alignment

- try various values for justify-content
- try various values for align-items

Take away questions