How to Build Your Killer App

Jeff Terrell, Computer Science Professor of the Practice

Monday, November 12, 2018

Abstract

These are the notes from a talk delivered during Global Entrepreneurship Week at UNC Chapel Hill on Monday, November 12, 2018. The talk description follows.

If you're working to develop an entrepreneurial idea or venture, you have to keep the modern mode of life in mind: we're online, on the go, all the time. That means that the ability to build a mobile or web app has become critical to supporting your entrepreneurial endeavors. Come hear Jeff Terrell, a UNC-Chapel Hill alumnus, entrepreneur and computer science professor of the practice, walk you through a beginner's guide on how to build an app, including:

- The lifecycle of an app.
- Designing an app.
- The parts of an app.
- Choosing a technology to use.
- Finding and working with developers.

About this talk

Target audience

- you know how important apps are to modern society
- you have an idea for an app, but no idea how to create it
- you don't know how to write a program

Goals

- **not** to teach you everything you need to know
- rather, to transition you
  - from: "I don't know what I don't know."
  - to: "I know what I don't know."
- also, to teach you how to work with developers

Outline

- the lifecycle of an app
- the parts of an app
- designing an app
- partnering with developers
- choosing technologies

The lifecycle of an app

Conception

- what is the idea?
- what need does it meet?
- does it make sense as an app?
- what are your goals? money, fame, improving society?
- akin to a value proposition in entrepreneurship

Design

- what will it look like?
- how will the user interact with it?
- what does each button or widget do?
- more on this later

Minimal Viable Product (MVP)

- lean/agile approach says: get something in front of people ASAP
- focus on the most essential features first
- when essentials are done and the app is usable, ship it (to early adopters)
- then those users can provide valuable feedback

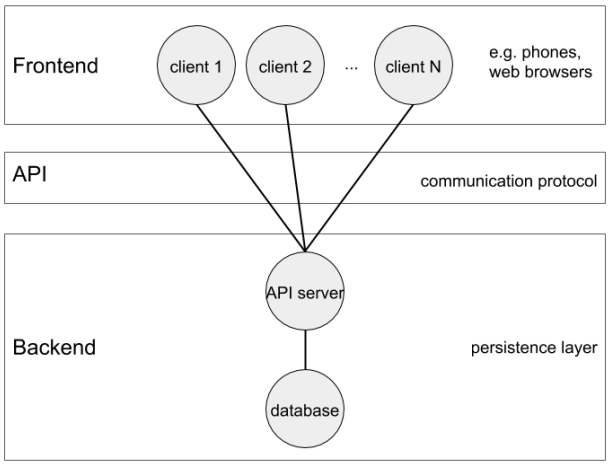
Public availability

- at some point, the app is ready for the public
- needs legit, vetted entries in app stores and a website
- (you'll need memberships in app stores to create entries: $25/life for Google Play, $99/year for Apple App Store)
- you can continue to commission new features, ideally funded by sales

Maturity

- even established apps need some care
- maintenance includes security fixes and keeping pace with new devices and app platforms, e.g. new versions of iOS

The parts of an app



The frontend

- the screens the user sees, and the devices they run on
- basically everything on the user's side of the network
- (a.k.a. "client side", as opposed to "server side")
- devices are outside your control, so can't be trusted
- note: frontend should fail gracefully when network is unavailable

Data storage

- where does the data live?
- some probably lives on the client side
- but most apps need to share data
  - sending messages between users
  - public leaderboards in games
- also, nice not to lose all data and progress when you lose or replace your phone
- so data needs to live in a central, authoritative place

The backend (persistence layer)

- where data is stored, and how it is accessed and manipulated
- lives on the server side of the network
- runs on devices (or virtual devices) that you control and trust
- always on and available (hopefully)

The API

- the frontend and backend need precise rules governing their communication
- these rules are called the API, or *application programming interface*

- API is a set of *API endpoints*, or specific ways of storing and/or retrieving specific types of data
  - e.g. API endpoints for an email app: send an email, retrieve email summaries, retrieve full info for an email, etc.
- note: some APIs are public and can be consumed directly, apart from the frontend
- most APIs use HTTP, the hyper-text transfer protocol, same as web browsers

API servers

- backend typically consists of the core database and >= 1 API server, which mediates access to the database
- why are API servers necessary?
  - to implement the specific interface of the API for clients' use
  - authentication: verifying that a user is who they claim to be
  - authorization: verifying that a user can access what they're asking for

Designing an app

What is it?

output of the design phase:

- a set of pictures of screens
- an indication of how screens connect (e.g. what happens when each button is clicked)

Why design?

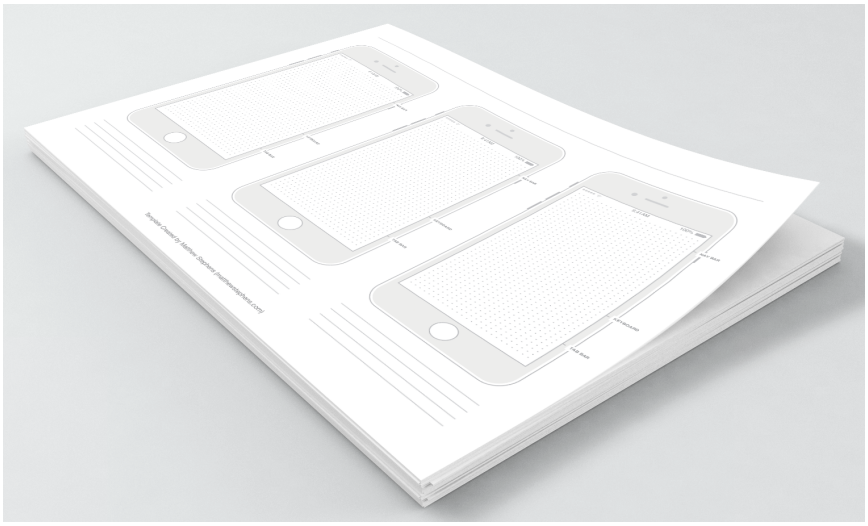
- "until you can write clearly about a topic, you don't understand it." -Fred Brooks
- likewise, until you design your app, you don't understand it
- designing reveals weaknesses, deficiencies, and inconsistencies

- also, designs are a powerful, precise tool for communicating your vision
  - difficult to overstate how important this is when engaging with developers

Approaches

Approaches => Hand-drawn sketches

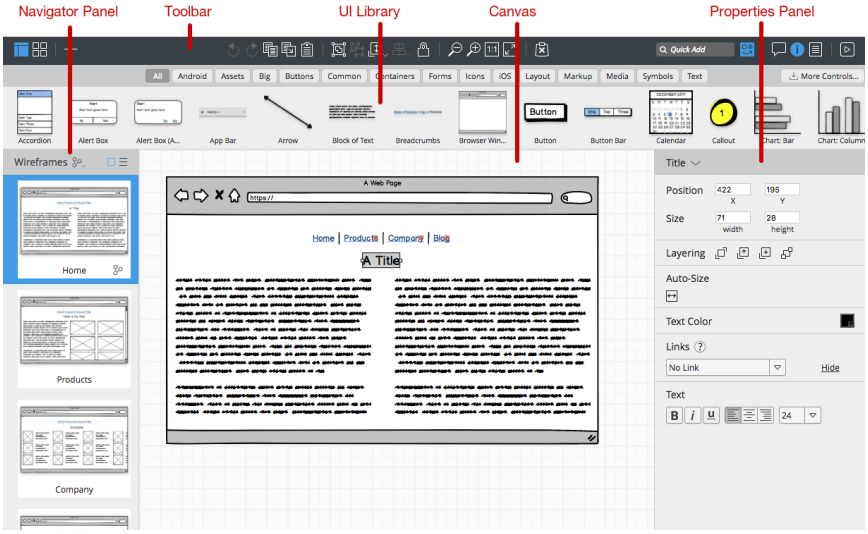
- on a whiteboard, draw rectangles for your screens
- fill in rectangles with what's on a screen
- or use a template, like this one:



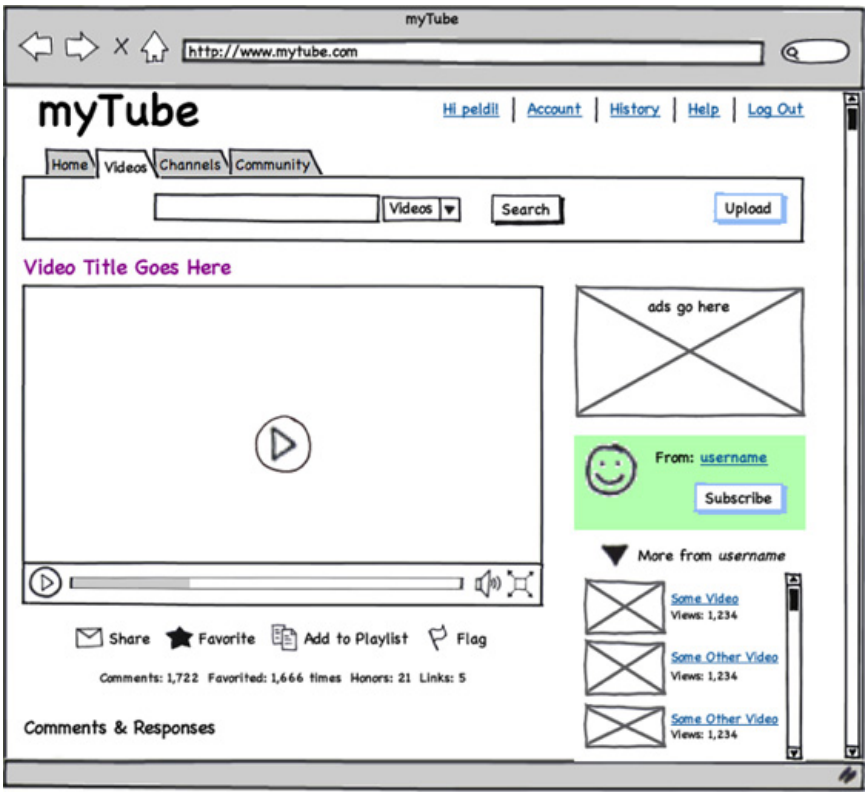
(source: `http://matthewstephens.com/`)

Approaches => Wireframing

- wireframe: a low fidelity digital design, sometimes in black-and-white
- a popular tool for this is Balsamiq (.com)



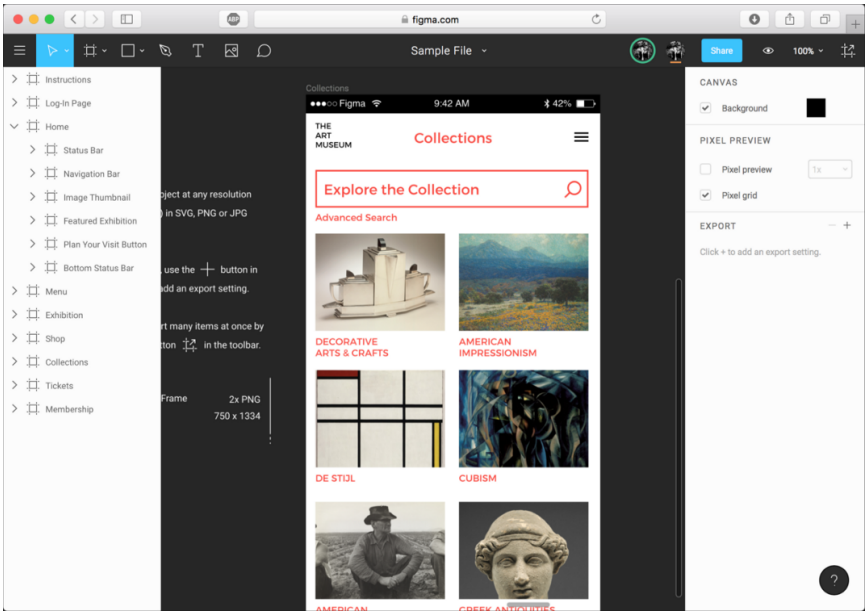
(from `https://docs.balsamiq.com/cloud/overview/`)



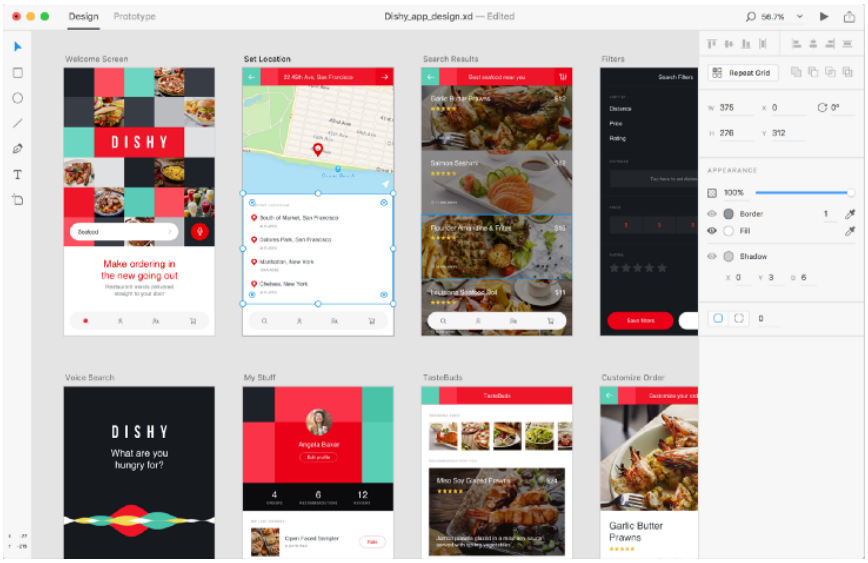
(from `https://www.mightybytes.com/blog/wireframing_techniques_for_a_peacefull_development_process/`)

Approaches => Pixel-perfect designs

- goal: screens are intentional down to finest detail of spacing and color
- note: usually only specified for a particular screen size, so not fully prescriptive
- very helpful level of clarity for development, but a dev will also need to know e.g. which fonts were used
- tools include Sketch, Figma, and Adobe XD



(From `https://blog.prototypr.io/2018-list-of-interface-prototyping-tools-7f1472dfcf`



(From `https://blog.prototypr.io/2018-list-of-interface-prototyping-tools-7f1472dfcf`

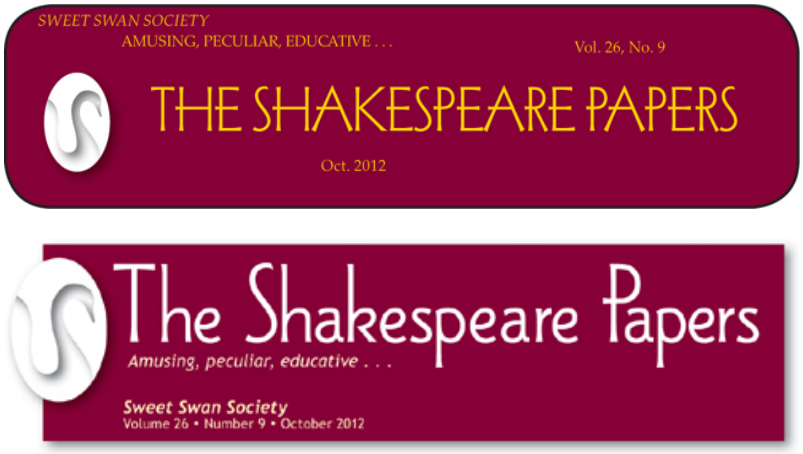
Clickable prototypes

- upload screen designs (images), then make regions clickable, linked to other screens
- provides a more realistic experience of what the app will be like
- Marvel (marvelapp.com) does this
- also Sketch and other tools can do this

Principles

Proximity

- items related to each other should be near each other
- items get visually grouped into larger units
- purpose: to organize the content
- it becomes easier for users to know where to look
- Example of proximity



(From *The Non-Designer's Design Book*, third edition, by Robin Williams)

Alignment

- every element should have a **visual connection** with some other element
- . . . even if they're far apart
- purpose: to unify and organize
- don't center-align items
- rarely use more than one alignment on a page
- Example of alignment

(From *The Non-Designer's Design Book*, third edition, by Robin Williams)

Repetition

- another way to create a visual connection between related items
- purpose: unify and add visual interest
- akin to a woman wearing a black dress and various red accessories

Contrast

- make different things visually different
- purpose: organize and create interest (attract the eyeballs)
- examples: fonts, colors, sizes, spacing, etc.
- "don't be a wimp": either be consistent or clearly different; mere similarity causes confusion
- Example of contrast

hugs

a dog bakery
gallery • daycare

We know you love your four-legged friends,
So hike on over to ***Hugs*** for shows of affection!

Sterling Silver Jewelry…perfect for braggin' on your pup
Toys Galore…for making your dog feel special
Custom art prints of you and your furry friend
Dog calendars, books, and mouse pads
Figurines and statues of all breeds
First Aid Kits

Friday, July 11 and Saturday, July 12
Receive a FREE mini snack pack of *Hugs* puppy cookies
with any hiking gear purchase!

"Somebody needs a Hug!!!"
Hugs, where biscuits, beds, and books beckon

503 OLD DOG TRAIL, MADRAS, OR, 99909
TELEPHONE: (505) 555-1212 FAX: (505) 555-1212

(From *The Non-Designer's Design Book*, third edition, by Robin Williams)

Resources

- *The Non-Designer's Design Book* (Third Edition) by Robin Williams
  - available online via library.unc.edu
- *Don't Make Me Think* (Third Edition) by Steve Krug
  - available via UNC libraries

Partnering with developers

Why partner with a developer?

- you know *what* you want but not *how* to create it
- developers know *how* to create apps but not *what* you want
- the more effective your collaboration, the better the app will be and the faster it will be done

Developer motivations

- money
- equity (typically more junior)
- experience [in a particular technology]
- to serve and help you out

Pitfalls of working with developers

Misunderstood features

- **problem:** what gets built isn't what you wanted
- maybe dev didn't listen carefully enough
- or maybe you didn't say it carefully enough
- or maybe your words meant different things to you vs. the dev
- detailed screens help a lot with this problem!

Scope creep

- **problem:** you realize you want more or different than what you originally asked for
- can be frustrating for developer
- they might have to throw away their work
- you might have to renegotiate payment terms with the dev
- nevertheless, quite common: very few visionaries are good enough to see all the angles up front

Wrong estimates

- **problem:** dev's estimates for when things would be done turned out to be wrong
- it's very challenging to accurately predict how long these things take (even for senior devs)
- also challenging for people to admit they were wrong
- when dev bills by the hour, you bear the risk; when they bill by the project, they bear the risk

Ghosting

- **problem:** the developer becomes unresponsive
- maybe the dev realizes they're in over their head
- or maybe they have given up hope of collaborating with you
- regardless, they are willing to cut their losses and bolt
- . . . or maybe they're just heads-down on a hard problem and don't want to talk to you until they've solved it
- difficult to discern, esp. without clear expectations around communication frequency

Yak shaving

- **problem:** the dev spends too much time getting distracted by easier problems
- devs are often unusually oriented to getting things done
- hard problems can intimidate
- dev can maintain denial and good feelings by solving smaller, possibly irrelevant problems

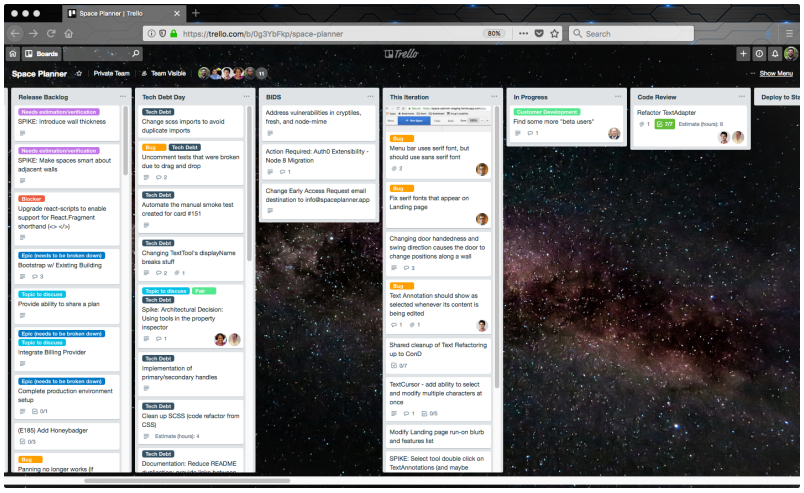
Technical debt

- 'technical debt' occurs when shortcuts are taken in the code, typically under time pressure
- each shortcut is a wart in the code that needs to be worked around
- if too many of these warts accumulate, the project slows down
- how much time pressure is the dev under?

Solutions

Maintain shared project state

- memories are fallible; commit to design artifacts as the authority
- 'design artifacts' include screens and written descriptions of features
- most misunderstandings can be avoided this way
- if you didn't write it down, your fault!
- I recommend Trello as a free, lightweight tool



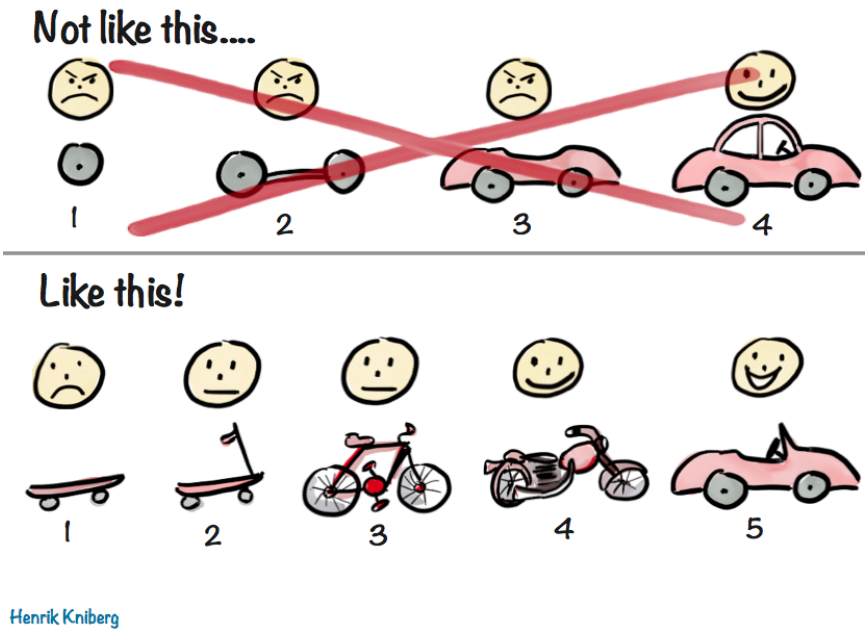
Iterate

- expect *working software* after every iteration (~ 2 weeks)
- dev should demo new features at an 'iteration meeting'

- features might be limited or artifically constrained, but they work end-to-end
- it's not finished until it's delivered
- good iteration meetings build trust and enable feedback loops:
  - on what's being built
  - on the collaboration process

Prioritize

- key: prioritize the most important features first
- iteration meetings are the perfect time to pivot



from *Making sense of MVP* blog post by Henrik Kniberg

Communicate

- key: **communicate frequently and candidly**
- retrospect: the good, the bad, the ugly

- appreciate and praise candid feedback and the courage it requires
- memorize some good questions, e.g. "How much tech debt did you incur to make this demo?"
- be available on short notice when the dev has questions

Finding developers

- network. go to meetups. talk to developers.
- learn where devs hang out (in person and online)
- can post job announcements in places, e.g. Stack Overflow
- consider remote vs. local employees and remote vs. in-person meetings
- consider junior devs (cheap but slow) vs. senior devs (expensive but efficient)

Hiring developers

- typically you'll want a contractor, not an employee
- I strongly recommend paying by the hour, not by the job
  - the job is *never* fully defined and understood up-front
  - you will have major friction trying to change or add anything later
  - estimates are never spot-on, so somebody loses
- rates can range from $10 to $150+ per hour
- can also engage professional firms, if you have the money

Choosing technologies

Web vs. mobile

- web app: like an informational web page, but changing as you interact with it (e.g. Gmail)
- web app advantages:
  - no need to install the app first

  - can use it from computers (and ideally mobile devices too)

- mobile app advantages:
  - good exposure in the app store
  - adds an app icon to phone's home screen
  - access device hardware (e.g. camera, location, accelerometer)

Web Foundations

- browsers only support HTML (content), CSS (style and layout), and Javascript (programming)
- many languages can generate HTML and CSS in a more programmer-friendly language
- many languages can be translated to Javascript

Web Approaches

- "frameworks" offer structure to your app, which can help for larger apps
- Most popular are Angular from Google and React from Facebook
- I don't recommend Angular; IMO it's difficult to learn and use
- I do recommend React; I find it simpler to understand and use
- I also like Redux, which plays well with React

Native mobile apps

- native apps: those written with official tools for that device
- native iOS apps are written in Objective C (old) or Swift (new)
- native Android apps are written in Java (old) or Kotlin (new)
- native apps are the most authentic and the fastest (important e.g. for games)
- but difficult to learn multiple languages and multiple approaches

Cross-platform mobile apps

- Apache Cordova packages up a web app as a native app
- Similar to a window in your mobile browser, but with its own app icon
- Facebook's React Native is similar, but simpler
- I recommend React Native for most new projects

Backend

- Backends can be built in any programming language
- "Frameworks" serve a similar role here: specific organization for code
- Most popular include Rails (for Ruby), Django (for Python), and Express (for Javascript)
- I recommend (a) not using Java and (b) using Clojure, which is simpler to understand and use IMO

The end

- You'll probably need help on this journey.
- Stop by the App Lab in Sitterson 027.
- Come to my office hours (see `https://terrell.web.unc.edu` for a signup link)
- Email me: terrell@cs.unc.edu