Evaluating Technologies;
Choosing a Platform;
Common Application Architectures

by Jeff Terrell
for UNC COMP 523, Software Engineering Laboratory
https://comp523.cs.unc.edu
Course context

- This lecture helps with
  - Assignment 4: application architecture
  - Assignment 5: architecture diagram
- Both are due Sept. 29
- This lecture is about 2 weeks early
- Suggestion:
  - connect with future you, who needs to know this stuff
  - or, come back and watch the recording of this lecture later
The assignments were carefully designed to guide you in the project.
You just finished the project web site (or will soon).
User stories are due at the end of this week.
A clickable prototype is due end of next week.
Then architecture decisions (Assignments 4 and 5) are due the week after.
The big idea

- You have to build an app.
- How do you decide which tools to use?
  - programming language
  - framework
  - database or backend
  - platform (application or infrastructure)
  - build tools, IDEs, libraries, etc.
Roadmap

- Survey and analysis of evaluation approaches
- The virtue of simplicity
- Common web/mobile application architectures
Part 1: Evaluation approaches
Survey and analysis
Recall Assignment 0: Project Preferences.

There were many ways to decide on a project.

- clarity of the proposal
- maturity of the idea
- client flexibility, friendliness, or availability
- social impact
- et cetera
Diverse priorities

So it is with evaluating a technology

- How new is it?
- How popular is it?
- Is it maintained?
- How easy is it to start with?
- Am I already comfortable with it?
- Will it help me get hired after graduation?
- How easy is it to understand?
Newness

Pros

- It can expand your mind
- It can be fun
- It feels cool to be using cutting edge things; you’re a pioneer
- We’re into computers and programming, so we must have some tolerance if not enjoyment of new things

Cons

- There may be bugs
- Documentation may be lacking
- You may be relatively alone using it, so may be hard to get help
Pros

- If others are successfully using this, that implies that you can too
- It can be easy to get help, even for uncommon uses
- Following the crowd tends to be easy

Cons

- Popularity is a cruel and whimsical master
- Many excellent engineers are poor marketers, and their excellent work isn’t popular
- Many excellent marketers are poor engineers, and their mediocre work is popular
Is it maintained?

Proxy measure: when was the most recent commit to the repository?

Pros
- If you discover and report a bug, it may get fixed quickly
- You may get new features or better performance “for free”

Cons
- Maybe bug fixes are still happening because it is buggy software, and there are plenty more bugs not yet found
- You might be coupling your project to one that is constantly moving, and you might also have to stay moving to keep up. (I.e. “free” upgrades usually have an associated cost.)
Ease

Word origin: to lie near; adjacent

Pros

- It’s an easy lift—for you—to use it
- No or little learning curve required
- Start being productive fast

Cons

- Maybe others on your team have a different sense of what’s easy—easy is *relative*
- Maybe a different tool would be worth the learning curve in the end
- Easy to start does not imply easy to finish
Part 2: The virtue of simplicity

What is simplicity
(Much of this content is from *Simple Made Easy*, a talk by Rich Hickey.)

- Word origin: one fold or one braid
Unity of concept, role, or task
Not necessarily just one instance or operation
The idea: lack of interleaving, not cardinality
Opposite: complex, which involves multiple distinct concepts tied together
Simplicity is *absolute*, not relative
Part 2: The virtue of simplicity

Why simplicity is superior
How many concepts can you hold in your mind at one time? (7 ± 2)

Complex things have multiple ideas and take up multiple slots in your attention

Key idea: complexity undermines understanding
What’s true of every bug? It passed the type checker and the tests.

So: understanding is essential to debugging your program.

And complexity undermines this.
A simple tool
A simple tool (mostly)
A complex tool

- Top Cover
- Hose Fill Port Locking Caps
- 12” Removable Pot
- Moisture Sensor
- Tomato Cage Holes
- Water Level Indicator
- 8” Removable Pot
- Controller Housing
- Basin
- Solar Panel and Mount
Another complex tool

Farm Bot - https://farm.bot/
A simple, composable tool
Simple physical tools

- They have a *unity of purpose and of mechanism*
- Therefore, you understand when and how to use them
Complex physical tools

- You must read the manual
- You hope nothing goes wrong! Because debugging requires understanding
- You’re probably not going to attempt to compose a complex tool with anything else
An example: simple or complex?
Riding mower: simple or complex?

- Unity of purpose, but not of mechanism
- Possibly a composition of simple parts
- Not easy (for me) to debug when there’s a problem
- On the other hand, debugging is isolated to a subsystem
Part 2: The virtue of simplicity

Case study: Ruby on Rails
Imagine what you could build if you learned Ruby on Rails...

Learning to build a modern web application is daunting. Ruby on Rails makes it much easier and more fun. It includes everything you need to build fantastic applications, and you can learn it with the support of our large, friendly community.
Ease

- It truly is easy, and people love it
- Helped me get consulting work for many years
- Once you know it, you can develop a web backend rapidly
- "Emphasizing ease gives early speed [but] "ignoring complexity will slow you down over the long haul." -Rich Hickey
There's a lot to know! "Convention over configuration"—but there are a lot of conventions.

It is hard to be productive until you know most of the conventions.

Even when productive, hard to know how things fit together, so can be difficult to extend things in new directions.

The ugly side of Rails: upgrading your app to a new version of Rails.

Lots of breakage. You do have a test suite, right?

Difficult to ask clients to spend money on something they can’t see.
Part 2: The virtue of simplicity

Frameworks vs. libraries
A *library* is code that somebody else wrote that you invoke to help you with some aspect of your program.
- Example: a grid canvas to let you draw lines and points
- Example: a library to play occasional sound effects

A *framework* is code that somebody else wrote that invokes your code
- Examples: Rails, Django, Angular.js, Ember.js, Flutter, React Native, Android, Java Spring, etc.
- frameworks tend to be difficult to understand
- they tend not to be simple
- you can sometimes function well without understanding... until you need to debug something or extend something
- a (single-purpose) library is like a physical tool: you know what it’s for and when to use it
- frameworks are more like starting a new job: you must figure out what’s expected of you, and although you might function OK on day 1, really getting it takes time
Part 2: The virtue of simplicity

State
State is a combination of value and time
It is complex by definition: two things folded together (that aren’t easily separable)
State and OOP

- State is perhaps the very essence of object-oriented programming
- ...and a source of tremendous incidental complexity when used in that way
- In OOP, it’s difficult to consider values and time separately
- General assertion: time is the Achilles’ heel of OOP
State is essential

- Can we get rid of state entirely? Not for interesting programs, no.
- But being able to deal with values and time separately is much simpler.
- This reduces complexity, which decreases bugs and increases understanding.
- A concrete example: Redux.js, which we’ll cover next week.
- Note: many of you would consider state to be easy
- So easy !≠ simple
Part 2: The virtue of simplicity

Dependencies
Core 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

This is another aspect that can inform your technology choices
Azer Koçulu published an open-source library as the \texttt{kik} package on npm.

Kik (company) wanted to use the \texttt{kik} package, and they approached Azer about it.

Azer didn’t want to give up the \texttt{kik} package name.

Kik approaches NPM through their conflict resolution policy about a solution.

NPM transfers ownership of \texttt{kik} to Kik.

Azer retaliates by unpublishing all of his packages from NPM.

One of those packages, \texttt{left-pad}, was a popular dependency, underlying many other packages in the ecosystem.

\texttt{npm install} started failing across the board for every package that has a dependency on \texttt{left-pad@0.0.3}.
**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.
HEAVIEST OBJECTS IN THE UNIVERSE

Sun  Neutron star  Black hole  node_modules
Part 3: Common architectures for web and mobile applications
Dynamic pages

- Most interesting pages in a web or mobile app display data
- The page has to be generated dynamically once the data is fetched
- This can happen server-side or client-side
- Server-side page generation is an older approach
- Client-side page generation involves Javascript running in browsers
- Javascript code puts HTML elements in the page, listens for user events, and communicates with the backend using an API
- Client-side tends to be more complex, but can have a nicer user experience instead of doing a full page reload for every interaction
Backend diagram

client

common checks (e.g. authentication)

router

endpoint1  endpoint2  endpoint3

database
A simpler approach: the router

1. (defn routes [req]
2.   (let [method (:request-method req)
3.           path (:uri req)
4.           method&path [method path]
5.           conn (:datomic/conn req)]
6.
7.   (cond
8.     (= [:get "/csv"] method&path)
9.     (csv/handler (d/db conn))
10.
11.    (= [:post "/update-person"] method&path)
12.    (update-person/handler conn (:params req))
13.
14.    ...
15.
16.    :else
17.    {:status 404
18.    :body (str "uri was " (pr-str (:uri req))))})