Web Development
Web development (at a 10k level)

- Typical web application components
  - Programming language
  - Framework
  - Template system

- Rendering approaches
Programming language

(Note: much of this is my opinion)
Compiled or Interpreted

- Compiled
  - To machine code
    - No run-time needed, fast
    - Good for running web interfaces on IoT devices with limited resources
  - To bytecode
    - Requires run-time VM with matching run-time libraries
  - Longer development cycle?
  - Inability to patch quickly
    - Must recompile entire app (Apache Struts bug and Equifax)
    - Dev and Ops involved to fix security flaws (versus just Ops)

- Interpreted
  - Requires interpreter and all packages application depends upon
    - 2MB Javascript interpreter!
  - Slow, but some languages with good support for JIT
    - Python (PyPy), Javascript (v8, NodeJS)
    - Performance closer to compiled
Typing system

- **Static**
  - Types checked at compile-time
  - Errors caught at compilation *before* deployment
    - Debug then deploy
  - Good for mission (and business) critical applications
- **Dynamic**
  - Types checked at run-time
    
    ```javascript
    function foo(somenum) {
      if (somenum == 3) {
        return '0';
      } else {
        return 0;
      }
    }
    ```
  - Errors
    - Caught at run-time via crashes (Python, JavaScript)
    - Or not caught at all via generation of nonsensical results with type coercion (PHP)
  - Deploy then debug
  - Good for rapid prototyping
Typing system

- Strongly typed
  - Explicit type conversion
    `python -c "print '5' + 8"`
    Traceback (most recent call last):
      File "<string>", line 1, in <module>
      TypeError: cannot concatenate 'str' and 'int' objects

- Weakly typed
  - Implicit type conversion/coercion/casting
    - PHP
      `$ cat math.php
      <?php print('5' + 8); ?>
      $ php math.php
      13`
Typing system

- **JavaScript**
  
  ```javascript
  variable = 3
  variable += 3 // 6
  variable += '3'  // ?
  typeof(variable)  // ?
  ```

- **C**
  
  ```c
  #include <stdio.h>
  int main() {
    char c=0x80;
    printf("%x\n",c);
  }
  ```

  `mashimaro <~> 1:24PM % ./a.out`
  
  `ffffff80`

  But only on x86/Linux since char is unsigned for C on ARM
Typing system

- Type-inferencing
  - Automatic detection of types so programmer doesn't have to declare it
    
```plaintext
define x = 3; <= x inferred to be a number
```
  - Reduces language verbosity
- Kotlin vs. Java
  - Local variable type inferencing added to Java 10 (3/2018)
Functional programming support

- First-class functions
  - Pure functions as objects
  - Functions as an atomic unit of composition
  - Programming pattern that avoids shared mutable state and side-effects
    - Similar to OOP's encapsulation and message passing
  - Helps manage complexity while reducing errors
    - CS 457/557
    - React, Angular, Redux...
Asynchronous support

- Event-driven programming and non-blocking operations
  - Optimizing single-threaded operation for performance
  - Blocking calls automatically yield to other parts of code
Asynchronous support

- **Example: Callbacks**
  - Register a function to be executed upon completion of another
  - **Example**
    - Synchronous file writing
      ```plaintext
      fileObject = open(file) // blocks until file opened
      fileObject.write("We are writing to the file.") // finally can write
      // do the other, totally unrelated things our program does
      ```
    - Asynchronous file writing
      ```plaintext
      // open asynchronously and register writeToFile callback function
      fileObject = open(file, writeToFile)
      // execution continues immediately
      // do the other totally unrelated things that our program does
      // writeToFile executed once open finishes
      ```
  - See skipped slides for animation of implementation in JavaScript

- **Other common mechanisms**
  - Closures, Promises (also see skipped slides)
Concurrency support

- Ability to leverage multi-core processors
- Programming paradigm and support for parallel execution
- Memory consistency model
Ease of development

- Programming ease
- Testing ease
- Library and package management support (maturity of ecosystem)
  - e.g. how much code you *don’t* have to write
- Developer base (for hiring and for answering questions)

Ease of deployment

- Migrating from development to production infrastructure (e.g. reproducibility of execution environment)
- Updating and patching software in packages
  - Can web app be patched in-place or does it require recompilation?
Web programming languages
Java, C#

- Prevalent in e-commerce, bank web sites
  - Pre-date most web frameworks that popularized scripting languages (PHP, Python, Javascript)
- Compiled+Interpreted (bytecode with JIT)
- Statically and strongly typed
  - Preferred for large sites and for critical applications
- Managed memory (garbage collected)
- Asynchronous support in Java (Event interface) and C# (async/await)
- Huge developer base, mature class support, adding conciseness
- But, with deployment
  - Recompilation of apps when security patches to libraries occur
  - Vendor lock-in
    - Rely on Oracle/Microsoft to keep platform libraries secure and deploy features
    - Must buy Microsoft operating systems to run full-blown ASP.NET applications (or rely on Microsoft to keep updating .NET core for Linux)
JavaScript

- Designed for web browsers by Brendan Eich
  - Based on Java (syntax, OOP)
  - Scheme in a browser (functional programming)

- Interpreted, but with fast JIT (v8)
- Dynamically typed (type checking at run-time)
- Weakly typed (type coercion)
- Managed memory (garbage collected)
- Asynchronous from the beginning for single-threaded, event-based operation (callbacks, promises, closures, etc.)
- Ease of development (400k packages for Node.js, front-end and back-end share same language)
- Ease of deployment (npm package management)

- Ideal for smaller web applications requiring quick development iterations and rapid results (IMO)
JavaScript

- Cons
  - Type coercion (weak typing)
    ```javascript
    var num = 0;
    var obj = new String('0');
    var str = '0';
    console.log(num == obj); // true
    console.log(num == str); // true
    console.log(obj == str); // true
    ```
  - Leads to loose equality
    - `==` (equality after coercion rules applied – "loose")
    - `===` (equality without coercion – "strict")
    - `Object.is()` (same-value equality)

```javascript
> const a = {
>     num: 0,
>     valueOf: function() {
>         return this.num += 1
>     }
> };
> const equality = (a==1 && a==2 && a==3);
> console.log(equality); // true
true
```
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>===</th>
<th>!==</th>
<th>Object.is</th>
</tr>
</thead>
<tbody>
<tr>
<td>undefined</td>
<td>undefined</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>&quot;foo&quot;</td>
<td>&quot;foo&quot;</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>{ foo: &quot;bar&quot; }</td>
<td>x</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>+0</td>
<td>-0</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>0</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>===</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>!==</td>
<td>0</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>&quot;0&quot;</td>
<td>0</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>&quot;17&quot;</td>
<td>17</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>[1,2]</td>
<td>&quot;1,2&quot;</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>new String(&quot;foo&quot;)</td>
<td>&quot;foo&quot;</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>null</td>
<td>undefined</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>null</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>undefined</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>{ foo: &quot;bar&quot; }</td>
<td>{ foo: &quot;bar&quot; }</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>new String(&quot;foo&quot;)</td>
<td>new String(&quot;foo&quot;)</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>0</td>
<td>NaN</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>&quot;foo&quot;</td>
<td>NaN</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
JavaScript

• Cons
  • Security issues: supply-chain attacks on npm packages

    Software

    How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript

    Code pulled from NPM – which everyone was using

By Chris Williams, Editor in Chief 23 Mar 2016 at 01:24 167 SHARE ▼

• Sites using packages that are abandoned (no security updates)
• Sites relying on packages taken over by malicious developer
  • https://hackernoon.com/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5
TypeScript

- Weak, dynamic typing leads to software engineering problems
- TypeScript (Microsoft)
  - Typed superset of JavaScript that compiles down to JavaScript
  - Statically typed objects
  - Strongly typed objects (explicit casting)
  - Checked at compile-time
- Similar approaches
  - PureScript
  - Flow (Facebook)
  - AtScript (Google/Angular)
Go (2009)

• Want the best of C/C++
  • Low-level systems programming
  • Bare-metal performance
• Want the best of Java, Python, Ruby, JavaScript
  • Managed, garbage-collected memory
  • Rich package/module support
• Want the best of JavaScript
  • First-class support for asynchrony/concurrency
Go language design

- Designed for simplicity and readability
  - "Simplicity is Complicated" (Rob Pike)
    - [https://www.youtube.com/watch?v=rFejpH_tAHM](https://www.youtube.com/watch?v=rFejpH_tAHM)
  - Features fixed (not trying to be like other languages)
    - Can fit language spec in your head (25 keywords)
    - One canonical way of doing things (unlike Perl or Python)
    - Easy to reason about code

- Multi-developer coding easier
Go

- Strongly and statically typed (e.g. Java)
  - Eliminates run-time type errors and type coercion errors
- Rich built-in types (maps, slices, first-class functions, multiple return values, iterators)
- Type-inferencing to support concise syntax
- Memory safety via managed memory that is garbage collected (use Rust if this bothers you!)
- Efficient compilation
  - Think built-in make to avoid 45 minute compile times
- Rich, built-in module support instead of includes
- Standardized code formatter `gofmt`
  - No more spaces versus tabs, no style guides needed
  - Readability built-in!
Go

- Asynchrony and concurrency
  - Non-blocking, concurrent operation with goroutines
    - Each goroutine mapped onto an underlying OS thread
    - True parallel execution (no global interpreter lock)
    - Concise syntax
  - Shared memory disallowed to support memory-safety
    - Based on research in Communicating Sequential Processes as organizing principle in parallel applications (see CS 415/515?)
    - Communication done via channels (explicit IPC)
    - Can reduce the use of semaphores/mutexes (see CS 533)
Go

- Example

```go
func HeavyComputation(ch chan int32) {
    ...
    ch <- result
}

ch := make(chan int32)

// Create a new goroutine with the same address space
// and use it to run the function. No need to create
// thread, call library to start thread.

go HeavyComputation(ch)

result := <-ch // Blocks if not ready
```
Go

- Static compilation to bare-metal
  - Performance close to C, C++
- Easy to deploy
  - Emits a single portable binary with all dependencies and environment included
  - No versioning issues in deployment
  - Used to build the world's smallest Docker container
    - [https://www.youtube.com/watch?v=zu8NSrNFZ4M](https://www.youtube.com/watch?v=zu8NSrNFZ4M)
- Obviates one of the main reasons for a container
- Being used to build Internet-scale applications
  - Docker, Kubernetes
  - High-performance web APIs to support modern frameworks switching over…
    - Google, Microsoft in particular
    - Django, Rails, NodeJS hard to scale
Go issues

- **Drawbacks**
  - Static typing makes handling arbitrary JSON/XML or implementing an object-relational mapping (ORM) framework for databases difficult
  - No support for exceptions (i.e. try/catch)
    - Done via multi-return
      ```go
      if, err := os.Open("filename.ext")
      if err != nil {
        fmt.Println(err)
        return err
      }
      ```
  - No de-facto web framework to use (yet)
  - Similar issues as statically linked binaries have for updating
Ruby

- Designed for programmer ease
  - Performance and concurrency not a concern
  - Example: Meta-programming support
    - Run-time generation of code
    - Not fast, but programmers love it
- Puppet, Chef (data center, DevOps management software) written in Ruby
- Github, early Twitter in Ruby
Python

- Interpreted, can be compiled to bytecode (.pyc)
- Dynamically typed (type errors caught at run-time)
- Strongly typed (type conversion must be explicit)
- Managed memory (garbage collected)
- Some async support
  - Twisted package for Python 2.7
  - async module for Python 3.6
- Not built with concurrency in mind
  - Global interpreter lock
  - Prevents Python programs from running on multiple cores
- Extensive packages, conciseness, and huge developer base
- But, with deployment
  - Requires tools for package management (pip) and virtual environment management
Web development (at a 10k level)

- Typical web application components
  - Programming language
  - Framework
  - Template system

- Rendering approaches
Web frameworks
Web frameworks

- Library support for building complex web applications
- Tied to a programming language
- Functions
  - Basic routing of URLs to code
  - Support for Model-View-Controller (or alternative) architecture
MVC (Model-View-Controller) architecture

- Developed in 1970s (Smalltalk), adapted everywhere
- First used in web applications in late 1990s via Struts
- Model
  - Code that encapsulates backend application data
  - Data representation and storage
- View
  - Code for rendering that generates HTML output and presents functionality to user (UI)
- Controller
  - Code connecting model and view. Takes in user requests to update model, builds model to supply the view
- Separation of concerns
  - Simplifies swapping out database backend or the UI frontend
MVC

- Not without drawbacks…
  - [https://www.infoq.com/articles/no-more-mvc-frameworks](https://www.infoq.com/articles/no-more-mvc-frameworks)
  - SAM pattern removing controller
Model-View-Presenter (MVP)

- MVC
  - Controller determines which views to pass back based on user actions
- MVP
  - User actions directly bind to specific "presenters"
  - Presenter handles specific interaction between a view and the model
  - Better suited for unit testing
Web development (at a 10k level)

- Typical web application components
  - Programming language
  - Framework
  - Template system

- Rendering approaches
Template system
Template engines

- Web app uses one language (e.g. JavaScript, Python, Java, Go) to produce another language (e.g. HTML)
- Initially, programming language generates entire HTML string
  - Java servlets
- Then, tree-building libraries, APIs to construct DOM/HTML
  - Repetitive, manual construction of pages
- Then, templating
  - Consist of a template language for specifying base page that can be filled in dynamically by web application
  - Template engine generates eventual HTML
  - Examples: JSP (Java Server Pages), Jinja2, Mustache
- Separates presentation (view) from the logic (controller)
Example: Jinja2 syntax

- **Extensible HTML**
  - `{{ arg }}` corresponds to template arguments. We can pass `arg=val` to our template to render
  - `{% %}` encompasses control statements (if, for, block, etc)
    - e.g. `{% if arg %} <h1>arg</h1> {% endif %}`
Jinja2 example

- CTF interface for CyberPDX crypto site
  https://crypto.cyberpdx.org
- "Solve" page consisting of
  - Form containing challenges yet to be solved in drop-down
  - Scoreboard containing shaded challenges already solved

Solve a challenge

```
11
answer
```

Solved challenges

```
01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

```
Controller/Presenter logic for generating lists "solved", "notsolved", "challenges" from model to pass to view upon a "get" to its URL

class Solve(flask.views.MethodView):
    @login_required
    def get(self):
        sols = [f for f in solved.keys()]
        nsol = [f for f in notsolved.keys()]
        return flask.render_template('solve.html',
            solved=sorted(sols),
            notsolved=sorted(nsol),
            challenges=sorted(challenges.keys()))
Solve a challenge</h2>
<form action="{{url_for('solve')}}" method="post">
    <select name="challenge">
        {% for n in notsolved %}
            <option value="{{n}}">{{n}}</option>
        {% endfor %}
    </select>
    <input type="text" placeholder="answer" name="guess" maxlength="20" />
    <input type="submit" value="submit" />
</form>

View implemented in templating engine for drop-down form submission using notsolved

solve.html

..scoreboard table in template using challenges & solved
Controller logic for processing solves (post), using "challenge" and "guess" in form

```python
@login_required
def post(self):
    ...
    username = flask.session['username']
    challenge = flask.request.form['challenge']
    guess = flask.request.form['guess']
    ...
    answer = challenges[challenge]
    if answer == guess:
        flask.flash("Correct. You have solved " + challenge)
    ...
```
Templatting issues

- Now 3 programming languages to learn and debug together!
  - HTML, web app, template
  - Program logic split between template language and web application language
- Example: Conditionally rendering content
  - Why was the "green" condition parsed in Jinja2 and not Python?
  - What about passing the template solved and challenges, then calculate notsolved in template?
Templating via DSLs

- Internal domain-specific languages
  - Templating language integrated with the host language
  - But, need to be careful
    - Want separation of concerns
    - Control logic must be separated from UI logic within language

- Examples
  - JSX (native XML support for JavaScript) (React)
  - Kotlin (JetBrains)
    - Statically typed (i.e. type-safe) HTML builder for JavaScript or Java
  - Elm, Hyperscript, Groovy, Flutter, etc.

https://medium.com/@daveford/80-of-my-coding-is-doing-this-or-why-templates-are-dead-b640fc149e22
## Summary survey

<table>
<thead>
<tr>
<th>Language</th>
<th>Framework</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Java servlets (minimal)</td>
<td>Java server pages</td>
</tr>
<tr>
<td></td>
<td>Spring, Apache Struts (MVC)</td>
<td></td>
</tr>
<tr>
<td>Javascript/NodeJS</td>
<td>Express (minimal)</td>
<td>Mustache, Handlebars</td>
</tr>
<tr>
<td></td>
<td>Sails, AngularJS (MVC)</td>
<td></td>
</tr>
<tr>
<td>Python</td>
<td>Flask (minimal)</td>
<td>Jinja2, Django templates</td>
</tr>
<tr>
<td></td>
<td>Django (MVT)</td>
<td></td>
</tr>
<tr>
<td>C#</td>
<td>ASP.NET routing (minimal)</td>
<td>Razor</td>
</tr>
<tr>
<td></td>
<td>ASP.NET MVC</td>
<td></td>
</tr>
<tr>
<td>Ruby</td>
<td>Sinatra (minimal)</td>
<td>Slim, HAML</td>
</tr>
<tr>
<td></td>
<td>Rails (MVC)</td>
<td></td>
</tr>
<tr>
<td>Go</td>
<td>Echo (minimal)</td>
<td>Amber, Mustache</td>
</tr>
<tr>
<td></td>
<td>Martini, Beego (MVC)</td>
<td></td>
</tr>
<tr>
<td>PHP</td>
<td>Fat-free, Slim (minimal)</td>
<td>Blade, Mustache</td>
</tr>
<tr>
<td></td>
<td>Laravel, Symfony (MVC)</td>
<td></td>
</tr>
</tbody>
</table>
Web development (at a 10k level)

- Typical web application components
  - Programming language
  - Framework
  - Template system

- Rendering approaches
Rendering approaches
Two rendering steps
- Web application renders server data into HTML/CSS for web browser
- Web browser renders HTML/CSS for user

Before, MVC and MVP all running on the server
- User navigation hits web application running on the server each time
- Impacts interactivity

Motivates a push towards client-side rendering and control of web application content (MV*, MVW)
Client-side web UI frameworks

- First instance: AJAX (Asynchronous JavaScript and XML)
  - JavaScript library code for reducing server-side rendering of content
    - Update DOM in place via HTTP request from JavaScript running on page
  - Eventually leads to more view operations placed into browser
    - jQuery, Bootstrap, Meteor, etc.
- Benefits
  - Client-side operations more responsive (no full page reload required)
  - Fewer connections and decreased load on server
  - Can have just the changed portions of page downloaded
- Drawbacks
  - Complexity at the client (package management, bundling, minification, component libraries, cache busting, bundle splitting, automated testing/building)
  - Not good for search-engine optimization
  - A huge number of approaches to choose from
Repetitive server rendering

• Dynamic-ish content
  • What if articles on a web application only change once a day? (e.g. a WordPress blog)
  • Every user request hits server and runs web application rendering code
• Questions
  • Why must server rendering always happen when the results are identical (or largely the same)?
Pre-rendered pages

- Also known as isomorphic or universal pages
- General approach of handling "dynamic" content that does not change
  - Server renders dynamic page into static HTML
  - Static page forward deployed and cached for performance
  - Can be returned to search engines and easily crawled for indexing
  - More secure! (Clients interact with static content)
- Example: WordPress site updated daily
  - Executes web application logic even though results may be the same for the entire day
  - Embedded AJAX potentially left out of search engine results
  - Addressed by pre-rendering and forward deploying site
- Supported now in most client-side frameworks
- Can one pre-render entire application and push it to the edge?
Single-page applications

- Ship web application content and logic all the way to client in a single package
  - Page rendering operations done on client-side
  - Typically, a single index.html file, a CSS bundle, and a JavaScript bundle
  - Examples: GMail, Google Maps, Facebook, Github
    - Via AngularJS, Ember.js, Meteor.js, Knockout.js, React, etc

- Advantages
  - Essential static content delivered ahead of time with no need to refresh (helped by HTTP/2 server push)
  - Faster UI since network calls minimized

- Disadvantages
  - Initial load time for heavy client frameworks
Single-page applications++ (hybrid)

- Latency an issue with single-page applications
- Hybrid approach
  - Serve initial page via traditional server rendering or via pre-rendered version
  - Ship full client-side SPA in the background
  - Seamlessly switch over once SPA downloaded
- Benefits
  - Fast initial load
  - Good with SEO
  - Eventual responsiveness that SPA provides
- Example: Angular Universal
Current-ish state

- Two features
  - Push web app logic, view, and style all to client
  - But that can also run on server (universal apps)
- Main frameworks
  - Angular (Google)
    - MVW (model-view-whatever)
  - React (Facebook)
  - Vue
    - MVVM (model-viewmodel-view) similar to MVP
- Backend only supplies model via an API
  - e.g via REST/gRPC or GraphQL (React)
  - Web app view directly interacts with API
- Complexity leads to type-checking being built into framework (Vue, React) or language (Angular/TypeScript)
What's next?

- Web components?
- All three (Vue, React, Angular) based on modular, re-usable type-checked components
- Component
  - Gets an input, and after some internal behavior / computing, returns a rendered UI template (a sign in / sign out area or a to-do list item) as output
- Web application a collection of re-usable components
- Being standardized via Web Components in W3C
Client-side web UI frameworks

- Which one should you learn?
Client-side web UI frameworks

- Which one should you learn? (2017 survey of startups)

Angular: 217
Ruby on Rails: 184
Express: 170
Django: 76
ASP.NET: 64
ReactJS: 59
Laravel: 28
Flask: 9
Meteor: 7
Spring: 6

crunchbase news