Course Overview

SWE 432, Fall 2019
Web Application Development
Course Topics

• How do we organize, structure and share information?

• How to make applications that are delivered through browsers
  • JavaScript, front-end and back-end development, programming models, testing, performance, privacy, security, scalability, deployment, etc.

• How to design user interactions, focusing on browsers
  • User-centered design, user studies, information visualization, visual design, etc.
Logistics

• No textbook, but suggested readings will be listed on course schedule

• Lab-style work included in many lectures (bring your laptop)
Grading

• 50% Homework
  • 5 assignments, ~2 weeks to do each, all done individually
  • Some code-related assignments will be autograded; you can resubmit an unlimited number of times until the deadline and view your score
  • Also graded by hand for some non-functional issues

• 10% Quizes
  • Pass/fail (Pass if you are in class and submit a quiz, fail if you don’t)
  • Use laptop or phone to complete the quiz in class

• 20% Midterm Exam, 20% Final Exam (not cumulative)
Policies

• My promises to you:

• Quiz results will be available instantaneously in class; we will discuss quiz in real time

• Homework will be graded within 1 week of submission

• Exams will be graded within 1 week
Policies

• Lateness on homework:
  • 10% penalty if submitted UP TO 24 hours after deadline
  • No assignments will be accepted more than 24 hours late
  • Out of fairness: no exceptions

• Attendance & Quizzes:
  • You can miss up to 3 with no penalty
  • Again, out of fairness: no exceptions beyond this
Course Staff

• Prof. Thomas LaToza
  • Office hour: ENGR 4431
    Wed 3:00 - 4:30pm or by appointment
  • Areas of research: software engineering, human-computer interaction, programming tools
Course Staff

• TAs: David Gonzalez
• Office Hours: TBA
Honor Code

• Refresh yourself of the department honor code

• Homeworks are 100% individual
  • Discussing assignments at high level: ok, sharing code: not ok
  • If in doubt, ask the instructor
  • If you copy code, we WILL notice (see some of my recent research results in “code relatives”)

• Quizzes must be completed by you, and while in class
Web Sites vs Web Apps?

Interactive?

User-generated content?

Informational vs fun?
What is the web?

- A set of standards
  - TCP/IP, HTTP, URLs, HTML, CSS, ...
- A means for distributing structured and semi-structured information to the world
- Infrastructure
Perspectives in web development
Systems Perspective

- How can we design **robust, efficient, & secure** interactions between computers?

- Individual web app may run on
  - Thousands of servers
  - Owned and managed by different orgs
  - **Millions** of clients
  - >TBs of constantly changing data

- What happens when a server crashes?
- How do we prevent a malicious user from accessing user data on a server?
Software Engineering Perspective

- How can we design for **change & reuse**?
- Individual web app may
  - **Hundreds of** developers
  - **Millions** of lines of code
  - New updates deployed many times a day
  - Much functionality reused from code built by other organizations
  - Offer API that allows other web apps to be built on top of it
- How can a developer successfully make a change without understanding the whole system?
- What happens when a new developer joins?
Human-Computer Interaction (HCI) Perspective

• How can we design web apps that are **usable** for their intended purpose?

• Individual web app may

  • **Millions of users**

  • Tens of different needs

• What happens when a new user interacts with the web app?

• How can we make a web app less frustrating to use?
Pre-Web

- “As We May Think”, by Vannevar Bush, in The Atlantic Monthly, July 1945

- Recommended that scientists work on inventing machines for storing, organizing, retrieving and sharing the increasing vast amounts of human knowledge

- He targeted physicists and electrical engineers - there were no computer scientists in 1945
Pre-Web - Memex

• MEMEX = MEMory EXtension

• Create and follow “associative trails” (links) and annotations between microfilm documents

• Technically based on “rapid selectors” Bush built in 1930’s to search microfilm

• Conceptually based on human associative memory rather than indexing
Pre-Web - Memex

Never built
Hypertext and the WWW

• 1965: Ted Nelson coins “hypertext” (the HT in HTMML) - “beyond” the linear constraints of text

• Many hypertext/hypermedia systems followed, many not sufficiently scalable to take off

• 1968: Doug Engelbart gives “the mother of all demos”, demonstrating windows, hypertext, graphics, video conferencing, the mouse, collaborative real-time editor

• 1969: ARPANET comes online

• 1980: Tim Berners-Lee writes ENQUIRE, a notebook program which allows links to be made between arbitrary nodes with titles
Origin of the Web

- Became what we know as the WWW
- A “global” hypertext system full of links (which could be single directional, and could be broken!)
Early Browsers
Original WWW Architecture

Links!!
URI: Universal Resource Identifier

URI:  <scheme>://<authority><path>?<query>

http://cs.gmu.edu/~tlatoza/papers.html

“Use HTTP scheme”

Other popular schemes: ftp, mailto, file

“Connect to cs.gmu.edu”

May be host name or an IP address
Optional port name (e.g., :80 for port 80)

“Request ~tlatoza/papers.html”

DNS: Domain Name System

- Domain name system (DNS) (~1982)
- Mapping from names to IP addresses
- E.g. cs.gmu.edu -> 129.174.125.139

![Diagram of DNS domain name space]

The hierarchical Domain Name System for class Internet, organized into zones, each served by a name server.
HTTP: HyperText Transfer Protocol

High-level protocol built on TCP/IP that defines how data is transferred on the web.

HTTP Request
- GET /~tlatoza/papers.html HTTP/1.1
- Host: cs.gmu.edu
- Accept: text/html

HTTP Response
- HTTP/1.1 200 OK
- Content-Type: text/html; charset=UTF-8
- <html><head>...
HTTP Requests

- Request may contain additional header lines specifying, e.g., client info, parameters for forms, cookies, etc.
- Ends with a carriage return, line feed (blank line)
- May also contain a message body, delineated by a blank line

GET request

Host: cs.gmu.edu
Accept: text/html

GET /~tlatoza/papers.html HTTP/1.1

Other popular types:
POST, PUT, DELETE, HEAD

Resource
HTTP Responses

“OK response”
Response status codes:
1xx Informational
2xx Success
3xx Redirection
4xx Client error
5xx Server error

“HTML returned content”
Common MIME types:
application/json
application/pdf
image/png

[HTML data]
Properties of HTTP

- Request-response
  - Interactions always initiated by client request to server
  - Server responds with results

- Stateless
  - Each request-response pair independent from every other
  - Any state information (login credentials, shopping carts, etc.) needs to be encoded somehow
HTML: HyperText Markup Language

HTML is a **markup language** - it is a language for describing parts of a document.
HTML: HyperText Markup Language

• NOT a programming language

• Tags are added to markup the text, encompassed with <>’s

• Simple markup tags: <b>, <i>, <u> (bold, italic, underline)

  <b>This text is bold!</b>

  This text is bold!
Web vs. Internet

**Internet**
- **Link layer**: PPP, MAC (Ethernet, DSL, ISDN, …)
- **Internet layer**: IP, ICMP, IPSec, …
- **Transport layer**: TCP, UDP, …
- **Application layer**: DNS, FTP, HTTP, IMAP, POP, SSH, Telnet, TLS/SSL, …

**Web**
- **Application layer**: HTML, CSS, Browser
- **Transport layer**:
- **Internet layer**:
- **Link layer**:
The Modern Web

• Evolving competing architectures for organizing content and computation between browser (client) and web server

• 1990s: static web pages

• 1990s: server-side scripting (CGI, PHP, ASP, ColdFusion, JSP, …)

• 2000s: single page apps (JQuery)

• 2010s: front-end frameworks (Angular, React, Vue…), microservices
Static Web Pages

• URL corresponds to directory location on server
  • e.g. http://domainName.com/img/image5.jpg maps to img/image5.jpg file on server

• Server responds to HTTP request by returning requested files

• Advantages
  • Simple, easily cacheable, easily searchable

• Disadvantages
  • No interactivity
Web 1.0 Problems

- At this point, most sites were “read only”
- Lack of standards for advanced content - “browser war”
- No rich client content… the best you could hope for was a Java applet
Dynamic Web Pages

HTTP Request
GET /syllabus/syllabi-fall16/SWE432BellJ.html HTTP/1.1
Host: cs.gmu.edu
Accept: text/html

HTTP Response
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

<html><head>...</html><head>...
Dynamic Web Pages

HTTP Request
GET /syllabus/syllabi-fall16/SWE432BellJ.html HTTP/1.1
Host: cs.gmu.edu
Accept: text/html

Web Server

HTTP Response
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

Syllabus Generator Application

There’s a standard mechanism to talk to these auxiliary applications, called CGI (Common Gateway Interface)
Server Side Scripting

- Generate HTML on the server through scripts

```html
<!DOCTYPE html>
<html>
<head>
    <title>PHP Test</title>
</head>
<body>
    <h2>You'll have a luck day!</h2> (<?= num ?>)
</body>
</html>
```

- Early approaches emphasized embedding server code inside HTML pages

- Examples: CGI
Server Side Scripting Site

Browser

Web Server

Database

HTML templates, server logic, load / store state to database

HTTP Request

HTTP Response (HTML)
Limitations

• Poor **modularity**
  
  • Code representing logic, database interactions, generating HTML presentation all tangled
  
  • Example of a Big Ball of Mud [1]
  
  • Hard to understand, difficult to maintain
  
  • Still a step up over static pages!

Server Side Frameworks

• Framework that structures server into tiers, organizes logic into classes

• Create separate tiers for presentation, logic, persistence layer

• Can understand and reason about domain logic without looking at presentation (and vice versa)

• Examples: ASP.NET, JSP
Limitations

• Need to load a whole new web page to get new data

  • Users must *wait* while new web page loads, decreasing responsiveness & interactivity

  • If server is slow or temporarily non-responsive, *whole user interface hangs!*

• Page has a discernible *refresh*, where old content is replaced and new content appears rather than seamless transition
Single Page Application (SPA)

- Client-side logic sends messages to server, receives response
- Logic is associated with a single HTML page, written in Javascript
- HTML elements dynamically added and removed through DOM manipulation
- Processing that does not require server may occur entirely client side, dramatically increasing responsiveness & reducing needed server resources
- Classic example: Gmail
SPA Enabling Technologies

- **AJAX**: Asynchronous Javascript and XML
  - Set of technologies for sending asynchronous request from web page to server, receiving response

- **DOM Manipulation**
  - Methods for updating the HTML elements in a page after the page may already have loaded

- **JSON**: JavaScript Object Notation
  - Standard syntax for describing and transmitting Javascript data objects

- **JQuery**
  - Wrapper library built on HTML standards designed for AJAX and DOM manipulation

JSON

```
{
"firstName": "John",
"lastName": "Smith",
"isAlive": true,
"age": 25,
"address": {
"streetAddress": "21 2nd Street",
"city": "New York",
"state": "NY",
"postalCode": "10021-3100"
},
"phoneNumbers": [
{
"type": "home",
"number": "212 555-1234"
},
{
"type": "office",
"number": "646 555-4567"
},
{
"type": "mobile",
"number": "123 456-7890"
}
],
"children": [],
"spouse": null
}
```

https://en.wikipedia.org/wiki/JSON
Single Page Application Site

Browser

HTML

events

HTML elements

Javascript

HTTP Request

HTTP Response (JSON)

Web Server

Presentation tier

Domain logic tier

Persistence tier

Database
Limitations

- Poor modularity *client-side*
  
  - As logic in client grows increasingly large and complex, becomes Big Ball of Mud
  
  - Hard to understand & maintain
  
  - DOM manipulation is *brittle & tightly coupled*, where small changes in HTML may cause unintended changes (e.g., two HTML elements with the same id)
  
  - Poor reuse: logic tightly coupled to individual HTML elements, leading to code duplication of similar functionality in many places
Front End Frameworks

• Client is organized into separate *components*, capturing model of web application data

• Components are reusable, have encapsulation boundary (e.g., class)

• Components separate *logic* from *presentation*

• Components dynamically generate corresponding code based on component state
  
  • In contrast to HTML element manipulation, *framework* generates HTML, not user code, decreasing coupling

• Examples: Meteor, Ember, Angular, Aurelia, React
Front End Framework Site

Browser

Component presentation
Component presentation
Component presentation
Component logic
Component logic
Component logic
Front end framework

Web Server

Presentation tier
Domain logic tier
Persistence tier

Database

HTTP Request
HTTP Response (JSON)
Limitations

• Duplication of logic in client & server
  • As clients grow increasingly complex, must have logic in both client & server
  • May even need to be written twice in different languages! (e.g., Javascript, Java)
  • Server logic closely coupled to corresponding client logic. Changes to server logic require corresponding client logic change.
  • Difficult to reuse server logic
Microservices

• Small, focused web server that communicates through *data* requests & responses

• Focused *only* on logic, not presentation

• Organized around capabilities that can be reused in multiple context across multiple applications

• Rather than horizontally scale identical web servers, vertically scale server infrastructure into many, small focused servers
Microservice Site

Browser

- Component presentation
- Component logic
- Front end framework

HTTP Request

Web Servers

- Microservice

HTTP Response (JSON)

Database

HTTP Request

- Microservice

HTTP Response (JSON)
Architectural Styles

• Architectural style specifies
  • how to partition a system
  • how components identify and communicate with each other
  • how information is communicated
  • how elements of a system can evolve independently
Constant change in web architectural styles

• Key drivers

  • Maintainability (new ways to achieve better modularity)
  • Reuse (organizing code into modules)
  • Scalability (partitioning monolithic servers into services)
  • Responsiveness (movement of logic to client)
  • Versioning (support continuous roll-out of new features)

• Web standards have enabled *many* possible solutions

• Explored through *many, many* frameworks, libraries, and programming languages