ECE 458
Engineering Software for Maintainability

Introduction to Web Development
Tyler Bletsch
The "back end"

The "front end"

- Can get much fancier (middleware, clustering, proxies, etc.)

Web server

Apache, nginx, etc.

Database server

PostgreSQL, MySQL, MongoDB, etc.

Web browser

(Chrome, Safari, Firefox, etc.)
Part 1: Nuts and bolts

A level of abstraction below where you’ll be working, but you need to know what’s going on
HTTP: Web request/response
(requesting http://example.com/)

Web Browser

Establish TCP connection

HTTP request
GET / HTTP/1.1 ← Request
Host: example.com ← Headers
Connection: close

HTTP response
HTTP/1.1 200 OK ← Response
Content-Type: text/html ← Headers
Content-Length: 1256
Connection: close

<!doctype html> ← Body
<html>
<head>
  <title>Example Domain</title>
  ...
</head>
</html>
HTTP: Web request/response
(requesting http://example.com/)

Request verb: What are we asking of the server?

HTTP request
GET / HTTP/1.1 ← Request
Host: example.com ← Headers
Connection: close

HTTP response
HTTP/1.1 200 OK ← Response
Content-Type: text/html ← Headers
Content-Length: 1256
Connection: close

<!doctype html> ← Body
<html>
<head>
<title>Example Domain</title>
...

HTTP: Web request/response
(requesting http://example.com/)

Request URI: What path on the server do we want?

HTTP request
GET HTTP/1.1 ← Request
Host: example.com ← Headers
Connection: close

HTTP response
HTTP/1.1 200 OK ← Response
Content-Type: text/html ← Headers
Content-Length: 1256
Connection: close

<!doctype html> ← Body
<html>
<head>
    <title>Example Domain</title>
    ...

Web Browser

Web Server
HTTP: Web request/response
(requesting http://example.com/)

Web Browser

Headers: Metadata about our request

HTTP request
GET / HTTP/1.1 ← Request
Host: example.com ← Headers
Connection: close

HTTP response
HTTP/1.1 200 OK ← Response
Content-Type: text/html ← Headers
Content-Length: 1256
Connection: close

<!doctype html> ← Body
<html>
<head>
<title>Example Domain</title>
...

Web Server
HTTP: Web request/response
(requesting http://example.com/)

HTTP request
GET / HTTP/1.1 ← Request
Host: example.com ← Headers
Connection: close

HTTP response
HTTP/1.1 200 OK ← Response
Content-Type: text/html ← Headers
Content-Length: 1256
Connection: close

<!doctype html> ← Body
<html>
<head>
  <title>Example Domain</title>
  ...
</head>

Most common status codes

200: OK
• Success.

301/302: Redirect
• Forward the browser to another URL.

404: Not Found
• Couldn't find the requested thing, bad URL?

500: Internal Server Error
• An error on the server side. Likely cause: your server-side code errored out.

Reference link.
HTTP: Web request/response
(requesting http://example.com/)

Web Browser

HTTP request
GET / HTTP/1.1 ← Request
Host: example.com ← Headers
Connection: close

Web Server

HTTP response
HTTP/1.1 200 OK ← Response
Content-Type: text/html ← Headers
Content-Length: 1256
Connection: close

<!doctype html> ← Body
<html>
<head>
  <title>Example Domain</title>
</head>
...
HTTP: Web request/response
(requesting http://example.com/)

Web Browser

HTTP request
GET / HTTP/1.1
Host: example.com
Connection: close

Web Server

HTTP response
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 1256
Connection: close

<!doctype html>
<html>
<head>
<title>Example Domain</title>
...

Body: Actual data returned

Note: Some types of requests can have a body, too.
Two main purposes of a request

- The browser can load site-authored code written in **Javascript**; this code can itself make web requests.

- This gives rise to two categories of request:
  - The **browser** directly requesting content
    - Done as part of document request process (when you click link)
    - Gets you content that’s consumed by the browser.
    - Data types: HTML, CSS, JS code, images, etc.
    - Example: “Show me the HTML and any content I need to display it”
  - **Javascript on the site** requesting content (often API calls)
    - Could be requesting anything, but usually requesting Javascript-parsable content
    - Data types: JSON data objects, XML data objects
    - Example: “Look up the student list”, “Create a student object”, etc.
    - Often this is done as part of a “**RESTful API**” (covered later)
Request verbs and their semantics

<table>
<thead>
<tr>
<th>Verb</th>
<th>Purpose</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Retrieve data</td>
<td>Must not alter server state; read-only.</td>
</tr>
<tr>
<td>POST</td>
<td>Add/modify data</td>
<td>Request includes a body.</td>
</tr>
<tr>
<td>PUT</td>
<td>Add new data object</td>
<td>Request includes a body.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Remove a data object</td>
<td></td>
</tr>
</tbody>
</table>

There are a few more verbs than this, but they’re usually handled automatically by your framework or aren’t in common use.

Examples of browser-based requests

GET /about.html
- Request the about.html document, which may refer to the image “/logo.png”, the stylesheet “/styles/main.css”, and the javascript “/js/site.js”. The browser would then GET each of those in turn.

POST /contactform
- Send form-based content to the given URL, which would process it and do something. Returns an HTML document.

Note: Browsers do not typically issue PUT and DELETE directly.

Examples of javascript-based requests

GET /students
- Request a JSON list of students, like [{“name”: “Jimmy”, “age”: 14}, …]

PUT /students
- Body is JSON of a student, adds that student to the server’s database

POST /students/251
- Body is JSON of an updated version of student number 251; server commits these changes

DELETE /students/251
- Student 251 is deleted from the server’s database

Read more in general.
Read more with respect to REST.
Types of content

- Remember the **Content-Type** header? Common types:
  - **Hypertext Markup Language (HTML)**: Describes structure and content of a web document, marked up with `<tags>`.
  - **Cascading Stylesheets (CSS)**: Describes *how* the HTML content should be shown (color, spacing, etc.).
  - **Javascript (JS)**: Code to be run in the web browser.
  - **Images (PNG, JPEG, GIF, etc.)**: Pictures
  - **Javascript Object Notation (JSON)**: A text-based record format
    - Has plain numbers, strings in “quotes”, lists in brackets `[1,2,3]`, and dictionaries in braces `{“key”: “value”, ...}`
    - Lists/dictionaries can nest, so you can represent whole data structures.
  - **Extensible Markup Language (XML)**: An older text-based record format. Uses `<tags>` like HTML, but customizable. Thankfully dying.
Examples of types of content

- HTML
- CSS
- JS
- JSON
- XML
- JPEG
Types of HTTP

- This shouldn’t matter a ton day-to-day, but will come up when *deploying* your webserver

HTTP Versions:
- HTTP 1.1: Classic. Still most common.
- HTTP 2: Newer standard, more efficient.

HTTPS: HTTP Secure
- Applies *certificates* and *encryption* to ensure confidentiality & integrity
- Will skip details here, but you’ll need to set it up
- Only tricky bit: you need a **certificate** for your site
  - Rolling your own server? [Let's Encrypt](https://letsencrypt.org) can do this for you
  - Using a cloud service? If they can’t do this for you, they’re clowns
Knowing who you’re talking to

- HTTP is anonymous by default: “Some rando sent a request”
- We want **authentication**: recognition of a specific user
- How to identify distinct users?
  - **Cookies**: Server can ask browser to include a bit of text in every subsequent request
    - Example: `SESSION=123`
    - Server remembers what this means (see next slide)
  - **Storage**: Javascript provides **SessionStorage** (kept as long as the tab is open) and **LocalStorage** (kept indefinitely)
    - Not automatically sent by browser to server, but can be included when Javascript makes a request
    - Can achieve authentication, also other stuff
      - When doing authentication, use **LocalStorage** (closing a tab shouldn’t log you out)
What to store in cookies / storage?

- Classic: If using cookies, you can just have some kind of session identifier, and have the server keep a list of those
  - Cookie: "SESSION=123"
  - Database: “Session 123 was created when user ‘bob’ logged in”
  - Conclusion: “This is bob”

- Modern: Store a JSON Web Token (JWT) in a cookie or LocalStorage
  - Server can provide a bit of JSON that is cryptographically signed by the server (can’t be tampered with)
  - Can be used to hold login info
  - Example: {“logged_in_user”: “bob”}
  - Server need not remember what it means, because client cannot fabricate it
Even lower level: networking details

Q: What is the IP address for example.com?
A: It's 26.3.2.1

Domain Name System (DNS) translates hostnames to IP addresses.

The Host header allows the same web server to handle multiple domains. This may or may not come up in deployment, so I wanted to let you know.
Web application architecture
Types of web applications

- Two main types of web applications:
  - *Content generation*, AKA “**Server-side rendering (SSR)**”:
    - Browser makes requests for HTML, server renders HTML that addresses the request
    - Google Search works this way
  - *API driven*, AKA “**Client-side rendering (CSR)**”:
    - Browser loads static HTML content and JS code
    - Javascript in browser makes data requests of the server; javascript updates the document displayed in the browser, thus showing new content
    - Google Maps works this way
    - The API in this case is often a **RESTful** API (it follows certain rules and design patterns for simplicity)
Content generation example: Google Search

The form data hits the server, server generates HTML response

Simple HTML form

```html
<form action="/search" method="GET">
  <input name="q" type="text">
  <input type="submit" value="Google Search">
  <input type="submit" value="I'm Feeling Lucky">
</form>
```

Simple HTML results

```html
<a href="https://www.petfinder.com/dog-breeds/">
  List of Dog Breeds | Petfinder
</a>
```
API driven example: Google Maps

- Document is generated and updated dynamically, calls server as needed

Click and drag the map?
- Java script repositions image tiles, updates browser view
- Can load landmark metadata as JSON, can load new map tiles as images
Hybrid approaches

• You can mix the two approaches. Example:
  • Google homepage has a classic form for search
  • Also has a menu widget with my face on it

[Diagram of Google homepage with a classic search form and a menu widget with the user's face]
About API-driven sites

- On API-driven sites, tons of software abstraction is used
- Examples:
  - Entities on page are **widgets**, often created and managed as separate software modules
  - Calls to server-side API are often wrapped in abstraction
    - You don’t “issue an request for a JSON object”, rather “this table has its data sync’d to the server’s” (lots of magic hidden behind the scenes)
- You typically use client-side **frameworks** for this, like React
Don’t break the browser!

• A common pitfall of API-driven sites: **breaking URLs** and **breaking the back button 😞**

• Once Javascript is rendering your page content rather than it being loaded as HTML from the server, it’s easy to forget that URLs are useful and should still work!

*Good site: Google Maps*

*Bad site: Sakai*

Panning/zooming changed the URL 😊

Two different assignments, same URL. Can’t bookmark, back button breaks, etc. 😞
Asynchronous web programming

- For API-driven sites, much of interaction is **asynchronous**
  - Doesn’t happen in sequential order, but rather as several requests, each with their own delays

- This can make code nasty. Common solutions:
  - **Callbacks**: Your code provides a function that another module calls back when something happens (such as data being ready)
  - **Promises**: An object representing a request in progress, will either resolve to the complete answer or be rejected as an error
  - **Async/await**: Javascript semantics to help with using promises; can make async code look more like regular sync code

- We won’t go deeper now, but [this article covers it well](#).
Where does data live?
Client server topology

Web browser

Web server
Apache, nginx, etc.

Database server
PostgreSQL, MySQL, MongoDB, etc.

Our focus now

The “front end”

The “back end”

• Can get much fancier (middleware, clustering, proxies, etc.)
Storing data server-side

- Most data we care about is all about relationships
  - This ISBN refers to a book
  - This book is in this purchase order and that sales reconciliation
  - All these purchase orders of that book add up to this sum
  - Etc.

- Two main approaches:
  - **Relational database**: Tables of data expressing relationships between entities. Enforces constraints. Speaks **Structured Query Language (SQL)**.
  - **NoSQL database**: JSON documents expressing similar facts. May or may not enforce constraints.

[Read more about the tradeoffs](#)
• Tables often have **primary keys** (a column that uniquely identifies each entry)

• Records can refer to primary keys of other tables, this is a **foreign key**

• Using these relationships, can describe whole situation with no data duplication

• When tables are created, relationships and constraints are expressed.
NoSQL database example: MongoDB

- Stores JSON **Documents** in **Collections**
- Not focused on relationships
- Faster setup, more flexibility
- Can usually add constraints with add-on middleware

```
{
  first_name: 'Paul',
  surname: 'Miller',
  cell: 447557505611,
  city: 'London',
  location: [45.123, 47.232],
  profession: ['banking', 'finance', 'trader'],
  cars: [
    { model: 'Bentley',
      year: 1973,
      value: 100000, ... },
    { model: 'Rolls Royce',
      year: 1965,
      value: 330000, ... }
  ]
}
```
DO NOT USE DATABASES DIRECTLY! No need for that in the modern age!

Use a framework with an **Object Relational Mapper (ORM)**
- Automatically converts object-oriented operations to database operations
- Applicable to both SQL and NoSQL

Example: Django’s ORM

```
from datetime import date
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def __str__(self):
        return self.name

class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()

    def __str__(self):
        return self.name

from blog.models import Blog
b = Blog(name='Beatles Blog', tagline='All the latest Beatles news. ')
b.save()

b5.name = 'New name'
b5.save()

>>> Entry.objects.filter(pub_date__lte='2006-01-01')
```

Amount of SQL written: zero!
Bring it all together
Big picture

- Most of this stuff you will **not** worry about day-to-day!
- Why? You will use a **framework** that abstracts most of it

- Ideal development mindset: only program the **unique** things about your problem, avoid boilerplate (stock common code)
Example stacks

- Classic Python web app:
  - Back end: **Python** + **Flask** framework + **PostgreSQL** database
  - Front end: Custom HTML+CSS and standalone Javascript modules

- Modern Python web app:
  - Back end: **Python** + **Django** framework + **Django-REST** module + **PostgreSQL** database
  - Front end: **Javascript** + **React** UI framework

- Modern Javascript web app:
  - “MERN Stack”: **MongoDB**, **Express**, **React**, **NodeJS**
  - Back end: **NodeJS** + **Express** framework + **MongoDB** database
  - Front end: **Javascript** + **React** UI framework

- Modern Ruby web app:
  - Back end: **Ruby** + **Rails** framework + **PostgreSQL** database
  - Front end: **Javascript** + **Angular** UI framework

**IMPORTANT!!** These are just examples! Other combos are possible! There are fine technologies not listed! Do your own research!
My thoughts on “tech stacks”

- DANGER: Web development “thought leaders” are notorious trend chasers. Biases you’ll see:
  - **Tons of client-side Javascript**
    - If you’re making Google Maps, you need this (complex data representation needs complex and responsive UI)
    - If you’re making DukeHub, you may not (listing “which classes are there” can be done by simple server-generated HTML)
  - **Dependency mania**
    - Every little thing you might want to do: get a library for it!
    - Hidden cost: learning the library, hitting limitations of the library, continually updating the library, things breaking when the library changes, etc.
    - Also, dependencies are transitive!
  - **New and shiny**
    - It’s easy to make a framework *look* good in a hello-world demo (and it gets you clicks)
    - *Living* with the choice long-term is harder
    - If you pick something, finding support for it is CRITICAL, so popular is often wise (even if the thing itself is sub-optimal)
Homework: Do some research

• **Watch** the videos linked on the course site

• **Build something** with one or more things above (or something else) – be ready to show it off!

• Come to class with **opinions** (positive or negative) on the above videos and the technologies you experimented with

All of the above is due next week, see course site for details