Deployment, Caching, Web Security, and Monitoring

CIS 1962 (Winter 2026)
March 19th, 2026

Penn Engineering

Lesson Plan

5	Deployment
16	Optimization and Caching
27	Website Security
36	Defenses and Best Practices
50	Website Monitoring

Logistics

- Homework 6 due tonight!
- Homework 7 released, due 4/5 (After Passover)
  - Continuation of HW6, now making a full-stack application!
  - You'll be given a front-end, connect it to a backend
- Final Project details out next week 👀
  - Start thinking about what kind of website (frontend and backend) you might want to make!

Review Activity

https://edstem.org/us/courses/91614/lessons/161916/slides/951992

Let's review some content from the previous lecture before we start!

Deployment

How do we make the web applications we build publicly available to the world?

From Development to Production

So you've finally finished development, and you want to ship your web app for people to use. What are your options?

From Development to Production

Development environments are where you write, test, and debug code. Often in web development, we work on a localhost server on local machines with test data to make sure our apps work.

Production environments are live settings where real users get to interact with the application. At this point, code should be deployed with stable infrastructure, secure backends, and optimized performance.

Deployment

Deployment takes our source code used in the development and puts it in a publicly accessible server. With web applications, this often involves tying it to a domain name users than then access in a browser.

There are many common deployment options for websites:

- Static Site Hosting
- Shared Hosting
- Cloud Servers
- Platform as a Service

Deployment Workflow

1. Develop your app locally, make sure it's bug free and well-tested
2. Build your app, either locally or handled by the server host
3. Upload build files or source code repository to a server host
4. Set up app configurations, like environment variables, database credentials, domain name, and more
5. Your site goes live!

Static Site Hosting

Examples: Netlify, Vercel, Github Pages

Static Site Hosting is designed to host pre-built files (HTML/CSS/JS and other assets) with no constantly-on backend.

Services like Netlify and Vercel allow serverless functions and API endpoints that run only in response to HTTP requests to allow a backend/database connections to exist.

Shared Hosting

Examples: Bluehost, SiteGround, DreamHost

Shared Hosting features multiple websites sharing a server. Clients may get a slice of disc space, memory, and CPU on that server on an assigned server.

You can manage files and resources on dedicated dashboards, and have support for PHP and MySQL.

VPS/Cloud Servers

Examples: AWS, DigitalOcean, Azure VMs, Google Cloud

Virtual Private Servers(VPS) or Cloud Servers are hosting options where you rent a server or provision a cloud instance and host your application there.

This option provides the most control, but requires you to setup every part of your project yourself via SSH into your server.

Platform as a Service (PaaS)

Examples: Heroku, Render, Railway

Platform as a Service (Paas) is a hosting option that includes a server, but the platform itself handles the server management, including network, scaling, and security.

This option keeps your apps scalable, but may become expensive and less flexible since you have less control over the servers.

Demo: Vercel

https://vercel.com

Let's try to deploy to Vercel!

Don't worry, it's literally just a few buttons

(so long as you have a project Git repo)

Resources

- Vercel (Recommended for simple deployment cases, also very compatible with Next.js!)
- Render (Where HW4's backend was deployed)
- Netlify
- DreamHost
- Heroku
- Google Cloud

Optimization and Caching

What are some ways to improve the optimization and performance of our apps?

Code Splitting/Lazy Loading

Many modern build tools support **code splitting** out of the box. This improves performance and cuts down on initial load times.

Additionally React supports **lazy loading**, where certain resources (like components) are only loaded when they are needed.

```
const Page = React.lazy(() => import('./Page'));
```

React Suspense

React Suspense is a way to handle asynchronous loading states.

If a child of <Suspense> suspends while loading, a fallback component is rendering, making it easy to display a loading state for async tasks (like long fetches)

```
import React, { Suspense } from 'react';

<Suspense fallback={<div>Loading...</div>}>
  <LazyComponent />
</Suspense>
```

Data Caching

It's often helpful to temporarily store/cache data when developing in JS, so that it may be served faster from a cache rather than needing to be fetched or computed again.

- In-Memory Cache (runtime variables)
- localStorage/session storage
- IndexedDB
- Cookies
- HTTP Caching/Service workers

localStorage & sessionStorage

localStorage and sessionStorage are synchronous key-value pair stores that help with data persistence.

localStorage persists across browser usage and after reloads, while sessionStorage only persists within a browser tab.

Their storage maxes out at around ~5 MB.

localStorage Example

Storing and retrieving user information

```
const user = { name: 'Voravich', age: 57 };

// Save to localStorage
localStorage.setItem('user', JSON.stringify(user));

// Retrieve from localStorage
const storedUser =
JSON.parse(localStorage.getItem('user'));
console.log(storedUser.name);

// Remove an item
localStorage.removeItem('user');

// Clear everything
localStorage.clear();
```

sessionStorage Example

Tracking multi-part form progress

```
const formProgress = { step: 2, data: { email: 'voravich@example.com' } };

// Save to sessionStorage
sessionStorage.setItem('form', JSON.stringify(formProgress));

// Retrieve
const storedForm = JSON.parse(sessionStorage.getItem('form'));
console.log(storedForm.step); // 2

// Remove an item
sessionStorage.removeItem('form');

// Clear all
sessionStorage.clear();
```

IndexedDB

Browsers provide an asynchronous data storage that can store a lot of data (100+ MB) called IndexedDB.

It is often used in offline-first applications and large datasets.

```
const request = indexedDB.open('MyDatabase', 1);

request.onupgradeneeded = (event) => {
  const db = event.target.result;
  const store = db.createObjectStore('users', {
keyPath: 'id' });
};

request.onsuccess = (event) => {
  const db = event.target.result;

  const tx = db.transaction('users', 'readwrite');
  const store = tx.objectStore('users');
  store.add({ id: 1, name: 'Alice', age: 25 });

  const getRequest = store.get(1);
  getRequest.onsuccess = () => {
    console.log(getRequest.result); // { id: 1, name:
'Alice', age: 25 }
};
};
```

Cookies 🍪🍪🍪

Cookies are small data (max ~4 KB) that is stored on the browser, that get sent with every HTTP request to the server.

This makes them especially useful for authentication and user tracking/preferences.

```
document.cookie = "theme=dark; path=/; max-age=" + 30*24*60*60; // 30 days

// Read cookie
const theme = getCookie('theme');
console.log(theme); // "dark"

// Apply preference
if (theme === 'dark') {
  document.body.classList.add('dark-mode');
}
```

Why ask to use Cookies?

Cookies often get used to track personal data while browsing. Sites need your consent to track this data to comply with laws.

These cookies often let you define what kinds of cookies are applied to you, such as cookie for preferences, targeted advertising, tracking, and performance.

React Specific Optimizations

memo: usually prevents re-renders if props have not changed

```
import { memo } from 'react';

const SomeComponent = memo(function SomeComponent(props) {
 // ...
});
```

useMemo: Hook often used to reduce expensive recalculations

```
const value = useMemo(() => computeExpensive(), [dependencies]);
```

useCallback: Memorizes a function to prevent recreation on every render

```
const handleClick = useCallback(() => { console.log('Button clicked');}, []);
```

Website Security

What are some common threats to websites and how do we prevent them?

The Importance of Security

Nowadays, websites have to handle a lot of sensitive user information (personal information, payment details).

Security breaches on websites happen daily, with large data breaches causing companies legal, financial, and reputational harm.

Many vulnerabilities come from unsafe practices when writing JavaScript that malicious actors than exploit.

Common Attacks: XSS

Cross-Site Scripting, or XSS, allows attackers to inject JavaScript into web pages in a way that is visible to other users, allow them to steal sensitive data, hijack accounts, or spread malware.

This could take the form of:

- Reflected XSS: unvalidated user input includes a runnable script

```
https://site.com/?q=<script>alert(1)</script>
```

- DOM XSS: Client-side JS allows browser to process malicious input

Demo: Inject Some Code

```
<body>
    <script>
        document.body.innerHTML += "<div>" + decodeURIComponent(location.hash.substr(1)) + "</div>";
    </script>
</body>
```

Serve up this HTML locally, then visit:

[host]/#hello

and

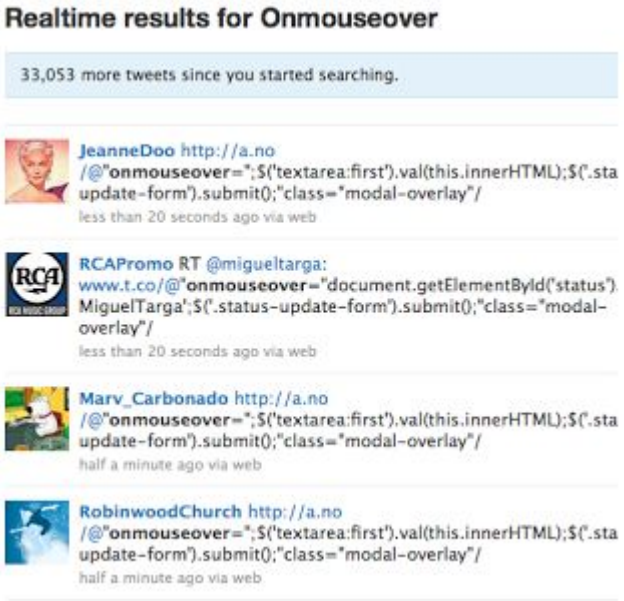
[host]/#<img%20src=x%20onerror=alert('XSS')>

Case Study: Twitter and XSS

[Article]

Due to improper input sanitation with URLs in tweets, people were able to write JavaScript to be embedded into tweets themselves.

People wrote worms (self-replicating malware) that caused tweets to be sent automatically as people viewed infected tweets!



Common Attacks: Session Hijacking

Session Hijacking, or **sidejacking**, is an attack that exploits unsecure session handling. Unprotected session cookies and authentication tokens can be stolen to impersonate users on websites.

This attack can be prevented by using an HTTPS (HTTP Secure) protocol on all pages of a website, not just login. Most website hosting platforms like Heroku and Vercel already provide HTTPS.

Padlock means your are secure/on HTTPS!

Case Study: Firesheep

Article

Firesheep was a Firefox browser extension that was able to capture session cookies transmitted over unsecured Wi-Fi networks (like public Wi-Fi) and allow hijackers to impersonate users on websites with their login credentials.

Threats like these are why you are warned about signing into public Wi-Fi networks or are encouraged to use a VPN!

Common Attacks: CSRF

Cross-Site Request Forgery (CSRF) allows attackers to send requests to real websites (if credentials are available and a session cookie is still active) from false, malicious websites.

Many modern sites protect against CSRF by blocking cookies on cross-site POST requests (cookie attribute: SameSite=Lax/Strict), while frameworks like Express also have CSRF tokens to secure cookie-based authentication.

Case Study: TikTok CSRF

The Bug Report

A certain TikTok API endpoint allowed someone to change the password on users who signed up with third party apps via CSRF. Combined with a reflected XSS bug in a URL parameter, this essentially allowed a "one-click account takeover."

Class Activity

https://edstem.org/us/courses/91614/lessons/161916/slides/952001

Defenses and Best Practices

How do you best defend against threats to web applications?

Defenses and Best Practices

- Input Validation and Sanitization
- Content Security Policy
- Secure Use of Cookies
- Use JavaScript Frameworks' built-in security features
- Keep your dependencies up to date, detect vulnerable dependencies, and fix/remove/replace them

Validation & Sanitization

Ensuring that any user input cannot be interpreted as code will prevent XSS from occurring.

Validation: Ensure that inputs match the types and format you want, for instance emails match a specific regex.

Sanitization: Remove potentially dangerous characters or markup, like HTML tags

- **Escaping**: turning special characters that could be interpreted as HTML (<, >, “) into text forms (&lt, &gt).
- Frameworks (like React) may help by auto-escaping characters!

Validation Example

```
document.getElementById('greeting').innerHTML = "Hello, " + userInput + "!";
```

Can XSS if userInput is: `<script>alert("XSS")</script>`

```
document.getElementById('greeting').textContent = "Hello, " + userInput + "!";
```

No XSS if `.textcontent` is used

```
return <div>Hello, {userInput}!</div>;
```

React auto-escapes

Content Security Policy

A **content security policy (CSP)** is a browser feature that restricts what resources a page can load, like scripts.

This helps catch XSS and other injection attacks by disallowing unsafe scripts.

CSP Example

The CSP for this slide deck (on Google Slides) is as follows:

```
require-trusted-types-for 'script';
report-uri https://csp.withgoogle.com/csp/docs-tt;
base-uri 'self';
object-src 'self' blob:;
report-uri https://docs.google.com/presentation/cspreport;
script-src 'nonce-jxHw5dBcS6GrK_YnPYLpsw' 'unsafe-inline' 'strict-dynamic'
https: http: 'unsafe-eval';
worker-src 'self' blob:;
```

This blocks common unsafe code like innerHTML, eval, (Trusted Types), whitelists certain scripts (script src), and reports CSP violations to Google.

Same-Origin Policy & CORS

Website origin: Combination of a websites protocol (HTTP/HTTPS), hostname, and port.

`https://example.com:443`	!=	`https://example.com`
`http://example.com`	!=	`https://example.com`

By default, browsers have a **Same-Origin policy**, which blocks cross-origin requests. You have to enable Cross-Origin Resource Sharing (CORS) in order to allow your server's resources to be shared with others.

Browser Cookies

Browser Cookies are used to session-based data on the web.

Safe cookie usage often uses certain security flags (or have safe defaults:

- `HttpOnly`: JS can't access this cookie, protects from theft during XSS
- `Secure`: Cookie is sent through HTTPS(encrypted connections)
- `SameSite`: Controls whether browser sends cookies on cross-site requests, either Lax (on top-level navigation) or Strict (never). Prevents CSRF

Dangerous Code/APIs

`eval()` is evil!!!!

This function lets you run code from a string, which is extremely vulnerable to XSS attacks.

Additionally, APIs like localStorage and sessionStorage are easily accessible with JS with little protection, meaning that you should avoid keeping sensitive data in those structures.

Security Tools

Dependencies in your package.json may contain vulnerabilities and security flaws that attackers scan for as targets. It's important to keep your dependencies updated, or be aware of these vulnerabilities!

`npm audit` is a command that will detect and fix vulnerabilities.

Platforms like https://snyk.io/ can help scan for vulnerabilities in your GitHub repos with a wide database of vulnerabilities.

Demo: npm audit

```
npm i react-syntax-highlighter@15.6.1
```

This dependency is an older version (Current 16.1.0) that includes a vulnerability. `npm audit` will catch this.

```
Tested 29 dependencies for known issues, found 1 issue, 1 vulnerable path.


Issues to fix by upgrading:

  Upgrade react-syntax-highlighter@15.6.1 to react-syntax-highlighter@16.0.0 to fix
  ✗ Arbitrary Code Injection [Low Severity][https://security.snyk.io/vuln/SNYK-JS-PRISMJS-9055448] in prismjs@1.27.0
    introduced by react-syntax-highlighter@15.6.1 > refractor@3.6.0 > prismjs@1.27.0
```

Example: SHA1-Hulud

In September and November 2025, an npm supply chain attack compromised many npm packages.

Malware was hidden inside trusted npm packages that were hijacked, executed malicious code during the pre-install step, and gave attackers access to the dev/build environment.

Article

Impacted packages

Resources

- OWASP 2025 Top 10 (Common Security Threats)
- MDN Web Security Docs
- Web Security Academy
- Google Web Fundamentals- Security
- https://snyk.io/

Website Monitoring

What's the importance of monitoring and in what ways can we monitor user activity in apps?

Monitoring

Web app development doesn't end at deployment. It is important to monitor user actions and app performance, and also detect suspicious activity that could lead to attacks.

- Front-end Logging
- Monitoring Services
- Performance metrics for optimization
- User activity data for UI/UX research

Logging in the Frontend

You should already be familiar with being able to `console.log()` on the frontend of your app. You may also send your logs to a third-party service or a backend.

For monitoring purposes, logging helps with:

- Logging app errors
- Logging user actions (button click, navigation)
- Performance metrics (page load times)

Monitoring Services

Monitoring services connected to your app provide a clean dashboard for errors and logs. These may also help automate your monitoring, see trends, and output alerts to services like Slack or email.

Performance Metrics

A deployed app may differ wildly when used by real users from a development environment.

Below are some important metrics you can test:

- First Contentful Paint (FCP)
- Largest Contentful Paint (LCP)
- Cumulative Layout Shift (CLS)
- Interaction to Next Paint (INP)

Google's `web-vitals` package covers all these!

User Analytics

Monitoring the behavior of users, such as where they click and how long they stay on your website, helps developers see patterns and prioritize chances to improve user experiences.

Things you may monitor include:

- Navigation, Clicks, and Scroll Depth
- Session length and bounce rates (how many people immediately leave instead of clicking more links)
- Interactions like video playing, image carousel use, etc.

Google Lighthouse Audit

Google Lighthouse is a service that provides quick and easy analysis of performance metrics and various other metrics, like accessibility and SEO.

[Audit for our Class Website]