# Web Science Application Development

Jason Kuruzovich




October 8, 2025

# Part 2: O

Jason Kuruzovich, PhD
Associate Professor of Business

Feel free to reach out for help

Book a meeting:
https://bit.ly/jason-rpi

Rensselaer Polytechnic Institute
kuruzj@rpi.edu
518-698-9910

# Next Parts of Class

**Core Engineering Pillars**

- **Authentication & Authorization**
- **Testing** (Unit & Integration)
- **Deployment** (Architecture & CI/CD Pipelines)
- **Security Hardening** (DOS/Rate Limiting)
- **Agentic Systems** (LangChain & Tooling)

**In Parallel, We Will:**

- Deepen our understanding of **React fundamentals and patterns**
- Compare and evaluate **modern web frameworks** (e.g., Flask)
- Connect architectural decisions to real-world tradeoffs

# Authentication Learning Objectives

- By the end of class, students should be able to:
- Explain the difference between **Authentication vs Authorization**
- Describe **row-level security (RLS)** and why it matters
- Implement basic email/password auth with Supabase
- Enable OAuth login (Google)
- Enforce per-user data isolation using RLS
- Explain tradeoffs between JWT/session auth and OAuth

# Readings

## Industry Standards & Best Practice Guides

- 1. OWASP Authorization Cheat Sheet
- 2. WorkOS — OAuth and JWT Best Practices
- 3. Google Developers — OAuth2 Best Practices
- 4. Supabase Authentication

# What is the difference between authentication and authorization?

# Authentication vs Authorization

- Authentication = Who are you?
- Authorization = What are you allowed to do?

Example:

- Login to Netflix (auth)
- Only see your profile & watch history (authorization)

# Authorization: Identify and protect critical system actions

- Examples:
- Create purchase order
- Add vendor
- Issue refund
- Change pricing
- Modify user roles
- Export customer data

These require:

- Strong authentication
- Explicit authorization rules
- Often additional controls (logging, approvals, MFA)

# Core Concept: Identity

- User logs in → system issues token (JWT or session)
  - Token sent with every request
  - Backend verifies token
  - Database may enforce identity (RLS)

# What Is a JWT?

- **JSON Web Token (JWT):** a compact, URL-safe token for transmitting claims
- Structure:
- **Header** — algorithm & token type
- **Payload** — claims (user ID, role)
- **Signature** — cryptographically signed data
-

- Verified by server without storing session state

# How do we securely store passwords?

# Password Hashing with bcrypt

- **Never store passwords in plain text!**
- Hashing = one-way function (cannot reverse)
  - bcrypt adds a random salt automatically
  - bcrypt is intentionally slow (cost factor)
- **Flow:**
  - Registration: password → bcrypt.hash() → store hash
  - Login: password + stored hash → bcrypt.compare() → true/false
- **Why not SHA-256?**
  - Too fast — attackers can brute-force billions/sec
  - No built-in salt — vulnerable to rainbow tables

# Access Tokens vs Refresh Tokens

- **Access Token**
  - Short-lived (15 min – 1 hr)
  - Sent with every API request
  - Contains user claims (ID, role)
  - If stolen, limited damage window
- **Refresh Token**
  - Long-lived (days – weeks)
  - Used only to get a new access token
  - Stored securely (httpOnly cookie or server-side)
  - Can be revoked to "log out" a user

# Token Refresh Flow

- 1. User logs in → server returns access token + refresh token
- 2. Client sends access token with each API request
- 3. Access token expires (e.g., after 15 minutes)
- 4. Client sends refresh token to /api/refresh
- 5. Server verifies refresh token, issues new access token
- 6. If refresh token is expired/revoked → user must log in again
- **Benefits:**
  - Users stay logged in without re-entering credentials
  - Compromised access tokens expire quickly
  - Server can revoke refresh tokens (unlike stateless JWTs)

# How does OAuth fit in?

# OAuth 2.0: Delegated Authentication

- "Let users log in with Google/GitHub instead of a password"
- **Key Players:**
  - Resource Owner — the user
  - Client — your app
  - Authorization Server — Google, GitHub, etc.
  - Resource Server — API that holds user data
- **Flow (Authorization Code):**
  - App redirects user to provider's login page
  - User approves → provider sends auth code to your app
  - App exchanges code for access token (server-to-server)
  - App uses token to fetch user profile

# JWT vs Session-Based Auth

- **Session-Based:**
  - Server stores session in memory/database
  - Client gets a session ID cookie
  - Server looks up session on each request
  - Easy to revoke (delete the session)
- **JWT-Based:**
  - Server does NOT store state — token is self-contained
  - Client stores token (localStorage, cookie, memory)
  - Server verifies signature on each request
  - Hard to revoke before expiry (need blocklist)
- **Trade-off: JWTs scale better; sessions revoke easier**

# Middleware: Protecting Routes

- Middleware = function that runs before route handler
- **Authentication middleware pattern:**
  - Extract token from Authorization header
  - Verify token signature and expiration
  - Attach user info to request object
  - Call next() to proceed, or return 401/403
- **401 Unauthorized — no token or invalid credentials**
- **403 Forbidden — token present but invalid/expired**
- Apply per-route: app.get('/secret', authMiddleware, handler)

# Security Best Practices

- **Passwords:**
  - Always hash with bcrypt (cost factor >= 10)
  - Never log or return passwords in API responses
- **Tokens:**
  - Keep access tokens short-lived
  - Use HTTPS in production (tokens travel in headers)
  - Store secrets in environment variables, not code
- **General:**
  - Validate all input (username, password length)
  - Rate-limit login attempts to prevent brute force
  - Use generic error messages ("Invalid username or password")

# Row-Level Security (RLS)

- Problem: even authenticated users shouldn't see other users' data
- **RLS = database enforces per-user data isolation**
- How it works:
  - Database policy checks user ID on every query
  - SELECT * FROM todos → only returns YOUR todos
  - Even if app code has a bug, DB blocks unauthorized access
- Supported by: Supabase (Postgres), Oracle, SQL Server
- **Defense in depth — don't rely solely on app-level checks**

# Lab 6: JWT Authentication Hands-On

- Build a REST API with Express + JWT from scratch:
  - Set up Express server with health check
  - Implement user registration with bcrypt
  - Implement login that returns access + refresh tokens
  - Create middleware to protect routes
  - Add refresh token endpoint
  - Test everything with curl
- **Automated grading: push to GitHub → 8 tests run automatically**
  - Check results in the Actions tab

# How do we know our code actually works?

# Types of Tests

- **Unit Tests**
  - Test a single function in isolation
  - Example: does bcrypt.hash() produce a valid hash?
  - Fast — run hundreds in seconds
- **Integration Tests**
  - Test components working together
  - Example: does POST /api/register actually store a user?
  - Slower — may need a running server or database
- **End-to-End (E2E) Tests**
  - Test the full user flow from client to server
  - Example: register → login → access protected route
  - Slowest — but catches real-world bugs

# The Testing Pyramid

- Many unit tests (fast, cheap, isolated)
- Fewer integration tests (moderate speed, test contracts)
- Few E2E tests (slow, expensive, high confidence)
- **Why a pyramid?**
  - Unit tests catch most bugs quickly
  - Integration tests verify components connect correctly
  - E2E tests confirm the system works as a whole
- **Anti-pattern: "ice cream cone" — too many E2E, too few unit**

# Testing in This Lab

- **Our autograder uses integration tests via curl**
  - Each test sends an HTTP request and checks the response
  - Tests run sequentially: register → login → use token
  - This is exactly how you'd test a real API
- **What the 8 tests cover:**
  - Health check (server is running)
  - Registration (happy path + duplicate handling)
  - Login (valid credentials + wrong password)
  - Protected route (no token, valid token, garbage token)
- **Key idea: tests document expected behavior**
  - If the tests pass, the API contract is met

# Automated Testing & CI/CD

- **CI = Continuous Integration**
  - Every push triggers automated tests
  - Catch bugs before they reach production
  - GitHub Actions runs our autograder on every push
- **CD = Continuous Deployment**
  - If tests pass, deploy automatically
  - If tests fail, block the deployment
- **Why this matters:**
  - Manual testing doesn't scale
  - Automated tests give confidence to ship fast
  - Tests are a safety net — not a replacement for good design

# Additions: BCRYPT VS Others

## Comparison Table

| Algorithm | Memory Hard | GPU Resistant | Tunable | Modern Recommended |
|---|---|---|---|---|
| SHA-256 | ❌ | ❌ | ❌ | ❌ |
| PBKDF2 | ❌ | ⚠️ Moderate | ✔ | ⚠️ Limited |
| bcrypt | ❌ | ✔ | ✔ | ✔ (still good) |
| scrypt | ✔ | ✔ | ✔ | ✔ |
| Argon2id | ✔✔ | ✔✔ | ✔✔ | ⭐ Best choice |

“Argon2 is the newest algorithm among these and is currently considered to be the strongest password hashing algorithm available. It is designed to be memory-hard, meaning that it requires a lot of memory to compute. This makes it difficult for attackers to use specialized hardware like GPUs and ASICs to crack passwords. Argon2 has several parameters that need to be kept in mind when using it, such as the amount of memory to use and the number of iterations to perform.”

https://www.reddit.com/r/cryptography/comments/11tqci2/argon2_vs_bcrypt_vs_scrypt_vs_pbkdf2/

## Argon2

Argon2 is a password-hashing function created by by Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2 is in the public domain and can be downloaded from GitHub. Argon2 was declared the winner of the Password Hashing Competition (PHC). There were 24 submissions and 9 finalists. Catena, Lyra2, Makwa and yescrypt were given special recognition. The PHC recommends using Argon2 rather than legacy algorithms.

There are three versions of Argon2. Argon2d provides the highest resistance against GPU cracking attacks. Argon2i is designed to resist side-channel attacks. Argon2id is a hybrid version. It follows the Argon2i approach for the first half pass over memory and the Argon2d approach for subsequent passes.

# Access + Refresh Token Implementation (Explicit Server + Client Code)

- Built on Lab 6 JWT Authentication

- Goal:
- Access Token → React memory
- Refresh Token → HTTP-only cookie (server controlled)

# Full Architecture Overview

- SERVER:
- • Issues access token (short-lived)
- • Issues refresh token (long-lived)
- • Stores refresh token in HTTP-only cookie

- CLIENT:
- • Stores access token in React state
- • NEVER stores refresh token in JS

# SERVER: /api/login (Issues Both Tokens)

```
const accessToken = jwt.sign(payload, JWT_SECRET, { expiresIn: '15m' });
const refreshToken = jwt.sign(payload, REFRESH_SECRET, { expiresIn: '7d' });

// Store refresh token server-side (optional DB/in-memory)
refreshTokens.push(refreshToken);

// THIS stores refresh token in browser cookie
res.cookie('refreshToken', refreshToken, {
  httpOnly: true,
  secure: true,
  sameSite: 'strict',
  maxAge: 7 * 24 * 60 * 60 * 1000
});

res.json({ token: accessToken });
```

# What Actually Stores the Refresh Token?

- THIS LINE on the SERVER:
- res.cookie('refreshToken', refreshToken, { httpOnly: true })

- Browser receives Set-Cookie header →
- Browser stores cookie automatically

- React NEVER sees the refresh token.

# CLIENT: Login Code

- const res = await fetch('/api/login', {
- method: 'POST',
- headers: { 'Content-Type': 'application/json' },
- credentials: 'include', // allows cookie to be set
- body: JSON.stringify({ username, password })
- });

- const data = await res.json();
- setToken(data.token); // Access token in memory

# SERVER: /api/refresh

```
app.post('/api/refresh', (req, res) => {
  const token = req.cookies.refreshToken;
  if (!token) return res.sendStatus(401);

  jwt.verify(token, REFRESH_SECRET, (err, user) => {
    if (err) return res.sendStatus(403);

    const newAccessToken = jwt.sign(
      { username: user.username },
      JWT_SECRET,
      { expiresIn: '15m' }
    );

    res.json({ token: newAccessToken });
  });
});
```

# CLIENT: Silent Refresh on Page Load

- useEffect(() => {
- fetch('/api/refresh', {
- method: 'POST',
- credentials: 'include' // sends cookie automatically
- })
- .then(res => res.json())
- .then(data => setToken(data.token));
- }, []);

- Browser automatically attaches refresh cookie.

# Protected Route Call (Client)

- fetch('/api/secret', {
- headers: {
- Authorization: `Bearer ${token}`
- }
- });

- Access token explicitly sent in header.

# Handling Expired Access Tokens

- If /api/secret returns 403:
- 1. Call /api/refresh (cookie auto-sent)
- 2. Receive new access token
- 3. Retry original request

- Refresh token never exposed to JS.

# Logout Flow

- CLIENT:
- setToken(null);
- fetch('/api/logout', { credentials: 'include' });

- SERVER:
- res.clearCookie('refreshToken');
- revoke token in DB/in-memory store

# Security Model Summary

- XSS Protection:
- Refresh token cannot be read by JavaScript

- CSRF Protection:
- sameSite='strict' on cookie
- Access token sent manually in header

- Access token short lifetime limits damage window

# Backend as A Service

- One alternate way is to have a full backend as a service
  - Firebase https://firebase.google.com
  - Supabase https://supabase.com
  - AWS Amplify https://aws.amazon.com/amplify/

  What are advantages of backend as a service providers?

  What are the drawbacks?

# AWS Amplify

- https://github.com/RPI-WS-spring-2026/amplify-vite-react-template

# Backend as a Service (BaaS) + TypeScript Overview

- Context: React + AWS Amplify Quickstart

- What is BaaS?
- Firebase | Supabase | AWS Amplify

- Plus: What is TypeScript?

# What is Backend as a Service (BaaS)?

- BaaS = Cloud-managed backend infrastructure

- Instead of building your own backend:
- • Auth handled for you
- • Database hosted for you
- • APIs auto-generated
- • File storage included

- You focus on frontend + business logic

# Core Services Provided by BaaS

- 1. Authentication (email, OAuth, MFA)
- 2. Database (SQL or NoSQL)
- 3. Auto-generated APIs
- 4. File storage
- 5. Hosting + CI/CD
- 6. Security rules & access control

# Firebase (Google)

- Database: Firestore (NoSQL, document-based)
- Auth: Built-in providers
- Cloud Functions + Hosting

- Strengths:
- • Very fast to start
- • Strong real-time support

- Tradeoff: NoSQL structure, vendor lock-in

# Supabase

- Open-source Firebase alternative
- Database: PostgreSQL (SQL, relational)
- Auto-generated REST + GraphQL APIs

- Strengths:
- • Real SQL database
- • Open-source flexibility

- Tradeoff: Slightly more setup

# AWS Amplify

- Built on AWS services (Cognito, AppSync, DynamoDB, S3)
- Strong React integration

- Provides:
- • Authentication (Cognito)
- • GraphQL / REST APIs
- • Storage (S3)
- • Hosting + CI/CD
- Enterprise-ready and scalable

# How Amplify Works in React

- Install Amplify libraries
- Configure backend via CLI or Amplify Studio

- Example:
- import { Amplify } from 'aws-amplify';
- import config from './amplifyconfiguration.json';
- Amplify.configure(config);

# What is TypeScript?

- TypeScript = JavaScript + Static Types

- Developed by Microsoft
- Adds compile-time type checking
- Compiles into standard JavaScript

- Improves safety + developer experience

# Why TypeScript in Amplify?

- Amplify generates typed models
- Better IDE autocomplete
- Catches errors before runtime

- Example:
- function greet(name: string): string {
-  return 'Hello ' + name;
- }

# JavaScript vs TypeScript

- JavaScript:
- let age = 25;

- TypeScript:
- let age: number = 25;

- Prevents: age = 'twenty five'; ❌

# When to Use BaaS vs Custom Backend

- Use BaaS when:
- • MVP / startup
- • Small team
- • Need rapid deployment

- Use custom backend when:
- • Complex business logic
- • Deep infrastructure control required

# Summary

- BaaS = Managed backend infrastructure

- Firebase → Fast, NoSQL
- Supabase → SQL, Open-source
- Amplify → AWS ecosystem integration

- TypeScript = Safer, typed JavaScript
- Ideal for modern React apps