Week 12: Sessions and Authentication

INFSCI 2560: Web Technologies and Standards

Agenda

- HTTP Sessions
- Cookies
- Sessions with Passport.js
- Authentication with OAuth

Housekeeping

Project 3b - due Wednesday

Final Project Descriptions - due Monday, November 18

- Each team member submits
- Describe what your role will be in the project
- Describe key functionality and tech stack you plan to use.

HTTP is a stateless protocol

- The HTTP Protocol is stateless protocol, which means every request/response client/server communication is independant
- This is by design because it enables HTTP to scale easily
- However there are many use cases where keeping track of state across requests

Examples of HTTP Sessions

eCommerce - shopping carts, storing address & payment

Complex Web Applications - Gmail, Github

Social media - Facebook, Twitter

Media and News - NYTimes limit of 10 articles per month

HTTP Sessions

- HTTP Sessions provide a mechanism for stitching together a series of independent requests
- A **web session** is a sequence of network HTTP request and response transactions associated to the same user
- Sessions can store variables – such as access rights and localization settings – for every interaction a user has with the web application for the duration of the session
- Can apply to anonymous or authenticated (logged-in) users
- When authenticated, sessions allow the ability to track requests to:
  - Apply access controls to create a secure web application
  - Regulate access to user data
  - Increase the usability of the application through user preferences

Cookies

A **cookie** is a small piece of data sent by a server to a browser and stored on the user's computer while the user is browsing.

Cookies can be attached to every HTTP request using the cookie HTTP Header

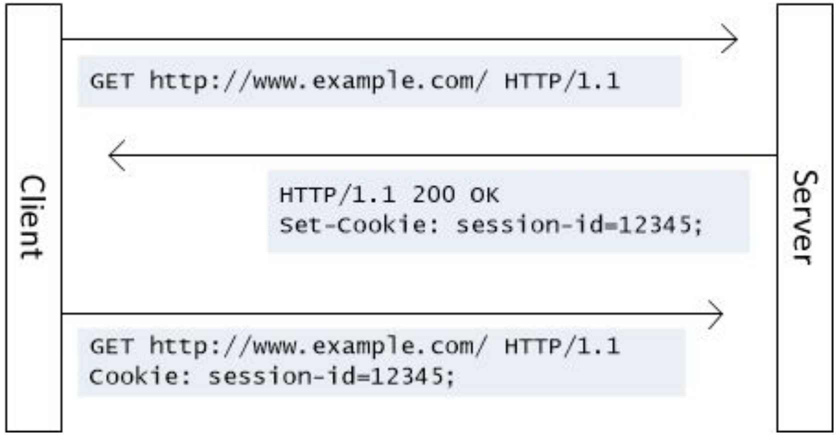
This allows us to track a little bit of state across what would otherwise be completely independent HTTP requests and responses

Cookies are primarily used for: (1) Session management, (2) personalization and (3) tracking.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies

Cookie-Based Session Management

- This cookie is sent back to the server when the user tries to access certain pages
- The cookie allows the server to identify the user and retrieve the user's session from the session database, so that the user's session is maintained.
- A cookie-based session ends when the user logs off or closes the browser.
- We will be using Passport.js, which integrates nicely with Node.js and Express
- Session IDs should be random, very hard to guess identifiers



```
GET http://www.example.com/ HTTP/1.1
```

```
HTTP/1.1 200 OK
Set-Cookie: session-id=12345;
```

```
GET http://www.example.com/ HTTP/1.1
Cookie: session-id=12345;
```

Image from HTTP Cookies in ASP.NET Web API

Types of Cookies

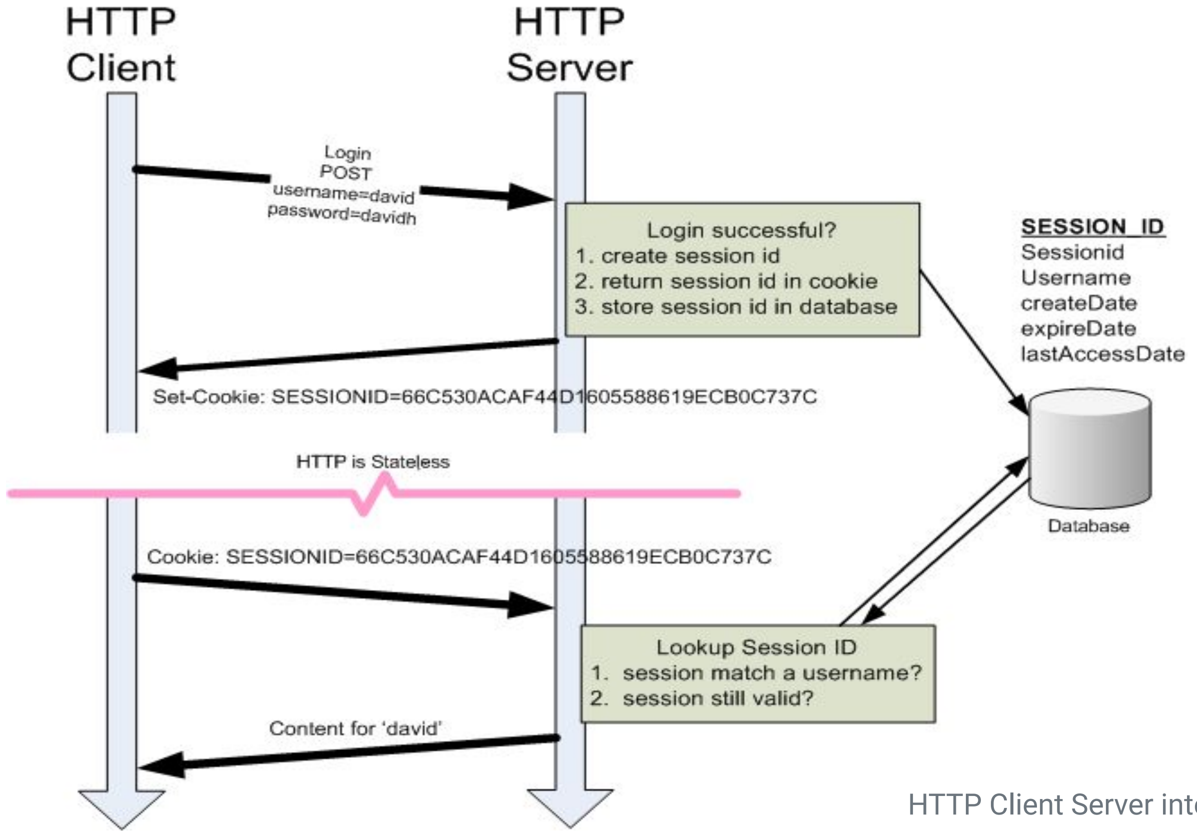
- **Session Cookie** - This type of cookies dies when the browser is closed because they are stored in the browser's memory. Can be used for shopping carts or anonymous user preferences that don't need to be saved across a multiple browser sessions
- **Persistent or Permanent Cookie** - Stored in a file or database in the browser. Expire at a specific date (`Expires`) or after a specific length of time (`Max-Age`).
- **Third Party Cookie** - A cookie set by a different domain from the servier. These cookies are used for tracking patterns and advertising
- **Secure Cookie** - Cookies that are only transmitted over an encrypted connection. Browser won't send the cookie over an insecure connection
- **Zombie or Evercookies or Supercookies** - Cookies that get recreated after they are deleted and persist on the client all the time. Popular with advertising and analytics trackers. VERY HARD TO DELETE

Cookie Attributes

- **Name** - Specifies the name of a cookie for retrieving the cookie
- **Value** - Specifies the value of cookie. Max size of all cookies is 4093 bytes per domain
- **Secure** - Specifies if the cookie should only be transmitted over encrypted HTTPS connections. Default is false
- **Domain** - Specifies the domain name associated with the cookie. Helps the browser determine when to send a cookie with HTTP requests (don't want to send all cookies to all servers)
- **Path** - Specifies a server path ("/", "/users/", "/login") for sending the cookie.
- **HTTPOnly** - Means the cookie will only be available on the HTTP protocol, not accessible to JavaScript
- **Expires** - Specify when the cookie expires and should no longer be sent with HTTP Requests. If set to 0 the cookie will expire when the browser closes

```
 Set-Cookie: id=a3fWa; Expires=Wed, 13 Nov 2019 07:28:00 GMT; Secure;
HttpOnly
```

How Cookies Work


HTTP Client
HTTP Server
Login POST username=david password=davidh
Login successful?
1. create session id
2. return session id in cookie
3. store session id in database
SESSION_ID
Sessionid
Username
createDate
expireDate
lastAccessDate
Set-Cookie: SESSIONID=66C530ACAF44D1605588619ECB0C737C
HTTP is Stateless
Database
Cookie: SESSIONID=66C530ACAF44D1605588619ECB0C737C
Lookup Session ID
1. session match a username?
2. session still valid?
Content for 'david'

HTTP Client Server interaction with cookies. Image from Hacking Articles

How Cookies Work

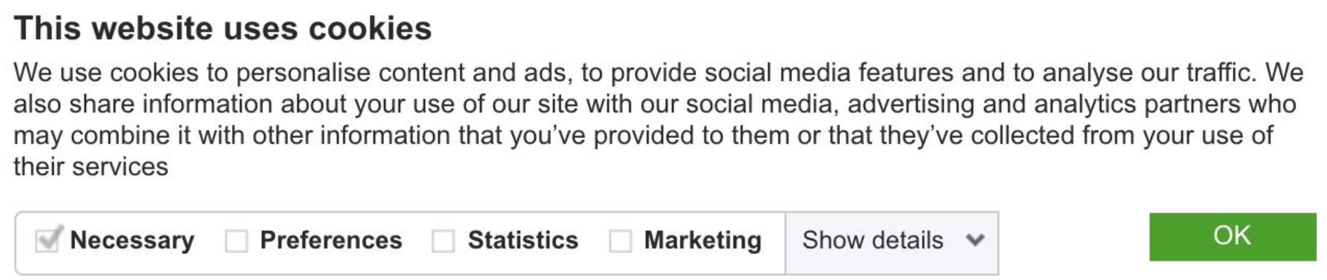
The Set-Cookie HTTP response header sends cookies from the server to the browser/device.. A simple cookie is set like this:

```
Set-Cookie: yucky-cookie=strawberry
Set-Cookie: yummy-cookie=oatmeal
```

Now, with every new request to the server, the browser will send back all previously stored cookies to the server using the Cookie header.

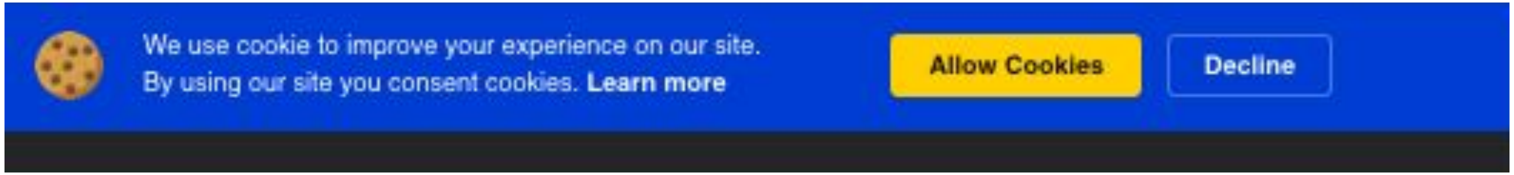
```
GET /sample_page.html HTTP/2.0 Host: www.example.org

Cookie: yummy_cookie=oatmeal; yucky-cookie=strawberry
```

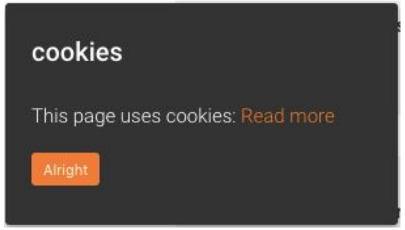


This website uses cookies

We use cookies to personalise content and ads, to provide social media features and to analyse our traffic. We also share information about your use of our site with our social media, advertising and analytics partners who may combine it with other information that you've provided to them or that they've collected from your use of their services



We use cookie to improve your experience on our site. By using our site you consent cookies. Learn more



cookies

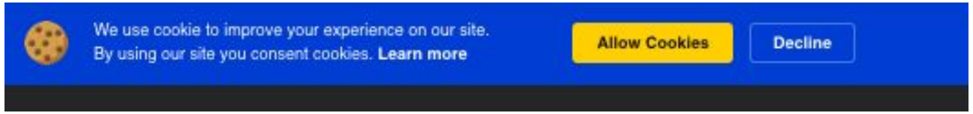
This page uses cookies: Read more

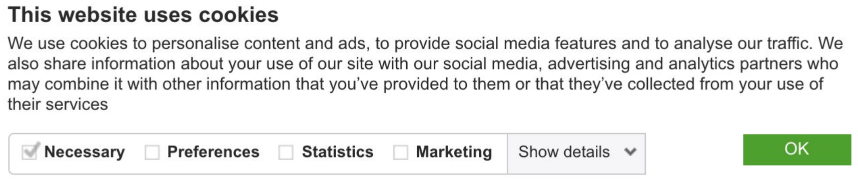
Cookie Laws

The EU Directive 2009/136/EC of the European Parliament means that before somebody can store or retrieve any information from a computer, mobile phone or other device, the user must give informed consent to do so.

Enforced in the US via the General Data Protection Regulation (May 25, 2018) which establishes that a business must have a “legal basis” for collecting personal data from individuals located in the EU.

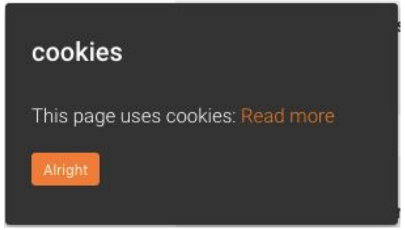
The California Consumer Privacy Act (CCPA) will become effective on January 1, 2020.


We use cookie to improve your experience on our site.
By using our site you consent cookies. **Learn more**


This website uses cookies

We use cookies to personalise content and ads, to provide social media features and to analyse our traffic. We also share information about your use of our site with our social media, advertising and analytics partners who may combine it with other information that you’ve provided to them or that they’ve collected from your use of their services

[✓] Necessary ☐ Preferences ☐ Statistics ☐ Marketing


cookies

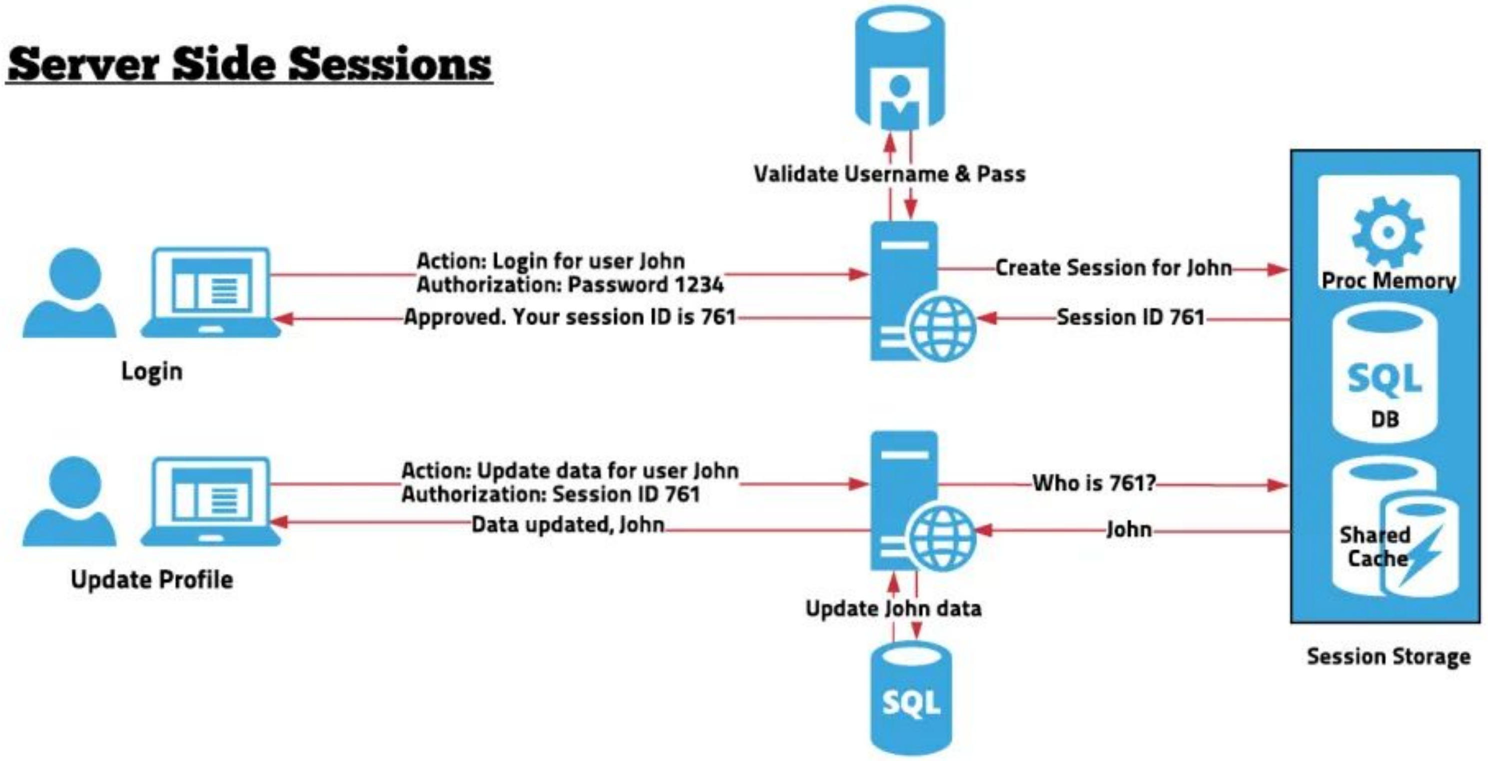
This page uses cookies: Read more

Sessions with Passport.JS

Session State

- By themselves, cookies don't tell us who is making HTTP requests
- They only indicate that the requests are coming from the same browser
- What we need is a connection between the cookie and session ID and user information like name, email, and other information
- The information about the user is maintained in the session state
- Then we use cookies and sessions to track across requests, but this means state needs to be maintained somewhere...

Server Side Sessions



Problems with Server Side State

- Sessions based on Session ID cookies are not stable and hard to scale
- Cookies are not supported by some mobile and desktop applications
- Cookies only work for a single domain
- To address these issues there are new approaches to managing session state by saving state on the client like JSON Web Tokens
- Read these articles about session state - The Rise and Fall of Server-Side Sessions Part 1 and Part 2

Authentication

- Passport.js is an authentication middleware for Node. It is designed to authenticate requests.
- Passport is an Express middleware library that takes away a lot of the headaches of authenticating users and maintaining their state across requests.
- Passport supports a lot of different authentication *strategies* from username and password to using Facebook, Twitter, or others
    1. Click here to see all the possible strategies supported by Passport.js
- To set up Passport you must do three things
    1. Set up the Passport middleware
    2. Tell Passport how to *serialize* and *deserialize* users - This is code that translates session IDs into user objects with their state
    3. Tell Passport how to authenticate users - How to verify the user, in our case we will check passwords stored in a database

Setting up the Middleware

Passport.js leverages several libraries to simplify the management of authentication and session state

Basically we just need to load and wire up a series of libraries that will do all the hard work

And write a bit of implementation code and glue code

```
const bodyParser =
   require('body-parser');
const cookieParser =
   require('cookie-parser');
const session =
   require('express-session');
const flash = require('connect-flash');
const passport = require('passport');
const MongoStore =
   require('connect-mongo')(session);
```

Setting up the Middleware

Then we need to tell our Express app how to generate session IDs and where to store session state

We don't want our session ids to be easily guess, so we use a SECRET, a randomly generated string, to make it hard to guess our session IDs

We also tell express-session that we are going to use connect-mongo to store session state in the mongo database instead of keeping state in memory

Finally we initialize Passport

```
app.use(session({
        secret: process.env.SECRET,
        store: new MongoStore({
            mongooseConnection :
            mongoose.connection,
            autoRemove : 'native' })
      }));
      app.use(flash());
      app.use(passport.initialize());
      app.use(passport.session());
```

Serializing and Deserializing Users

This code translates the sessionId to a User object and back.

This means we need to write some code, which is typically kept in a separate file

- In this case in a file called setuppassport.js

```
const passport = require('passport');
      const User = require("./models/user");
      const LocalStrategy = require('passport-local').Strategy;

      // serializing and deserializing users from the mongodb
      module.exports = function() {
        passport.serializeUser(function(user, done) {
          done(null, user._id);
        });

        passport.deserializeUser(function(id, done) {
          User.findById(id, function(err, user) {
            done(err, user);
          });
        });
      };
```

Let’s build a complete app that uses authentication.

- In this example we are going to implement our own **authentication mechanism**
- We will implement a **signup page** that presents a web form to the user to create an account
- We will implement a **login page** for users to log into existing accounts
- We will **modify the view** depending on if the user is logged in or not
- And finally we will create a **profile editing page** that can only be accessed if the user is logged in

Break (10 minutes)

Creating the User Model

This should look familiar.

This defines a user model with a username, password, creation data and some additional information

```
// Data model for users
const userSchema = new Schema(
   {
     username: {type: String, required: true,
                          unique: true },
     password: {type: String, required: true},
     created: { type: Date, default: Date.now },
     displayName: String,
     bio: String
    }
 );
```

Encrypting the Password in the Database

- The data model above stores passwords as Strings, which is fine but we need to add some additional layers of protection
- Storing passwords "in the clear", that is, unencrypted is a big no no
- If hackers got into our database, which they could easily, it would mean they could steal our user's passwords
- So we use a library called **bcrypt** to encrypt the passwords when a user creates an account

```
// Encrypting the password
   var noop = function() {};
   userSchema.pre("save", function(done) {
     var user = this;
     if (!user.isModified("password")) {
       return done();
     }
     bcrypt.genSalt(SALT_FACTOR, function(err, salt)
{
     if (err) { return done(err); }
          bcrypt.hash(user.password, salt, noop,
          function(err, hashedPassword) {
            if (err) { return done(err); }
            user.password = hashedPassword;
            done();
          });
        });
      });
```

Making a function to check encrypted passwords

- This function checks if two password strings are the same (to validate user authentication)
- It encrypts the password submitted by the user when the try to log in and them compares the two encrypted values.
- If they match, success!

```
// checking password function
userSchema.methods.checkPassword =
    function(guess, done) {
        bcrypt.compare(guess,
        this.password, function(err, isMatch)
          {
              done(err, isMatch);
          });
        };
```

Signup pages

```
// Route to signup page
      router.get("/signup", function(req, res){
        res.render("signup");
      });

      router.post("/signup", function(req, res, next){
      var username = req.body.username;
      var password = req.body.password;

      User.findOne({username: username }, function(err, user){
        if (err) {return next(err);}
        if (user) {
          req.flash("error", "User already exists");
          return res.redirect("/signup");
        }
        var newUser = new User({
          username: username,
          password: password
        });
```

```
console.log(username);
        newUser.save(next);
      });
      }, passport.authenticate("login",
{
        successRedirect: "/",
        failureRedirect: "signup",
        failureFlash: true
      }));
```

Logging In

```
// setup the authentication strategy
passport.use("login", new LocalStrategy(function(username, password, done) {
  User.findOne({username: username}, function(err, user){
    if (err) { return done(err)};
    if (!user) {
      return done(null, false, {message: "No user with that username"});
    }
    user.checkPassword(password, function(err, isMatch){
      if (err) { return done(err);}
      if (isMatch) {
        return done(null, user);
      } else {
        return done(null, false, {message: "Invalid password"});
            }
          });
        });
      }));
```

Logging In

```
// route to login page
      router.get("/login", function(req, res){
        res.render("login");
      });

      router.post("/login",
        passport.authenticate("login", {
             successRedirect: "/",
               failureRedirect: "/login",
             failureFlash: true
      }));
```

Create login view

```
<h1>Log in</h1>

<form class="px-5" action="/login" method="post">
  <input name="username" type="text" class="form-control"
placeholder="Username" required autofocus>
  <input name="password" type="password" class="form-control"
placeholder="Password" required>
  <input type="submit" value="Log in" class="btn btn-primary btn-block">
</form>
```

Passing Data to Views

```
// Make user information available to templates
        router.use(function(req, res, next) {
          res.locals.currentUser = req.user;
          res.locals.errors = req.flash("error");
          res.locals.infos = req.flash("info");
          next();
        });
```

Modify view if user is logged in

```
<body>

   <% if (currentUser) { %>
         <a class="nav-link" href="/edit">Hello <%= currentUser.name() %>!</a>
             <a class="nav-link" href="/logout">Logout</a>
       <% } else { %>
             <a class="nav-link" href="/signup">Signup</a>
             <a class="nav-link" href="/login">Login</a>
      <% } %>
    </nav>
```

Using Flash

```
const flash = require('connect-flash');
.
.
.

// Make user information available to
templates

router.use(function(req, res, next) {
  res.locals.currentUser = req.user;
  res.locals.errors =
      req.flash("error");
  res.locals.infos = req.flash("info");
  next();
});
```

```
//display flash values in view
<% errors.forEach(function(error) { %>
        <div class="alert alert-danger" >
          <%= error %>
        </div>
      <% }) %>
      <% infos.forEach(function(info) { %>
        <div class="alert alert-info"
role="alert">
          <%= info %>
        </div>
      <% }) %>
```

Access Control

```
// authentication middleware
       function checkAuthentication(req, res, next){
         if (req.isAuthenticated()) {
           next();
         } else {
           req.flash("info", "You must be logged into see this page" );
           res.redirect("/login");
         }
       };
```

Access Control

```
router.get("/edit", checkAuthentication, function(req, res){
        res.render("edit");
      });

router.post("/edit", checkAuthentication, function(req, res, next){
  req.user.displayName = req.body.displayname;
  req.user.bio = req.body.bio;
  req.user.save(function(err) {
    if (err) {
      next(err);
      return;
    }
    req.flash("info", "Profile Updated!");
    res.redirect("/edit");
  });
});
```

https://2560-passport-demo.glitch.me/

The complete working solution. Review this code. Know how it works and all comes together.

Authentication is hard.

system is difficult:

- How are you going to save passwords securely?
- How do you help with forgotten passwords?
- How do you make sure users set a good password?

OAuth or Open Authentication is an open-standard authorization protocol or framework used for authorization.

This is also known as a secure, third-party, delegated authorization.

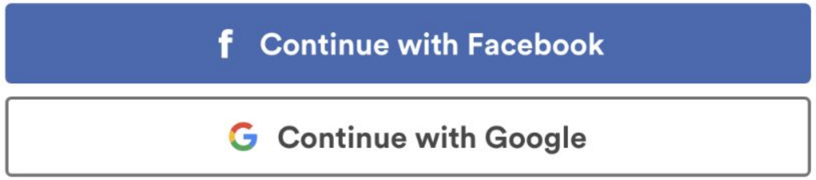
OAuth

For users:

- It allows a user to log into a website like AirBnB via some other service, like Gmail or Facebook.
- Don't have to memorize multiple passwords/

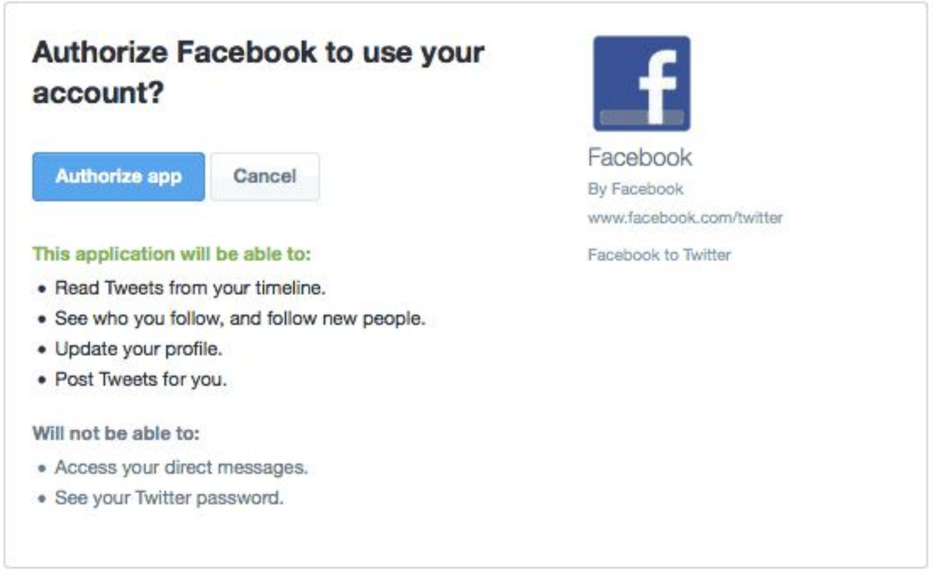
For developers:

- It lets you authenticate a user without having to implement log in



OAuth Examples

- Instagram can access your profile and post updates to your timeline
- Facebook posts are now shared with Twitter
- Use Google login to access other 3rd party sites.



Authorize Facebook to use your account?

This application will be able to:

- Read Tweets from your timeline.
- See who you follow, and follow new people.
- Update your profile.
- Post Tweets for you.

Will not be able to:

- Access your direct messages.
- See your Twitter password.

OAuth2 APIs

Companies like Google, Facebook, Twitter, and GitHub have OAuth2 APIs:

- Google Sign-in API
- Facebook Login API
- Twitter Login API
- GitHub Apps/Integrations

- OAuth2 is standardized, but the libraries that these companies provide are all different.
- You must read the documentation to understand how to connect via their API.

Basics steps of OAuth2

1. Get an API key
2. Whitelist the domains that can call your API key
3. Use Google/Facebook/Github’s API to create a login button and pass your client id
4. You can get information like:
    i. Name, email, etc
    ii. Some sort of **Identity Token**

You should update your database to store these users and their profile information.

Reminder about storing API keys

How are you supposed to store API keys?

Best practice: **Use Environment Variables**

- Set the environment variable on your host (e.g. Glitch)
- Can access the environment variable's value using process.env.VAR_NAME

Activity 9

https://glitch.com/~2560-oauth-github