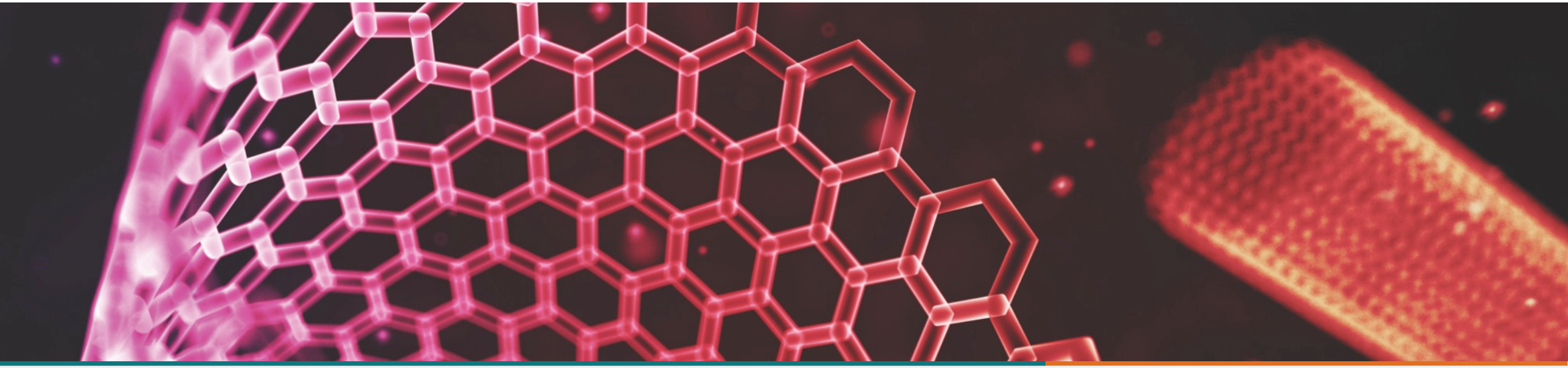
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

CS 546 – Web Programming I
Middleware and Authentication



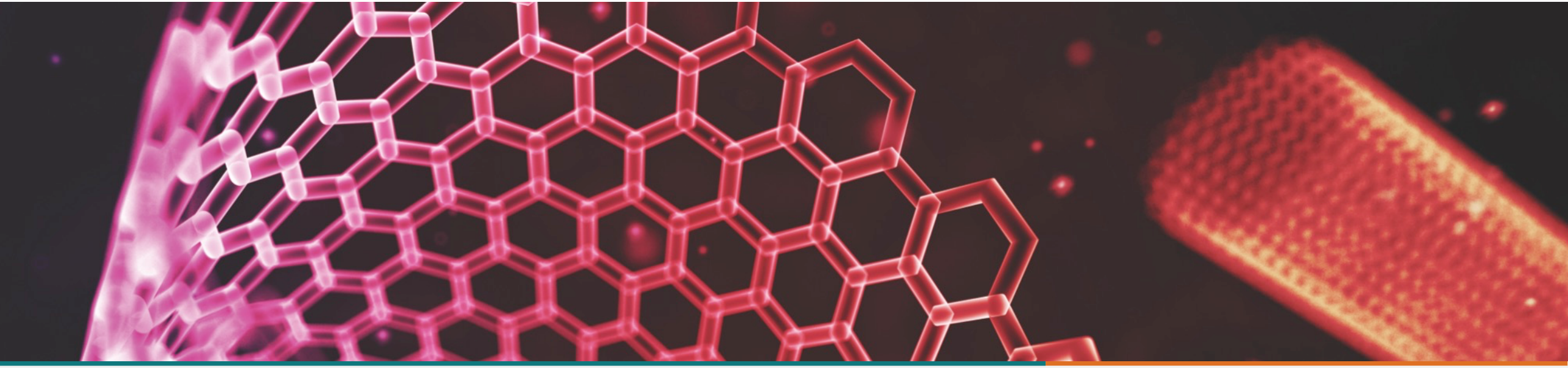
STEVENS
INSTITUTE of TECHNOLOGY

Schaefer School of
Engineering & Science

stevens.edu

Patrick Hill
Adjunct Professor
Computer Science Department
Patrick.Hill@stevens.edu

Middleware



What is Middleware?

A middleware is a function that has access to the request and response objects. These functions can:

- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware function in the stack.

You can apply middleware to the entire application, or portions of the application

- You can apply it to a portion of the application by supplying a path as the first parameter to the middleware function

It is called a middleware because it sits in the middle of the response and the request.

Practical Uses for Middleware

Middlewares are useful for a number of reasons, and have many common uses such as:

- Logging requests
- Authentication
- Access control
- Caching data
- Serialization

Writing a Middleware

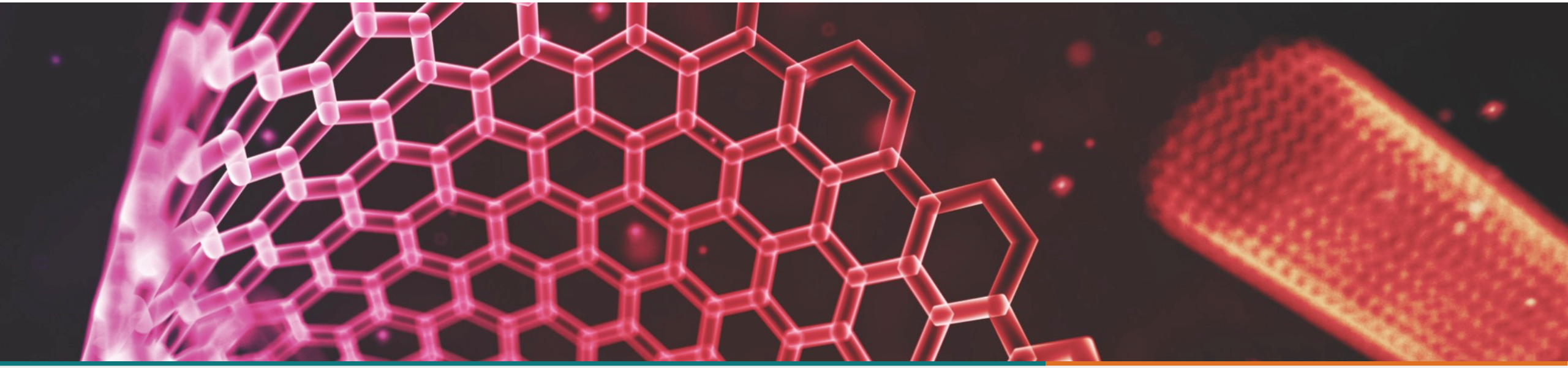
Writing a middleware is extremely easy

- Register your middleware, optionally providing a path to apply that middleware to
- Have your middleware perform a task and when done:
  - Have your middleware end the response
  - Have your middleware call the next middleware

As an example, see the ***app.js*** file, which has several middleware functions:

- One which will count the number of requests made to your website
- One which will count the number of requests that have been made to the current path
- One which will log the last time the user has made a request and store it in a cookie.
- One which will deny all users access to the ***/admin*** path.

Authentication and Authorization



What is Authentication?

Authentication is the process of verifying what user is currently operating in a system. You would be most familiar with it through the use of usernames and passwords.

For example, on MyStevens, you are authenticated by supplying your Stevens username and password. By providing this data, the system sends a Cookie to your browser that stores a session ID that associates your requests to your user account.

What is Authorization?

Authorization and authentication are often confused. While **authentication** handles who you are, **authorization** is the process of validating what you can access.

For example, as a faculty member, my user account is authorized to input grades to student accounts, whereas student accounts are not.

- Even more granular, I am authorized in the system for CS-546 grades, but not MGT-671

Authorization comes in many forms; typically, you will see three or more layers of authorization

- Public facing pages; even users that are not authenticated can see these pages. For example, your homepage or login page would be public facing
- Authenticated only pages; pages that all authenticated users can see
- Role or claim based pages; pages that users can only see if they have certain account types, such as faculty being able to access grading features whereas students cannot

What are the Ways We Can Authenticate?

There are many strategies to authenticate an HTTP Request. We will fully explore cookie-based authentication in a later section.

Some other common authentication strategies:

- Token based authentication; passing an API token in the querystring to validate that you are a particular user
- Basic Access Authentication; providing a username and password on every HTTP request
- JWT Tokens being used for authentication

How Can We Authorize Requests?

Each route can have many middlewares applied to it, allowing us to add a sequence of middlewares that block access to that route unless a user should have access to that route.

For example, let us say that there are two routes on the server, /SubmitGrades and

/SeeClassesInSemester. Let us say Faculty can access /SubmitGrades, and Students or Faculty can see /SeeClassesInSemester

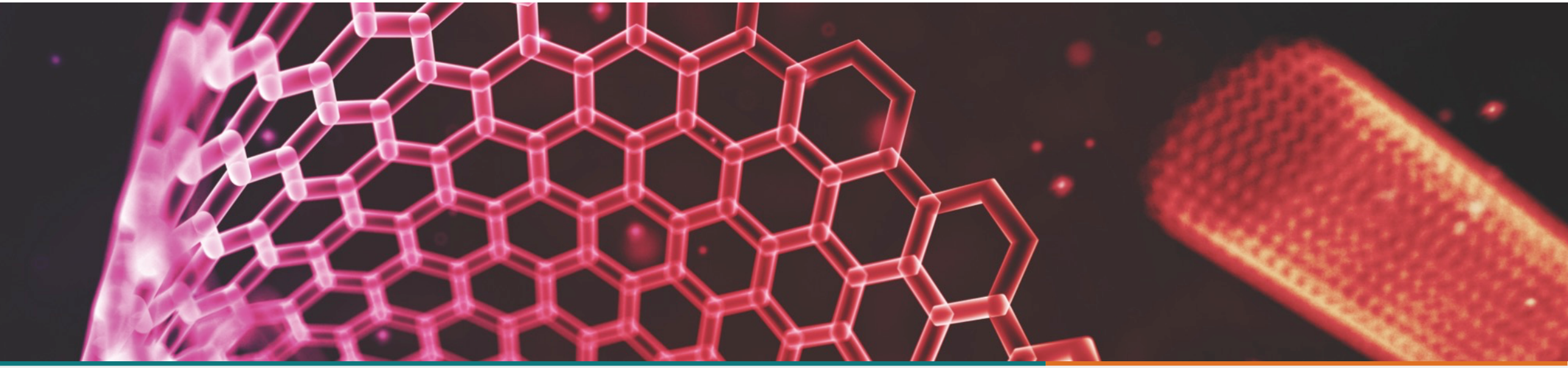
To the /SubmitGrades route, you would apply the following middlewares:

- One to authenticate the user; if no user is authenticated, respond with a status code of **403**
- Next, to check if the user is a faculty member; if not, respond with a status code of **403**

To the /SeeClassesInSemester route, you would apply the following middlewares:

- One to authenticate the user; if no user is authenticated, respond with a status code of **403**
- Next, check if user is faculty or a student; if not, respond with a status code of **403**

Using Cookies



What is a Cookie?

No, not this kind!

What is a Cookie?

An HTTP Cookie is a small piece of data that is shared between the server and the client.

- Can be read or set in client or server (using express-session, it's all stored on the server)
- Ultimately, sent back and forth as string data

Cookies are sent through headers.

HTTP Cookies cannot be deleted, but can be expired

- After their expiration date, they will automatically be removed

Cookies will be sent back to the server on every request automatically; only new or updated cookies will be sent in a response.

Cookies are a browser concept, and they are rarely passed back and forth when you are writing APIs or requesting resources programmatically.

Using Cookies with Node

There are two different ways we deal with cookies and Node.js

- **cookie-parser**
- **express-session**

cookie-parser

One way to deal with cookies is using the cookie-parser middleware. It allows us to easily handle our cookies as an object.

Another way is to manually parse headers and parse objects as well as we can, which can get redundant.

npm install cookie-parser

We then apply the cookie-parser as a middleware, without a route path so that it applies to the whole application.

Using Cookies with the cookie-parser

CLIENT SIDE

You can set by setting ***document.cookies ="key=value"***

Even though you are re-assigning, it will simply add it to your list of cookies.

You can get a list of all your cookies and their values using the ***document.cookies*** and parsing it to find the cookie of your choice.

Deleting cookies requires that you set the cookie with an expiration; ie:
document.cookies="key=value; expires=Thu, 01 Jan 1970 00:00:00 UTC";

SERVER SIDE

You can set cookies by calling the ***response.cookies(name, value, options)*** function.

You can get cookies by referencing the ***request.cookies*** object, which will have cookies keyed by name.

Deleting cookies requires you to expire the cookies, which we can do by setting the cookie with the *expires* option set to any time in the past, then calling ***response.clearCookie(name)***

We will see an example in our lecture code

express-session

Another way to handle cookies is using the express-session package.

npm install express-session

When we use express-session, the cookies are stored a bit differently. A cookie on the client is created that ONLY stores the session ID (which is generated by express-session). Any fields of data that we want to store in the cookie, are actually stored on the server in the **req.session** object. One downfall about this, is if the server reboots, all sessions are destroyed. That means the cookie stored on the client will be invalid. If we manually generate and store session ID's then we can avoid this. Another way to avoid this is use a package to cache the sessions on the server, so if the server does go down, the sessions can be restored from cache.

Using express-session

We first use it in our ***app.js*** like so:

```
app.use(
  session({
    name: "AuthCookie",
    secret: "some secret string!",
    resave: false,
    saveUninitialized: true,
    cookie: {maxAge: 30000}
  })
);
```

Using express-session

We can store data in the session in the **req.session** object. We can access and modify this in our middleware functions and in our routes!

For example, say we want to store an object representing the currently logged-in user. We can modify req.session to store this information.

```
req.session.user = { firstName: userDataFromDb.firstName, lastName: userDataFromDb.lastName,
userId: userDataFromDb._id };
```

Using express-session

We can store data in the session in the **req.session** object. We can access and modify this in our middleware functions and in our routes!

For example, say we want to store an object representing the currently logged-in user. We can modify req.session to store this information.

```
req.session.user = { firstName: userDataFromDb.firstName, lastName: userDataFromDb.lastName,
userId: userDataFromDb._id };
```

express-session stores the session ID in the `req.session.id` field. This field cannot be modified, it is read-only.

One thing about express-session is all the data is stored on the server. A single cookie is created on the client that ONLY stores the session ID, express-session then gets/sets the data associated with that session ID through middleware functions and in routes.

When using the traditional cookie approach, the actual data is stored in the cookie on the client. We will see this in our lecture code.

Using express-session

```
/*
Middleware function applied to /private route that checks to
see if there is a user stored in req.session.user this is set when
the user logs in successfully,if they are not logged in and try
to access the private route, we will redirect them to
home (usually, redirect to login page)
*/
app.use('/private', (req, res, next) => {
  console.log(req.session.id)
  if (!req.session.user) {
    return res.redirect('/');
  } else {
  next();
}
});
```

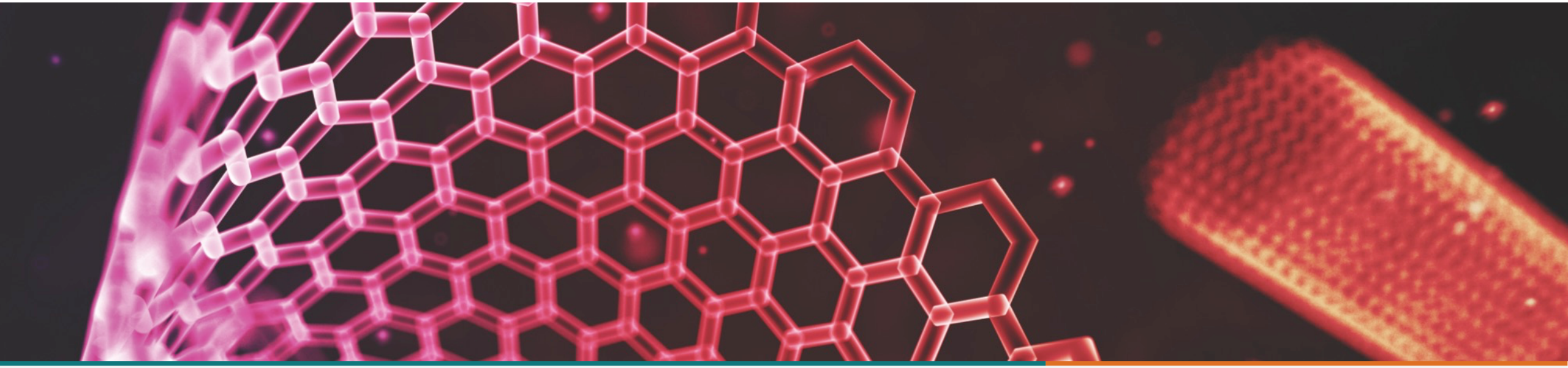
Using express-session

```
router.post('/login', async (req, res) => {
  /*  get req.body username and password
      const { username, password } = req.body;
      here, you would get the user from the db based on the username, then you would read the
      hashed pw and then compare it to the pw in the req.body

     let match = bcrypt.compare(password, 'HASHED_PW_FROM DB');

    if they match then set req.session.user and then redirect them to the login page
    I will just do that here */
  req.session.user = { firstName: userFromDb.firstName, lastName: userFromDb.lastName,
userId: userFromDb._id };
  res.redirect('/private');
});
```

Cookie Based Authentication using Express, Middleware, and MongoDB



Authentication

Authentication is the act of confirming the identity of a person, group, or entity. In web technology, this often means creating a **user login system**

- You will use a combination of data in order to identify a user.

There are many other forms of authentication in web technology:

- You can make an authentication system that allows you to limit API Access
  - Force users to have a token
  - Allow users a certain number of access hits a month
- You can selectively allow or dis-allow access to resources based on user login state

Implementing Authentication

In order to implement authentication and create a user login system, we will be breaking down the task into several steps:

- Creating and storing users
- Allowing users to login via a form
- Storing session data in a cookie
- Validating the data stored in the cookie
- Storing the user as part of the request object.

Let’s walk through this process.

Creating and Storing Users

The first step of authentication is very, very easy; you have to create, and store users.

There are some things you'll be storing, and some things you'll be storing in a very specific way

- First off, you will never store a plaintext password. You will be using the bcrypt package in order to create a hash of the password
- For the sake of authentication, you're going to be adding an array for users that will keep track of multiple session identifiers. These session identifiers will allow you to keep track of logged in browser sessions

You will need to create a form to allow users to signup, where you will need to check for:

- Duplicate username/emails/other non-duplicatable data
- Mixed case of username/email addresses
  - patrickhill@stevens.edu should be treated the same as PaTriCkHill@stevens.edu and PATRICKHILL@steVens.edu same for usernames: phill, PHILL, Phill should be treated as equal.
- Existence of passwords

Allowing Users to Login via a Form

This step is extremely easy!

You will need to provide users with some way to actually perform a login. You will need to setup a form that allows users to **POST** their username and password to a route.

This route will need to validate the username and password provided against entries in the database.

- You will retrieve the user with that matching username
- You will use ***bcrypt*** to compare if their supplied password is a match to their hashed password that is stored in the database
  - I have created a simple file, ***bcrypt_example.js*** to demonstrate how to use ***bcrypt*** to create and compare hashes.

If there is a match, you can proceed; if not, you will simply not allow the request to continue and display an error to the user.

Storing Session Data in a Cookie

If the user logged in with proper credentials, you will then create a session id! This session ID should be some sort of very long identifier, such as a GUID.

Rather than storing the user ID or username, and password in cookies, we instead are opting to store a session ID so that the username or password cannot be intercepted.

This session ID will be passed to the user via-cookie and will also be stored as one of many session ID's on the user in the database. So when the user logins in, the server can generate the GUID, store that in an array of session ID's in the user document and send that back as a cookie to the client in response.

Or we can just use express-session to handle our sessions for us

Validating the Data Stored in the Cookie

It is now time to make your middleware!

Your middleware should run on each request, and will check for a cookie containing a session ID

- If it contains a session ID, you will check the database for a single user that has that session ID stored in their session ID field
  - If there is a match, you've found your user!
  - If not, your request is coming from an unauthenticated source; expire their cookie.

- If not, your request is coming from an unauthenticated source.

Storing the User in the Request Object

In your middleware, you have access to the request and the response objects, and you can add properties to them easily.

If you are able to associate a session ID with a user, you may define a property on the request (or response!) object that stores the user, or some representation of them (ie: just storing the user ID).

The data you store will be accessible:

- In middleware that are defined after the authentication middleware
- In your routes

If you define middleware after your authentication middleware, you can attach them to particular paths (such as ***/user***) and, if a user is not logged in, you can redirect them. You can also do things such as check on other paths (ie: ***/admin***) to see if the user has permission to access those paths, and redirect if not.

Logging Out

Logging out is extremely easy, and only has two steps!

- After hitting a logout route, you will expire the cookie for the session ID
- You will remove the session ID from the user's session ID list
- You will invalidate any other cookies that are relevant to the user.

By doing both of those, you will have successfully invalidated the session and the user will no longer be authenticated.

The process is a little different when using express-session. Most of it is the same though, We do not have to manually generate session ID's or keep track of the sessions. We also could just store whatever data we need in the session.

Questions?