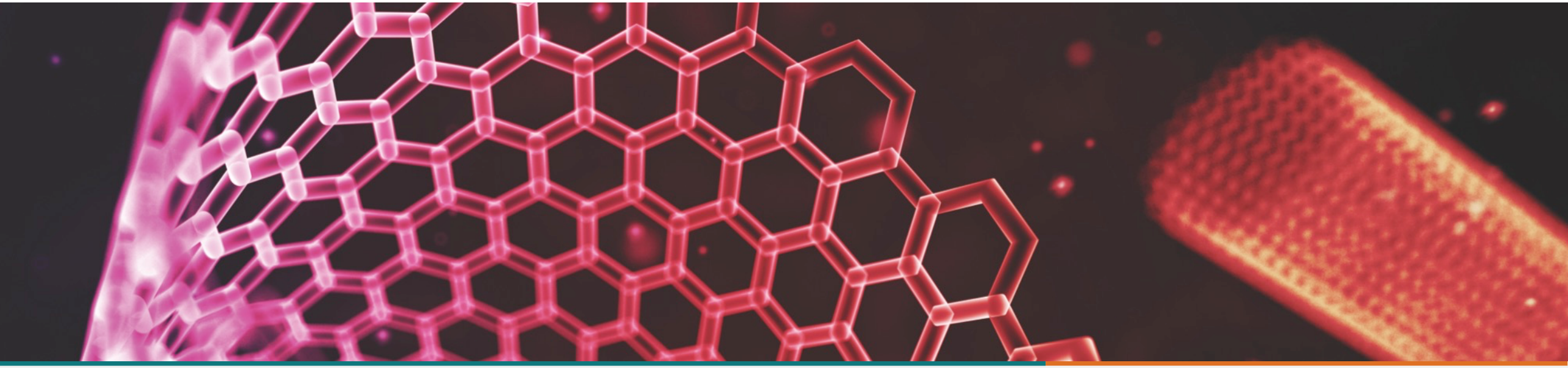
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

CS 546 – Web Programming I

Modules, Applications and Error Handling



STEVENS

INSTITUTE of TECHNOLOGY

Schaefer School of Engineering & Science

stevens.edu

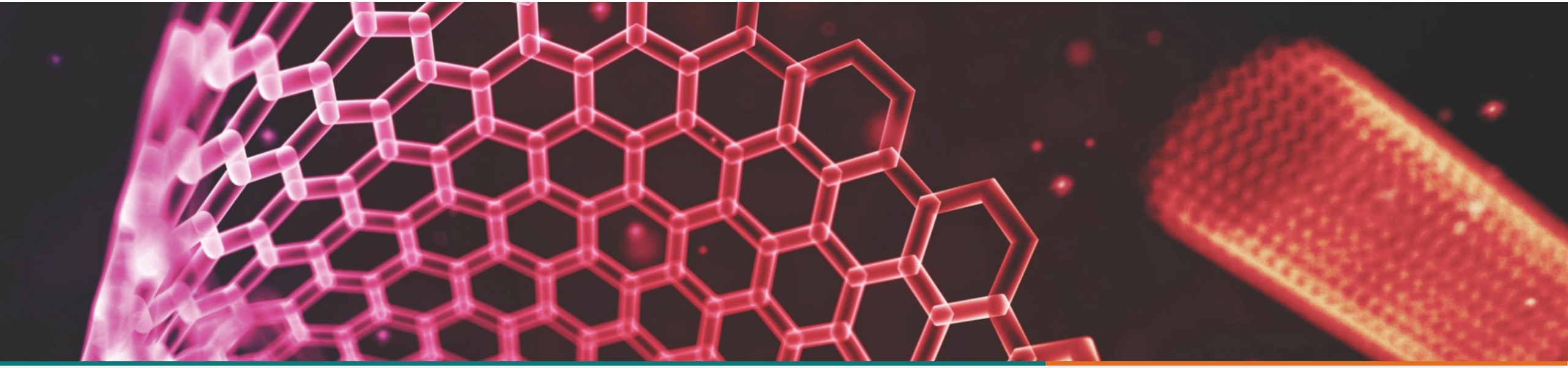
Patrick Hill

Adjunct Professor

Computer Science Department

Patrick.Hill@stevens.edu

Structure of a Node Application



What is a Node Application?

Throughout the term, you will be building a number of applications in Node. Ultimately, you can view a node app as a series of scripts that are run together. A web application, for example, would have the following scripts

- A script that, when run, listens on port 3000 for HTTP requests to retrieve data.
- A second script that runs as a background process and researches information to add to the databases.
- A third script that handles all the routes for the application.

What Makes a Node Application?

Generally, your application will have the following structure in this course;

- **ProjectFolder/**
  - package.json; describes the application, and its dependencies.
  - node_modules/; stores all the dependencies.
  - app.js; initializes and runs a server, or whatnot.
  - routes/; contains all of your routing scripts.
  - data/; contains all of your database access modules.
  - config/; stores all your configuration settings.
  - views/; contains your HTML views and templates.
  - static/; for static assets.
  - public/; for public assets (stylesheets, images, JS).
  - tasks/; contains any setup scripts like seed.js to seed your database

Package.json

The `package.json` is a very important file that stores information about your project, such as:

- Name
- Repository
- License Type
- List of dependencies
- List of developer dependencies
- Author
- Scripts

When starting a project, navigate to an empty folder and use the ***`npm init`*** command to interactively create the start of that file.

When submitting assignments in this course, you must submit the package.json file and include the author field with your name.

Package.json

Example of package.json :

```
{
  "name": "cs_546_first_package_json",
  "version": "1.0.0",
  "description": "This is our first app",
  "main": "app.js",
  "scripts": {
    "seed": "node ./tasks/seed.js",
    "test": "node test.js",
    "start": "node app.js"
  },
  "author": "Patrick Hill",
  "license": "MIT",
  "dependencies": {
    "axios": "^0.19.0",
    "bcryptjs": "^2.4.3",
    "express": "^4.17.1",
    "express-handlebars": "^3.1.0",
    "express-session": "^1.17.0",
    "handlebars": "^4.1.2",
    "jquery": "^3.4.1",
    "mongodb": "^3.4.0",
    "xss": "^1.0.6"
  }
}
```

Dependencies

It would be very difficult to reinvent every component of an application, every time you start coding.

Node allows authors to publish their code online on GitHub or on NPM (Node Package Manager); anyone can then download their code through the *npm* application and install it as a dependency towards a project.

References to these dependencies can be saved to your **package.json**

file. Dependencies are exposed in the form of modules.

You can install dependencies with the following command:

- `npm install PACKAGENAME`
- **Points will be deducted if you do not save all your dependencies so make sure you check your package.json to make sure all dependencies that you used are listed.**

Dependencies

Example of the dependencies object in package.json

```
"dependencies": {
  "axios": "^0.19.0",
  "bcryptjs": "^2.4.3",
  "express": "^4.17.1",
  "express-handlebars": "^3.1.0",
  "express-session": "^1.17.0",
  "handlebars": "^4.1.2",
  "jquery": "^3.4.1",
  "mongodb": "^3.4.0",
  "xss": "^1.0.6"
}
```

The Scripts Object

Your package.json command contains a field, “scripts”, that is an object containing different script tasks. For each key in the scripts object, you would have a value that contains the command for running each script (as if from the terminal).

- https://docs.npmjs.com/cli/run-script

For example, you could have a script for testing your code, and running your app, like so:

```
"scripts": {
"seed": "node ./tasks/seed.js",
"test": ”node test.js",
"start": "node app.js"
}
```

You can run the test command by running npm run test.

The start command can be run with `npm run start`, or the shorthand version, `npm start`.

You must edit your package.json file and add the start command to every lab going forward. Points will be deducted if you do not add the start command

What are Packages and NPM?

Node has a massive repository of published code that you can very easily pull into your assignments (where applicable) through the node package manager (npm).

You will require the modules that your packages export, and use code that other people have created, tested, and tried. You will then use these packages to expand on your own applications and build out fully functional applications.

Installing a Node Application

Node runs off a series of dependencies, which are managed through the package manager, NPM

When download a node application, you will be downloading it without dependencies, so you must install them using the following command.

- ***`npm install`***
- DependenciesWhen you run ***`npm install`***, npm will read the dependencies section of package.json and download all the dependencies
- External dependencies are stored in the node_modules folder

When you submit an assignment, you must submit it without the node_modules folder

- **Points will be deducted if you submit the node_modules folder**

Running a Node Application

Once your dependencies are installed, you can start your app with the ***`npm start`*** command.

The ***`npm start`*** command will run the start script from the scripts object in the package.json file.

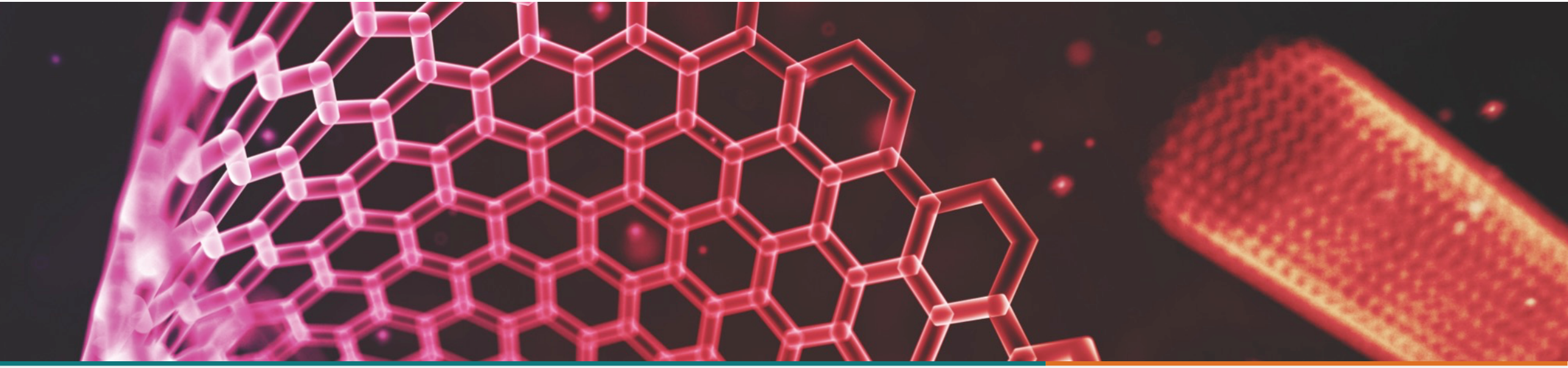
Again, You must edit your package.json file and add the start command to every lab going forward. Points will be deducted if you do not add the start command

Setting Up Your Application

When setting up an application, you will do the following steps:

- Make a new folder for the application
- Run `npm init` and go through the walkthrough
- Open package.json and update the `"scripts"` section and add a property of:
- `"start": "node app.js"`
- Where app.js is the name of the file you want to run on start
- Install your dependencies and save them to the package
- `npm install PACKAGENAME`
- Write some code in your starting file
- Run app with `npm start`

Modules



What is a Module?

Generally, a module is an individual unit that can be plugged into another system or codebase with relative ease. Modules do not have to be related, allowing you to write a system that allows many different things to interact with each other by writing code that glues them all together.

They are very flexible and allow you to organize your code very well!

In Node.js, you will be using modules everywhere. In our case, a module will be a specific object (think, an instance of a class) that has certain methods and data that you can access from other scripts. You will create your first module today.

Using Git to Get Today's Code

By this point, you should have also installed git. Git is a version control software, that you should be able to use via your command line. As part of this course, you will be learning how to version control your software.

Through your command line, issue the following command:

```
git clone https://github.com/stevens-cs546-cs554/CS-546
```

By cloning this repository (a codebase with a version history) you will make a local copy of the code in a folder called lecture_02/code CS-546

Navigate into the folder, so that you may run the following node scripts together.

Require

There is a special, global function called `require`, which will allow you to import code from other files, packages, etc.

When you require a file/package, you will be accessing whatever the programmer assigned to be exported in that file. From there, you can use the code.

This allows you to make very small, isolated code that performs related functions.

Example of a require statement: `const calculator = require("./calculator");`

This creates a reference to a module we wrote called calculator.js. We omit the .js from the statement and just use the filename. We would need to include the full path of the file. In this example, the file is in the same directory as the one requiring it.

If we were using an external module that we downloaded using `npm install`, then this is how we require it: `const prompt = require("prompt");`

How Do I Make My Code 'Requirable'?

- There is another global variable called module, which has a property called exports on it. When you require a file/package, it will take whatever is assigned to the `module.exports` variable in a package. You can export anything you want: a function, a variable, or an object that allows you to do any combination of these things.
- You can see an example of this in the calculator_module_example/calculator.js and calculator_module_example/app.js files.
- You can use those files in order to get started making your own modules!

There are a couple of ways we can use `module.exports` *to export our functions. One way is shown in* calculator_module_example/calculator.js *and the other way in* calculator_module_example/calculator_alt_export.js

Why Would I Use Modules?

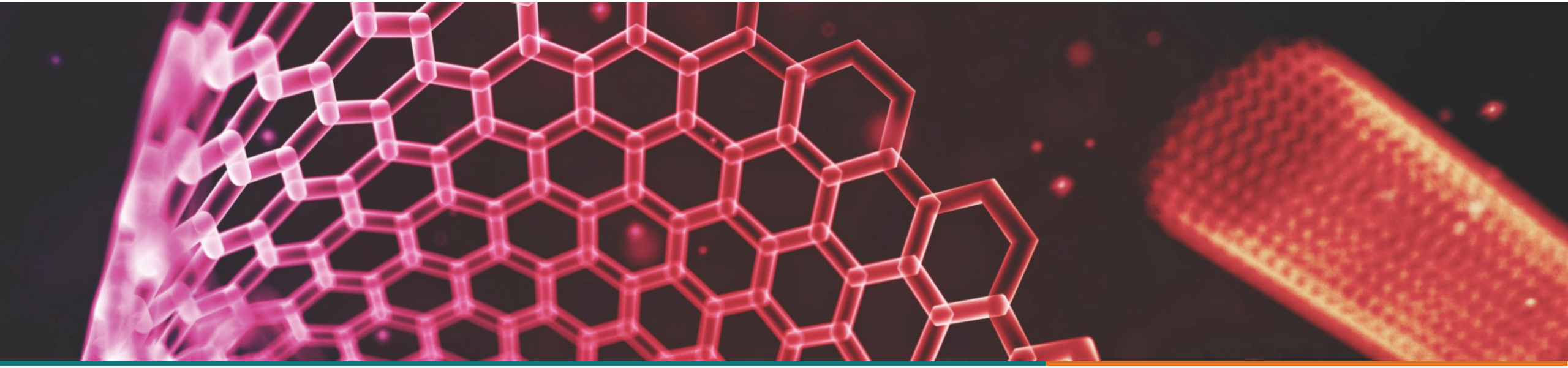
The more unrelated code you have together, the messier your application will become and the harder it will be to maintain.

- You can have accidental name collisions.
- It becomes harder to follow what the related components are in a large file.
- It becomes less readable overall.

Modules allow for many great things:

- You can strictly define what code is exported to be used, allowing you to make entire files with a defined structure.
- You can change the internal workings of a module to make it more performant and add more features while not updating external code.

Error Checking Module Methods



Error Checking

Modules are intended to be fundamentally nuclear and should be designed to act alone.

One of the most important aspects of this nuclear design is that each method exported in a module should have full error checking for it.

You should make sure each method checks that the arguments passed are valid in many ways:

- Check that arguments are provided (check if undefined)
- Check that arguments are of the expected type (use `typeof` operator) https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof
- Check that arguments are within proper bounds (i.e., if you are writing a division method, make sure that you cannot divide by 0;)

You must check for ALL edge cases that will cause your function to have unintended results.

Error Checking

Say you have the following code:

```
function divideTwoNumbers(num, den) {
return num / den;
}
```

JavaScript will let you call the function with the following input parameters without error!

```
console.log(divideTwoNumbers(10, 0)); //returns Infinity
console.log(divideTwoNumbers('Patrick', 'Aiden')); //returns NaN
console.log(divideTwoNumbers()); //returns NaN
console.log(divideTwoNumbers(10)); //returns NaN
```

Therefore, we need to check every input to make sure it will not produce any unintended results.

Throwing

When a method is given bad inputs, to prevent the method from running, you want to use the `throw` operator to stop execution of the current function with a user defined exception.

In JavaScript, you can throw any type. You can throw strings, numbers, booleans, objects, errors, or anything else.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw

By default, native JavaScript methods will throw an object that is an instance of the Error object.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error

Throwing

Looking at our previous example, Let's look at the updated function that checks the inputs and throws the errors using the *throw* statement when the input is not what we are expecting

```
function divideTwoNumbers(num, den) {
  if (typeof num != 'number') throw 'The Numerator Must Be A Number';
  if (typeof den != 'number') throw 'The Denominator Must Be A Number';
  if (den === 0) throw 'Error: Division by Zero';

  return num / dem;
}
```

Catching Errors

- You can catch errors by surrounding the methods that you call in `try`/`catch` blocks.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch

- **You do not want to catch the errors inside of your methods** (unless your method can recover from errors).
    - A recoverable error inside your method would be catching a failed file-save operation, catching that error, checking if it was because the filename was already in use, and changing over to save to a new filename.

You would not want to catch the errors that you throw from your method, otherwise the developer running your code would never be able to tell that an error occurred and they may want to deal with that error in their own way.

- You can catch particular error types by using the `instanceof` operator inside your catch statement.

Catching Errors

Again, Let's look at our previous example where we called our function. I have added one to the end that does have valid input. If I run this, then the first function call executes, it fails and then my application stops without ever executing the next function calls. So the first time it gets called the application just crashes and none of the following calls to that function execute.

```
console.log(divideTwoNumbers(10, 0)); //returns Infinity
console.log(divideTwoNumbers('Patrick', 'Aiden')); //returns NaN
console.log(divideTwoNumbers()); //returns NaN
console.log(divideTwoNumbers(10)); //returns NaN
```

Output:

```
if (den === 0) throw 'Error: Division by Zero';
                      ^

Error: Division by Zero
```

Catching Errors

If we surround it all but one try catch, the same thing happens but let's put another `console.log` after the calls to the function.

```
try {
  console.log(divideTwoNumbers(10, 0));
  console.log(divideTwoNumbers('Patrick', 'Joe'));
  console.log(divideTwoNumbers());
  console.log(divideTwoNumbers(10));
  console.log(divideTwoNumbers(10, 5));
} catch (e) {
    console.log(e);
  }
console.log('Hello world');
```

Catching Errors

We will get the following output:

```
Error: Division by Zero
Hello World
```

So, we can see that the other calls to ***divideTwoNumbers*** did not execute. If we wanted each one to execute, then we would need to surround each function call with its own ***try***/***catch***

Catching Errors

```
try {
  console.log(divideTwoNumbers(10, 0));
} catch (e) {
    console.log(e);
}

try {
  console.log(divideTwoNumbers('Patrick', 'Joe'));
} catch (e) {
    console.log(e);
}

try {
  console.log(divideTwoNumbers());
} catch (e) {
    console.log(e);
}
..... Other function calls...

console.log('Hello world');
```

Catching Errors

We will get the following output when we run it:

```
Error: Division by Zero
The Numerator Must Be A Number
The Numerator Must Be A Number
The Denominator Must Be A Number
2
Hello World
```

This is the proper way to handle errors in your modules. The only time we put multiple function calls into a single try catch is when the 2nd function call depends on the results from the first or something like that.

Making Custom Errors

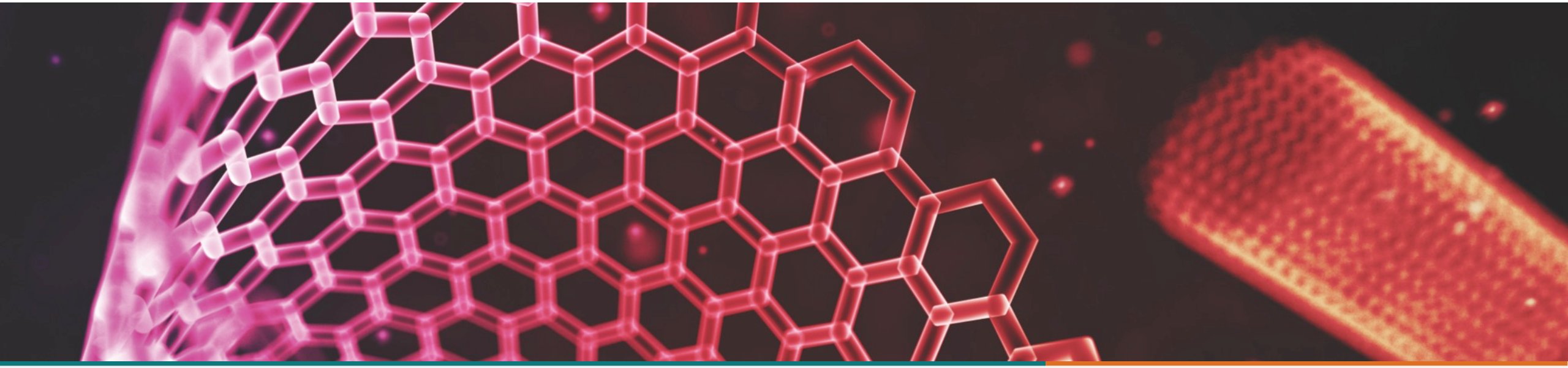
While you can throw any data type, you may find it useful for your method to throw different types of errors for different exceptions

- You can make an Argument Error for when your method is passed invalid arguments.

To make a custom error, you would extend the Error object with custom data so that you can check that particular type of error.

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error#Custom_Error_Types

Making a Calculator Module



The Goal of Your Module

The goal of your module is simple: take numbers and perform a basic numerical operation on them.

The module should not concern itself with things like getting user input, but it should concern itself with arguments that are valid.

You can see an example of a calculator module on our code for this week:

- https://github.com/stevens-cs546-cs554/CS-546/blob/master/lecture_02/code/calculator_module_example/calculator.js

Errors to Check for

***addTwoNumbers*(*num1*, *num2*):**

- Check that you are provided with 2 numbers.

***subtractTwoNumbers*(*num1*, *num2*):**

- Check that you are provided with 2 numbers.

***multiplyTwoNumbers*(*num1*, *num2*):**

- Check that you are provided with 2 numbers.

***divideTwoNumbers*(*num1*, *num2*):**

- Check that you are provided with 2 numbers.
- Check that denominator is not 0.

Exporting Methods

You export methods by attaching properties to the `exports/module.exports` global object.

When you require this file, you will be given a copy of this object.

- https://github.com/stevens-cs546-cs554/CS-546/blob/master/lecture_02/code/calculator_module_example/calculator.js
- https://github.com/stevens-cs546-cs554/CS-546/blob/master/lecture_02/code/calculator_module_example/app.js

You can see the alternative way to export methods using basic named functions in:

- https://github.com/stevens-cs546-cs554/CS-546/blob/master/lecture_02/code/calculator_module_example/calculator_alt_export.js

Writing a Quick Test Driver

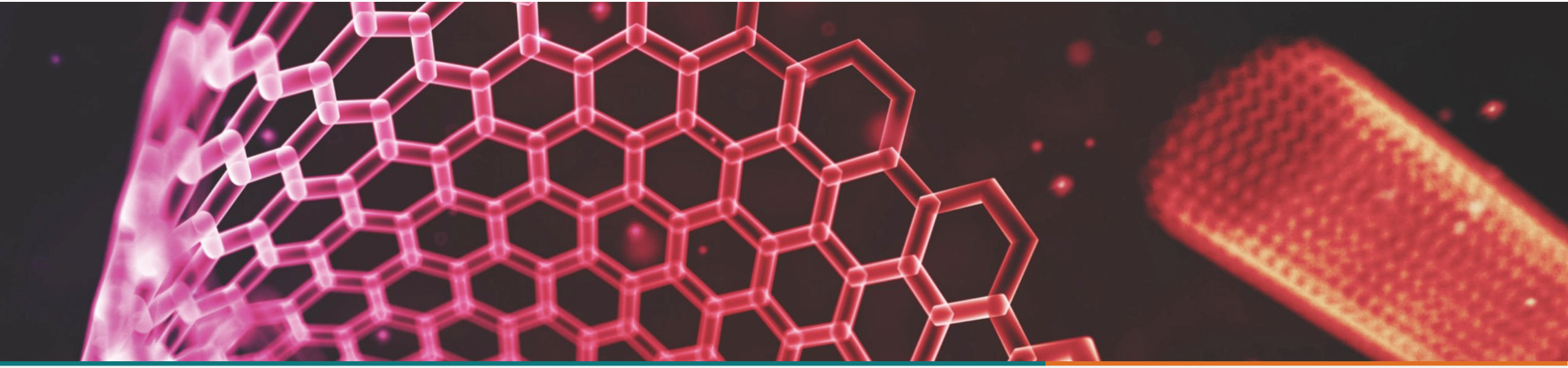
You can require your module by using a relative path in the require function:

```
const calculator = require("./calculator");
```

You can then test it by using exported methods.

- https://github.com/stevens-cs546-cs554/CS-546/blob/master/lecture_02/code/calculator_module_example/app.js

Making a Calculator App



Bringing It All Together

We can now go through the following steps together:

- Make a new folder
- Create a file, app.js
- Run `npm init` and go through the instructions
- Use app.js as your entrance point
- When done, install the prompt package: `npm install prompt`
- Require your module in app.js
- Require prompt
- Begin creating your application!
- Run your app when ready: `npm start`

Using Prompt

Many packages we use this term will require you to read a little documentation.

The prompt package is a simple package that allows you to easily get information from the terminal

- https://www.npmjs.com/package/prompt

You may find the types example on their GitHub repository very helpful

- https://github.com/flatiron/prompt/blob/master/examples/types.js