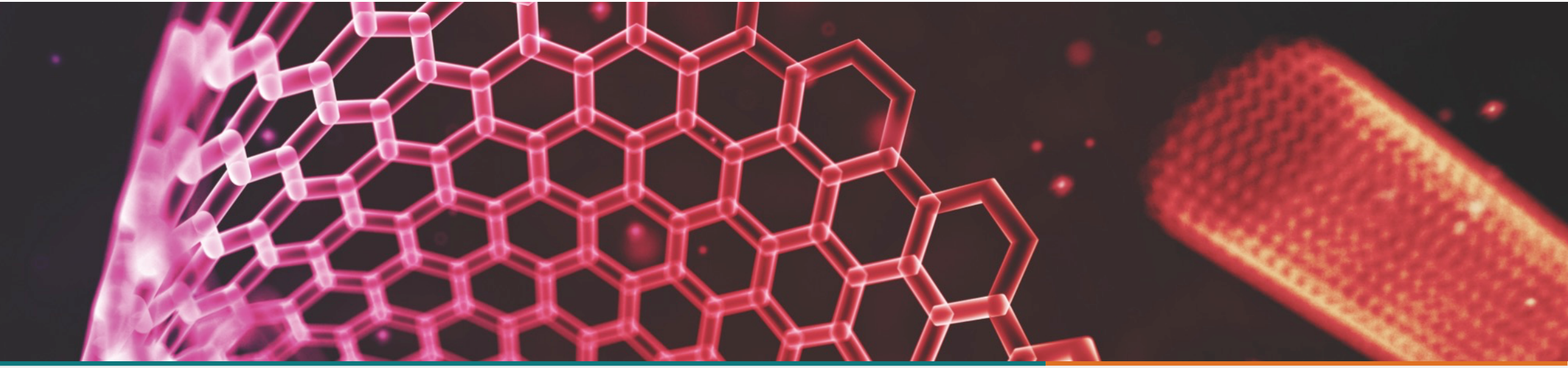
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

CS 546 – Web Programming I
MongoDB



STEVENS

INSTITUTE of TECHNOLOGY

Schaefer School of Engineering & Science

stevens.edu

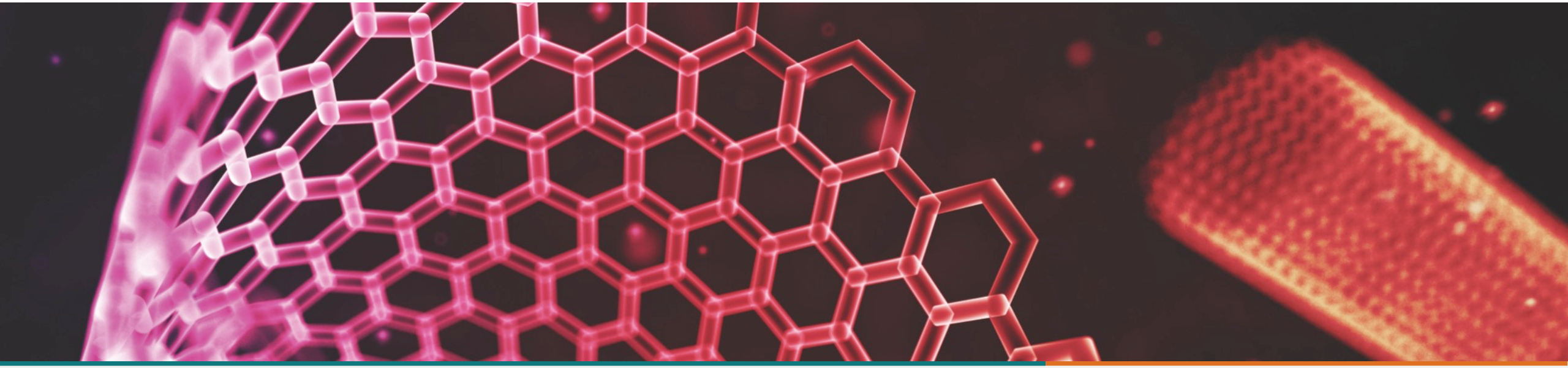
Patrick Hill

Adjunct Professor

Computer Science Department

Patrick.Hill@stevens.edu

Extremely Brief Prelude



JSON

Very often, we use JavaScript style objects to transmit data between processes, servers, systems, etc. The standard followed when transmitting objects is called JSON: JavaScript Object Notation.

Formally, "JSON is a syntax for serializing objects, arrays, numbers, strings, booleans, and null".

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

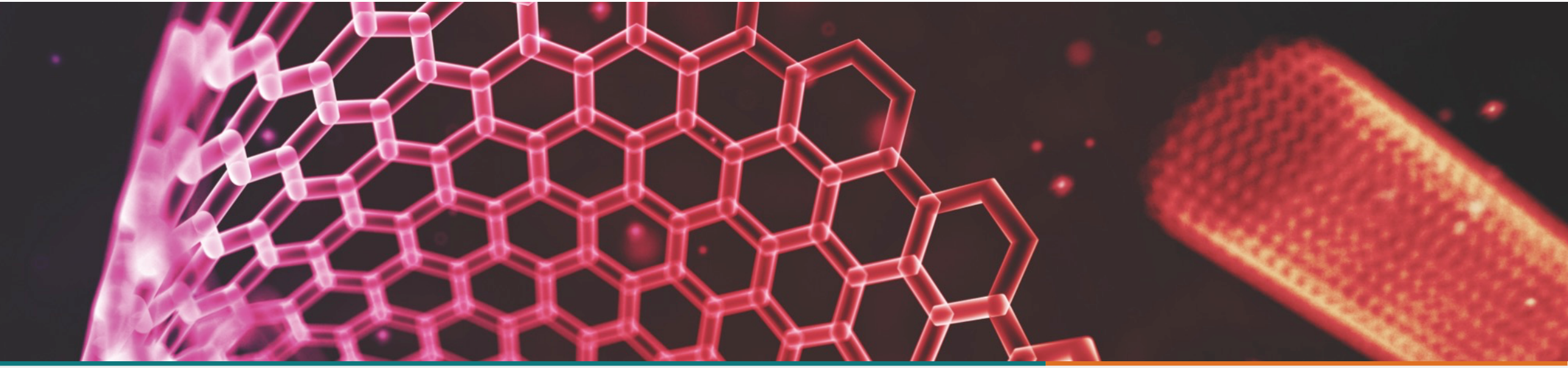
In MongoDB, data is stored in a binary version of JSON. In general, we will be using ***JSON*** for the rest of the semester to translate information between clients, servers, processes, etc.

- We can parse a JSON string to be an object using ***JSON.parse(someJSONstring)***
- We can take a JavaScript object and serialize it to JSON using ***JSON.stringify(myObject)***

JSON

```
{
    "name": "Patrick Hill",
    "education": [{
            "Level": "High School",
            "Name": "Bayside High School"
        },
        {
            "Level": "Undergraduate",
            "Name": "Baruch College"
        },
        {
            "Level": "Graduate",
            "Name": "Stevens Institute of Technology"
        }
    ],
    "hobbies": ["Playing music", "Family"]
}
```

Introduction to Databases



What Is a Database?

A database is an organized collection of data. It allows you to create, read, update, and delete data. Unlike storing in memory, databases allow you to persist your data.

MongoDB is a **document-based database**.

- **Document-based** databases store semi-structured data (think, JSON!) and only lookup by key (a unique identifier)

You interact with MongoDB by submitting queries to your database that describe operations that you wish to do.

What Is a Document-Based Database?

Traditional databases are stored in tables that are composed of columns describing data and rows of data. They have a pre-defined schema to them, constraining the type of data you can store in each table.

Document-based databases forgo this and allow you to simply store and retrieve data at a particular location. They are very good at ID lookups but suffer slightly on querying.

This is less off an issue now than in previous years.

Why MongoDB?

MongoDB is incredibly easy to setup and use, allowing you to focus on web-development as a whole and how all the parts work together, rather than focusing on the nuances of databases.

It stores JSON-like structures, making it easy to conceptualize.

- Easy to have nested objects (subdocuments)

Much like objects, each collection stores data in a dictionary fashion using a key/value pair.

The querying language is composed by JSON that describe queries, making it easy to pick up.

The Structure of MongoDB

MongoDB only has a few layers to it:

1. **Databases**: You can create a database to contain related collections
2. **Collections**: Each database has a number of collections. Collections are sets of documents that you decide are related by their content. Your documents do not have to have the same fields.
3. **Documents**: Documents are self contained pieces of data that you store in a collection. Each document must have an ID and can have any other set of fields; these fields can be smaller subdocuments.
4. **Subdocuments**: A document field can describe another document that will be stored in its parent. This is akin to an object that has a second object stored as a property. This is referred to as a subdocument.

Basic Operations in MongoDB

For now, we will be focusing on four different operations:

1. **Insert**: will take an object and insert it into the database.
2. **Find**: will take an object describing fields and values to match and returns an array of matching documents.
3. **Update**: will take two objects; one that contains an object describing fields and values to match, and one that will describe the update to perform. It can update multiple if you provide a third object with settings telling it to update multiple documents.
4. **Remove**: will take an object describing fields and values to match and will remove the object. It can remove multiple if you provide a second object with settings telling it to update multiple documents.

Demonstration

https://github.com/stevens-cs546-cs554/CS-546/tree/master/lecture_04/code

The first thing you should take note of is a file called ***mongoConnection.js***

- This file allows you to create one shared database connection for your entire app!

The second thing you should take note of is a file called ***mongoCollections.js***

- In this file, you can setup a module that will export async functions (promises) that resolve to database collections; this ensures that the connection is working and allows you to organize code better.

The third thing you should notice is ***dogs.js***

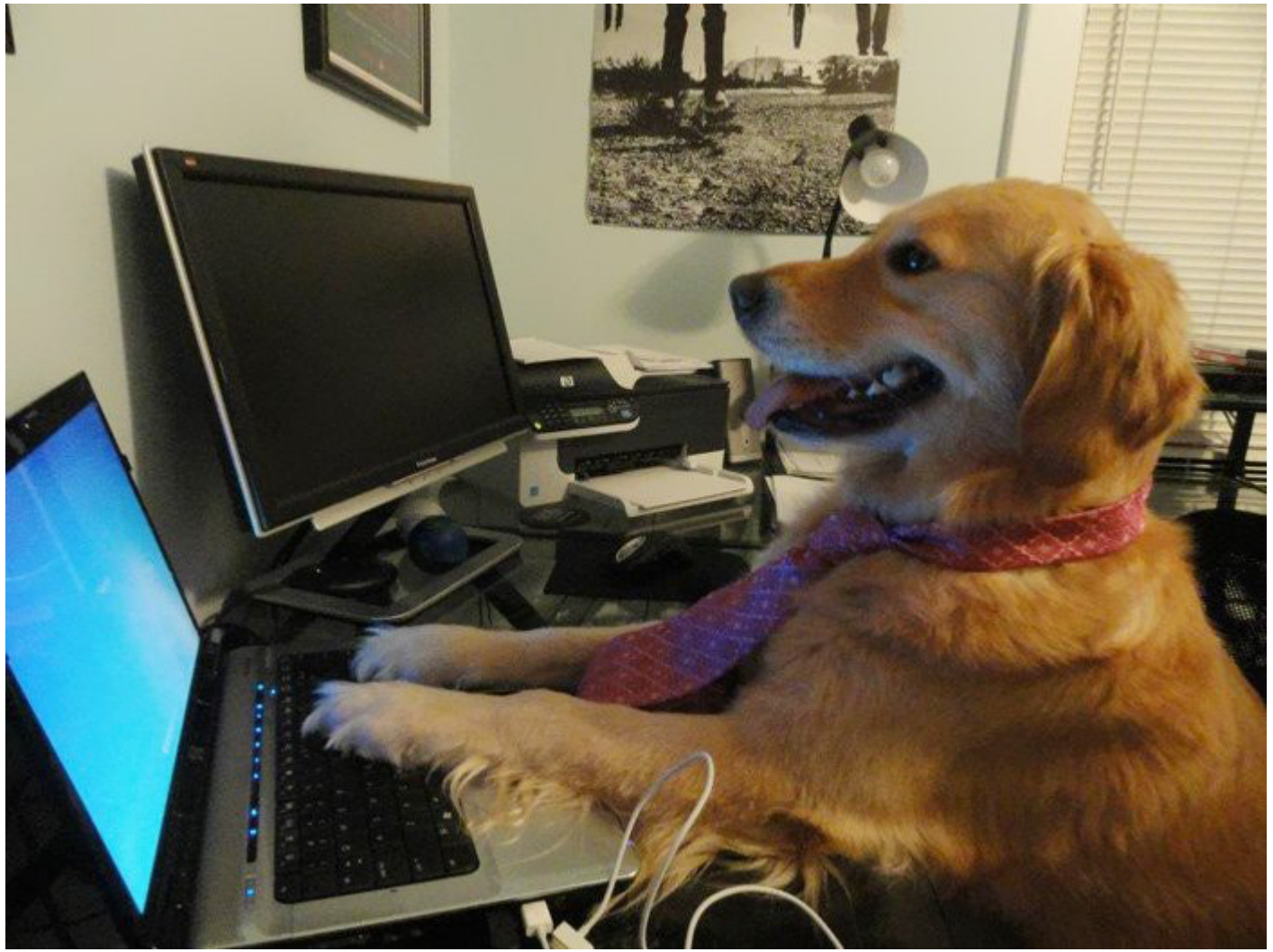
- This file sets up a module that shows the basic pattern of Create, Read, Update, and Delete!

Demonstration This Week

This week, we will be looking at a dog blog. Yes, for dogs that blog.

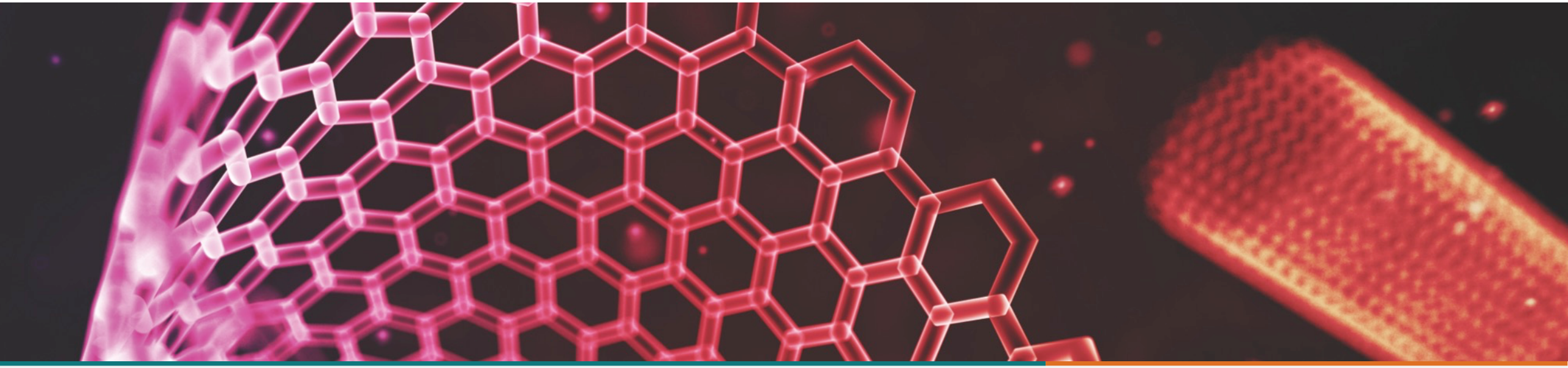
We will be following code that takes us through the blogs of 3 dogs:

1. Sasha, a Cheagle (Chihuahua-Beagle) that runs a blog about different types of cheese.
2. Max, a grizzled ex-cop Mastiff with a heart of gold.
3. Pork Chop, a confused Golden Labrador Retriever that, coincidentally, does not like pork and is fonder of turkey.

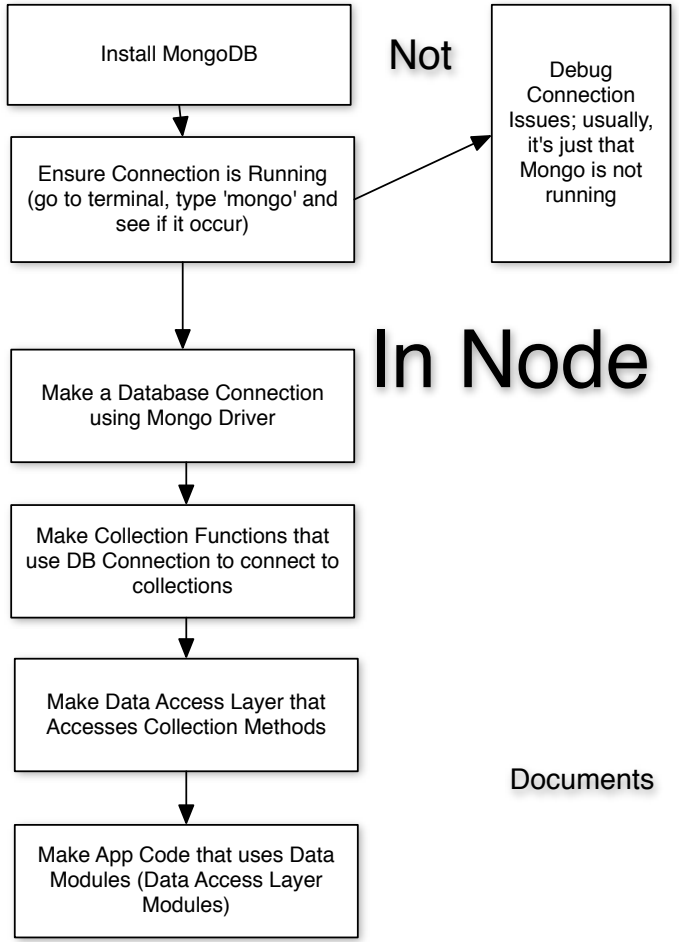


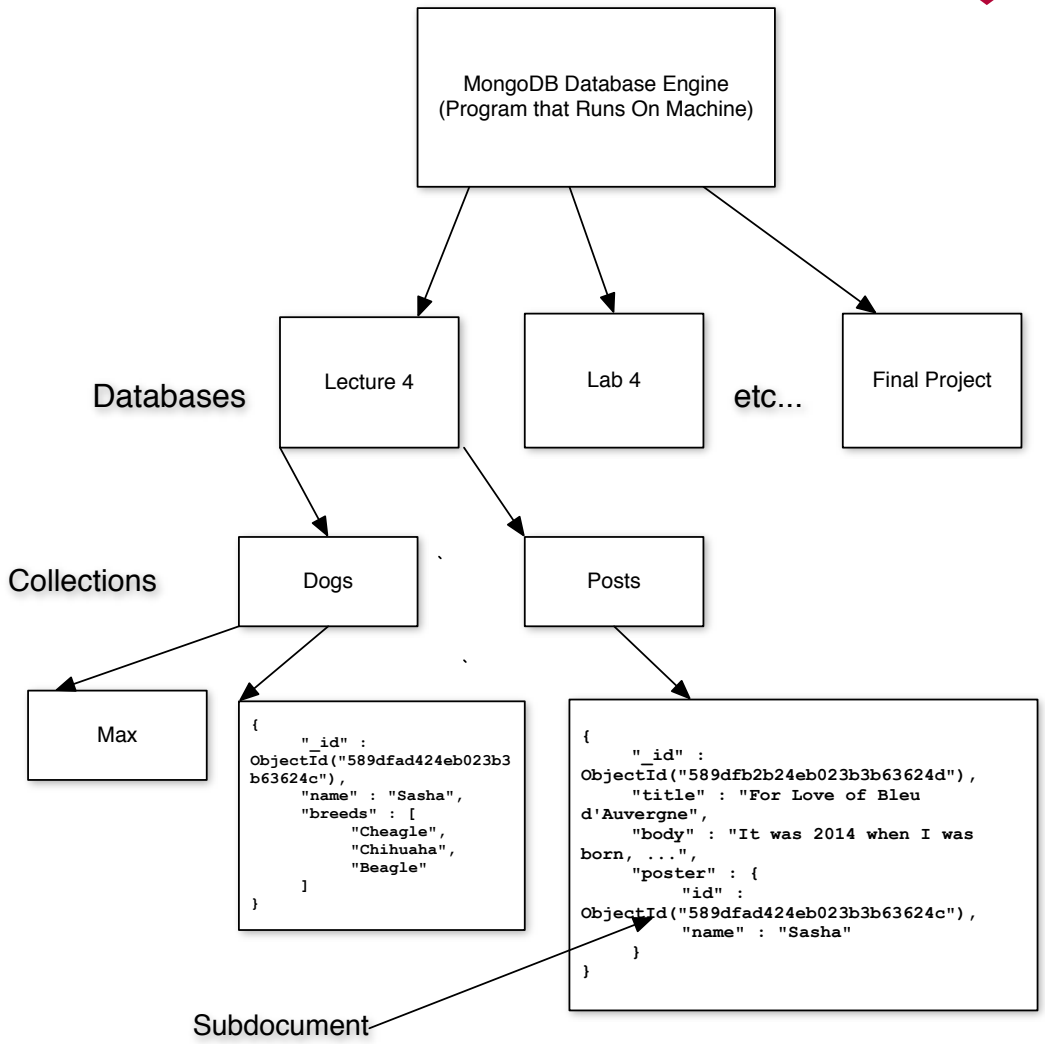
In honor of the previous professor, Phil Barresi, I have left his dog blogging pic

Connecting to MongoDB



MongoDB Flow





Installing the MongoDB Driver Package

We will be using the official MongoDB driver released by the official Mongo team.

- Other drivers add complexity that we need not worry about for the scope of this course.

```
npm install mongodb
```

Using the driver is simple, since MongoDB natively uses JSON to query and store documents.

- We will be using this driver in order to setup connections do basic querying against MongoDB.
- http://mongodb.github.io/node-mongodb-native/
- See the API section for the most recent driver for documentation!

The MongoDB driver methods return promises, which means we can `await` them from inside of `async` functions.

Connecting to Your Database

Once we deal with installing the package to the project and requiring it, we can create a database connection. This is demonstrated in `mongoConnection.js` file we saw before.

The basic algorithm of using a database is simple:

- You use connection information in order to connect to the database
- You store this database connection in a variable and use it to access collections
- You store this collection reference in a variable and use it to access collection operations

Connecting to Your Database

```
const MongoClient = require("mongodb").MongoClient;
const settings = require("./settings");
const mongoConfig = settings.mongoConfig;

let _connection = undefined;
let _db = undefined;

module.exports = async() => {

  if (!_connection) {
    _connection = await MongoClient.connect(mongoConfig.serverUrl,{useNewUrlParser:true,useUnifiedTopology: true
    });
    _db = await _connection.db(mongoConfig.database);
  }

return _db;
};
```

Creating and Using a Collection

In order to create, read, update, or delete documents you must first have a collection. Collections will be automatically created when a document is first inserted into a new collection.

We can retrieve data from a collection by retrieving a reference to the collection and querying it, which you can see in ***`dogs.js`***

We need to set up our collections in ***`mongoCollections.js`***

Creating and Using a Collection

```
const dbConnection = require("./mongoConnection");

/* This will allow you to have one reference to each collection per app */
/* Feel free to copy and paste this this */
const getCollectionFn = collection => {
  let _col = undefined;

  return async () => {
    if (!_col) {
      const db = await dbConnection();
      _col = await db.collection(collection);
    }

    return _col;
  };
};

/* Now, you can list your collections here: */
module.exports = {
  posts: getCollectionFn("posts"),
  dogs: getCollectionFn("dogs")
};
```

Abstracting Your Queries

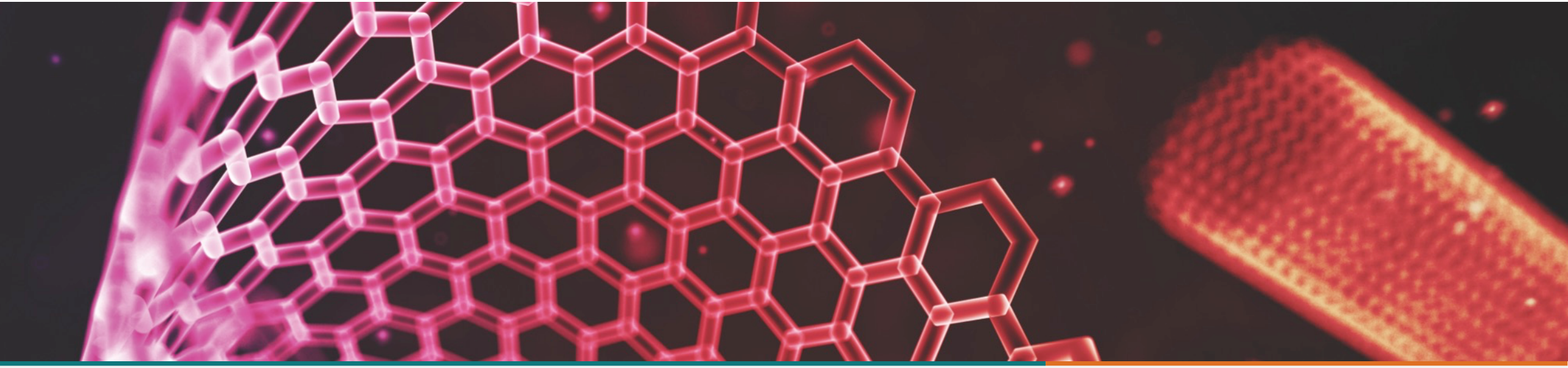
It is often very useful to create a file to abstract away your database querying.

By creating a layer between your application code and your database, you will allow yourself to:

- More easily make changes to the database later on
- Allow non-database programmers to more easily use the database (separation of concerns!)
- More easily improve performance at a later time
- Easier and more consistent error checking throughout your entire application.
- Make it a reasonable task to change your entire database when the first database your company chose ends up being unable to support large amounts of data and you need to transition over to another database.
  - Also helps when this happens again 2 years later.

In `dogs.js` we abstract the queries to hide them away from the rest of our application.

Basic Data Manipulation in MongoDB



Inserting Into Your Collection

Let's take a look at `app.js` now.

In this file, we will start off by creating our dogs, whom will be our bloggers today. We will be calling the methods exported from `dogs.js` in order to use those functions more easily.

- In MongoDB, you insert documents; these documents are just JSON objects; we insert documents into collections, as you can see from the ***`addDog`*** method in `dogs.js`

Inserting Into Your Collection

```
async addDog(name, breeds) {
  if (!name) throw 'You must provide a name for your dog';

  if (!breeds || !Array.isArray(breeds)) throw 'You must provide an array of breeds';

  if (breeds.length === 0) throw 'You must provide at least one breed.';
  const dogCollection = await dogs();

  let newDog = {
    name: name,
    breeds: breeds
  };

  const insertInfo = await dogCollection.insertOne(newDog);
  if (insertInfo.insertedCount === 0) throw 'Could not add dog';

  const newId = insertInfo.insertedId;

  const dog = await this.getDogById(newId);
  return dog;
}
```

Retrieving Data

In MongoDB, you retrieve documents by querying for matching properties. In our case, we query dogs based on their `_id` field. The `_id` field is auto-generated on insert. We can also search for dogs based on matching names, or other properties (to be seen soon!).

The `getDogById` method demonstrates retrieving a single dog that matches the query.

The `getAllDogs` method demonstrates retrieving all dogs from the database.

Retrieving Data

```
async getDogById(id) {
  if (!id) throw 'You must provide an id to search for';

  const dogCollection = await dogs();
  const doggo = await dogCollection.findOne({_id: id});
  if (doggo === null) throw 'No dog with that id';

  return doggo;
}
```

Retrieving Data

```
async getAllDogs() {
  const dogCollection = await dogs();

  const dogs = await dogCollection.find({}).toArray();

  return dogs;
}
```

Updating Data

In MongoDB we can update one or more documents at a time by passing an object that describes what documents you want to match, and a document describing how you want the update to occur.

For now, we will be replacing the document entirely by passing a new version of the document to the update call.

Updating Data

```
async updateDog(id, name, breeds) {
  if (!id) throw 'You must provide an id to search for';

  if (!name) throw 'You must provide a name for your dog';

  if (!breeds || !Array.isArray(breeds)) throw 'You must provide an array of breeds';

  if (breeds.length === 0) throw 'You must provide at least one breed.';

  const dogCollection = await dogs();
  const updatedDog = {
    name: name,
    breeds: breeds
  };

  const updatedInfo = await dogCollection.updateOne({_id: id}, {$set: updatedDog});
  if (updatedInfo.modifiedCount === 0) {
    throw 'could not update dog successfully';
  }

  return await this.getDogById(id);
}
```

Deleting Data

In MongoDB we can delete one or more documents at a time by passing an object that describes what documents you want to match.

```
async removeDog(id) {
  if (!id) throw 'You must provide an id to search for';

  const dogCollection = await dogs();
  const deletionInfo = await dogCollection.removeOne({_id: id});

  if (deletionInfo.deletedCount === 0) {
    throw `Could not delete dog with id of ${id}`;
  }
}
```

Questions?