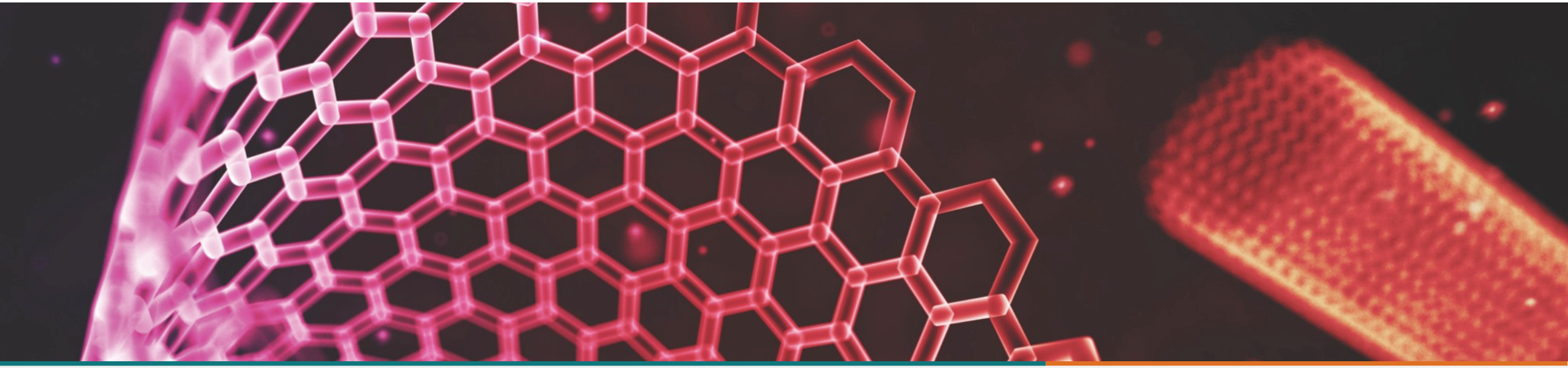
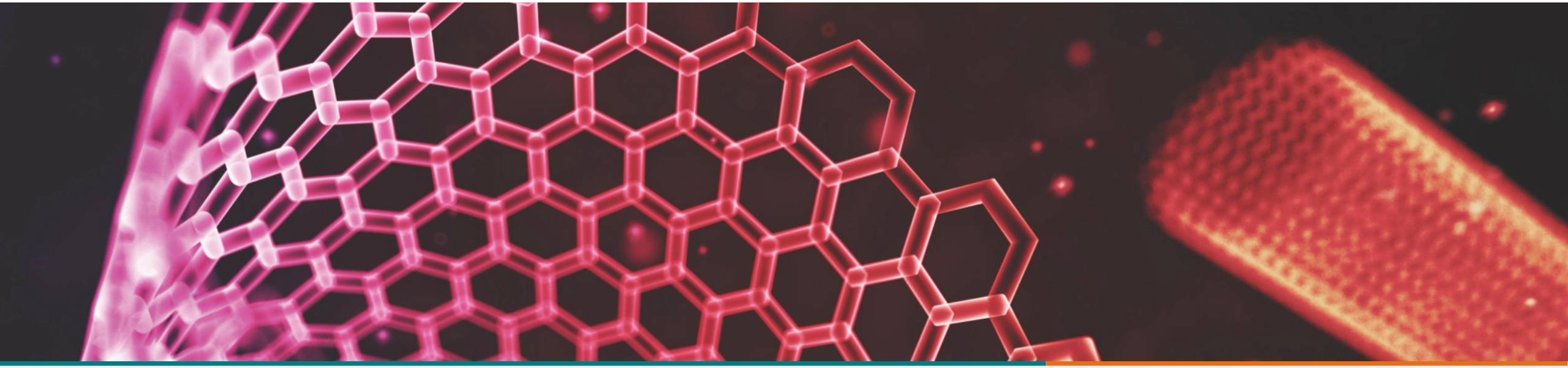
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

# CS 546 – Web Programming I

# API Development and Intermediate MongoDB



# Intermediate MongoDB



# Demonstration

In Lecture 6's repository, see ***advanced_mongo.js*** for examples. In this file, a module is exported detailing many of the functions listed.

I would recommend running node in the command line, requiring ***advanced_mongo.js***, and experimenting with it accordingly. Or, you can write your own driver to experiment.

**Note: the data for this collection will rebuild itself every time you require the file, and for simplicity's sake the id's are being stored as integers. At the end of every function, the changes will be logged. Feel free to change this!**

# Advanced Querying

We can find documents many more ways than just matching on multiple fields:

- Query by subdocuments.
- Query for matches inside an array.
- Query for a field to be one of many values.
- Matching fields that are less than (or equal to) a value.
- Matching fields that are greater than (or equal to) a value.
- Performing a logical query for all matching queries, or any matching queries.
- JavaScript based querying!

We can also do things like:

- Limiting and Skipping documents
- Returning only certain fields
- Sorting

# Advanced Updating

There are many ways we can update documents, rather than just replacing their entire content.

- We can change only specific fields
- Update subdocuments
- Increment fields
- Multiply fields value
- Remove fields
- Update to a minimum value
- Update to a maximum value
- Manipulate arrays

All of these are demonstrated in ***advanced_mongo.js***, where you can experiment with them accordingly through the node command line or writing your own file.

# Array Querying Operations

Naturally, as JSON documents, we can store arrays in MongoDB.

- Entries can be primitives or objects!

We can query documents based on arrays and update arrays and their entries. When dealing with arrays containing subdocuments, we can query for matching fields on subdocuments.

We can query arrays to find documents that have arrays with matching entries.

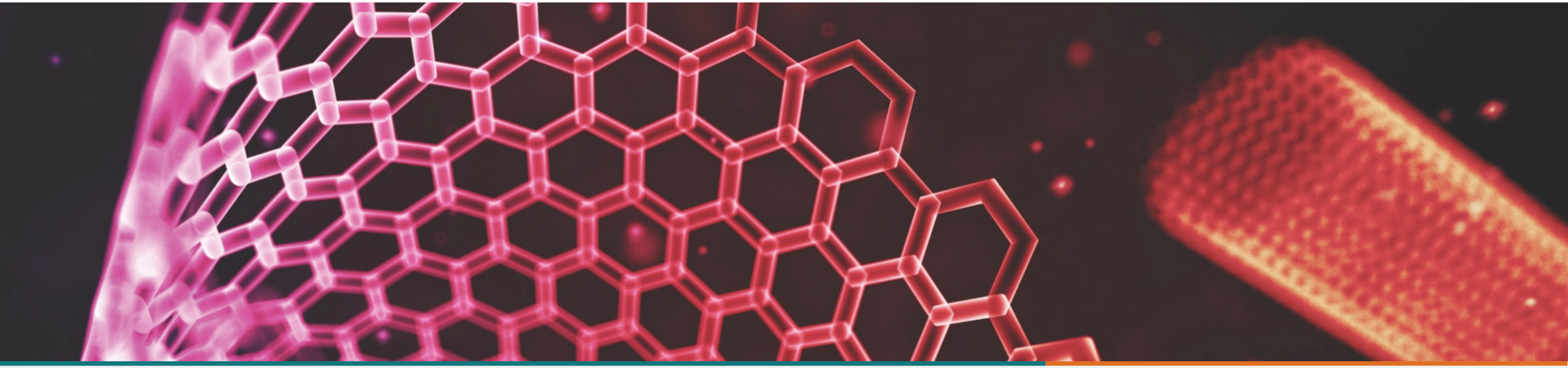
# Array Manipulation Operations

Arguably, the most difficult part of MongoDB is array manipulation due to the complex syntax of  combining arrays and subdocuments.

There are many ways of updating arrays:

- Adding to the array if it does not already exist
- Adding to the array whether or not it exists
- Popping the first or last element
- Remove a single matching element
- Removing all matching elements

# Comparison Query Operations



# Comparison Query Operations

MongoDB has a number of Comparison Query Operations:

| | |
|---|---|
| `$eq` | Matches values that are equal to a specified value. |
| `$gt` | Matches values that are greater than a specified value. |
| `$gte` | Matches values that are greater than or equal to a specified value. |
| `$in` | Matches any of the values specified in an array. |
| `$lt` | Matches values that are less than a specified value. |
| `$lte` | Matches values that are less than or equal to a specified value. |
| `$ne` | Matches all values that are not equal to a specified value. |
| `$nin` | Matches none of the values specified in an array. |

# $eq

Specifies equality condition. The $eq operator matches documents where the value of a field equals the specified value.

```
{ <field>: { $eq: <value> } }
```

# $gt

$gt selects those documents where the value of the field is greater than (i.e. >) the specified value.

```
db.inventory.find( { qty: { $gt: 20 } } )
```

```
db.inventory.update( { "carrier.fee": { $gt: 2 } }, { $set: { price: 9.99 } } )
```

# $gte

$gte selects the documents where the value of the field is greater than or equal to (i.e. >=) a specified value (e.g. value.)

```
db.inventory.find( { qty: { $gte: 20 } } )

db.inventory.update( { "carrier.fee": { $gte: 2 } }, { $set: { price: 9.99 } } )
```

# $in


The $in operator selects the documents where the value of a field equals any value in the specified array. To specify an $in expression, use the following prototype:

For comparison of different BSON type values, see the specified BSON comparison order.

```
{ field: { $in: [<value1>, <value2>, ... <valueN> ] } }
```

# $lt

$lt selects the documents where the value of the field is less than (i.e. <) the specified value.

```
db.inventory.find( { qty: { $lt: 20 } } )
```

```
db.inventory.update( { "carrier.fee": { $lt: 2 } }, { $set: { price: 9.99 } } )
```

# $lte

$lte selects the documents where the value of the field is less than or equal to (i.e. <=) the specified value.

```
db.inventory.find( { qty: { $lte: 20 } } )
```

```
db.inventory.update( { "carrier.fee": { $lte: 2 } }, { $set: { price: 9.99 } } )
```

# $ne

$ne selects the documents where the value of the field is not equal to the specified value. This includes documents that do not contain the field.

**db.inventory.find( { qty: { $ne: 20 } } )**

**db.inventory.update( { "carrier.state": { $ne: "NY" } }, { $set: { qty: 20 } } )**

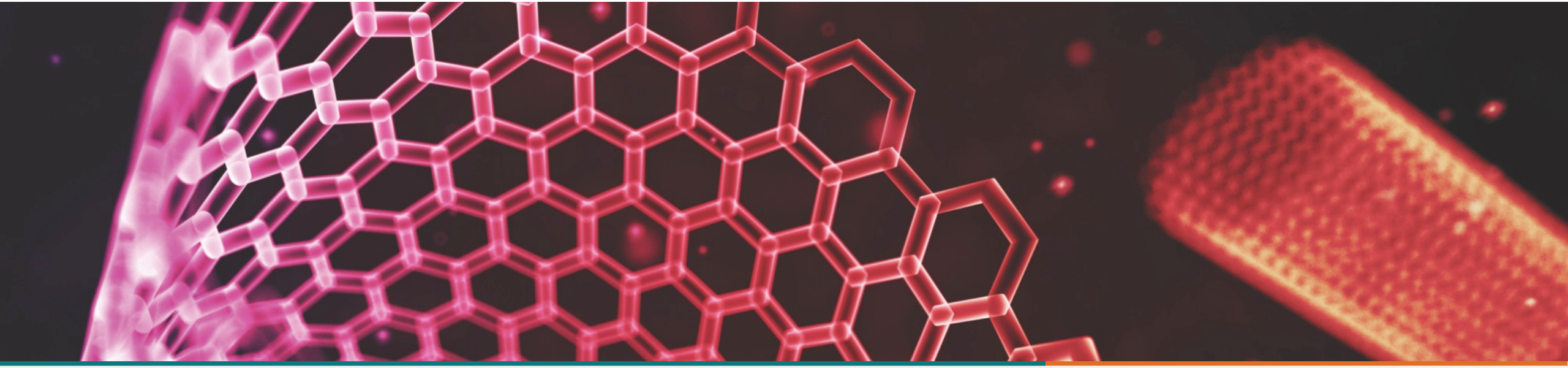
# $nin

$nin selects the documents where:

- the field value is not in the specified array **or**
- the field does not exist.

**db.inventory.find( { qty: { $nin: [ 5, 15 ] } } )**

# Logical Query Operations



# Logical Query Operations

MongoDB has a number of Logical Query Operations:

| Name | Description |
|---|---|
| `$and` | Joins query clauses with a logical **AND** returns all documents that match the conditions of both clauses. |
| `$not` | Inverts the effect of a query expression and returns documents that do *not* match the query expression. |
| `$nor` | Joins query clauses with a logical **NOR** returns all documents that fail to match both clauses. |
| `$or` | Joins query clauses with a logical **OR** returns all documents that match the conditions of either clause. |

# $and


$and performs a logical AND operation on an array of *one or more* expressions (e.g. <expression1>,<expression2>, etc.) and selects the documents that satisfy ***all*** the expressions in the array.

The $and operator uses *short-circuit evaluation*. If the first expression (e.g. <expression1>) evaluates to false, MongoDB will not evaluate the remaining expressions.

```
db.inventory.find( { $and: [ { price: { $ne: 1.99 } }, { price: { $exists: true } } ] } )
```

# $not

$not performs a logical ***NOT*** operation on the specified <operator-expression> and selects the documents that ***do not*** match the <operator-expression>.

This includes documents that do not contain the field.

```
db.inventory.find( { price: { $not: { $gt: 1.99 } } } )
```

# $nor


$nor performs a logical NOR operation on an array of one or more query expression and selects the documents that **fail** all the query expressions in the array.

The $nor has the following syntax:

```
{ $nor: [ { <expression1> }, { <expression2> }, ...  { <expressionN> } ] }
```

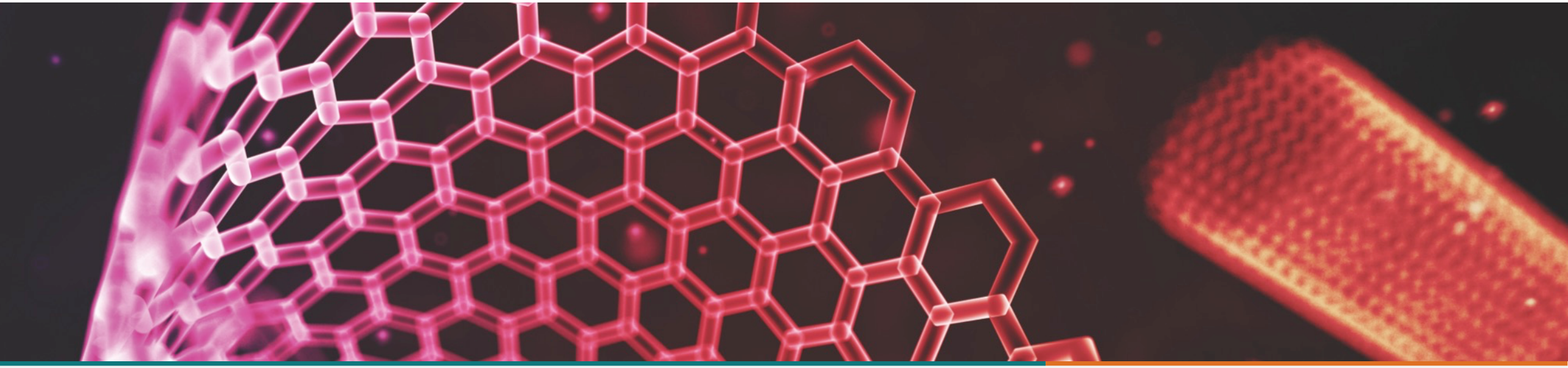
# $or

The $or operator performs a logical OR operation on an array of *two or more* <expressions> and selects the documents that satisfy *at least* one of the <expressions>.

The $or has the following syntax:

```
{ $or: [ { <expression1> }, { <expression2> }, ... , { <expressionN> } ] }
```

# Skipping and Limiting the Number of Documents Returned



# Limit

When we query a collection, it shows all the documents contained in it. The limit cursor is used for retrieving only numbers of documents that we need. We can use MongoDB **limit()** method to specify the number of documents we want it to return.

```
const movieCollection = await movies();

const movieList = await movieCollection.find({}).limit(20).toArray();

return movieList;
```

# Skip

When we use the MongoDB ***limit()*** method, it shows from the beginning of the collection to the specified limit. If we want it to start not from the beginning and skipping some documents, we can use the ***skip()*** method for this task.

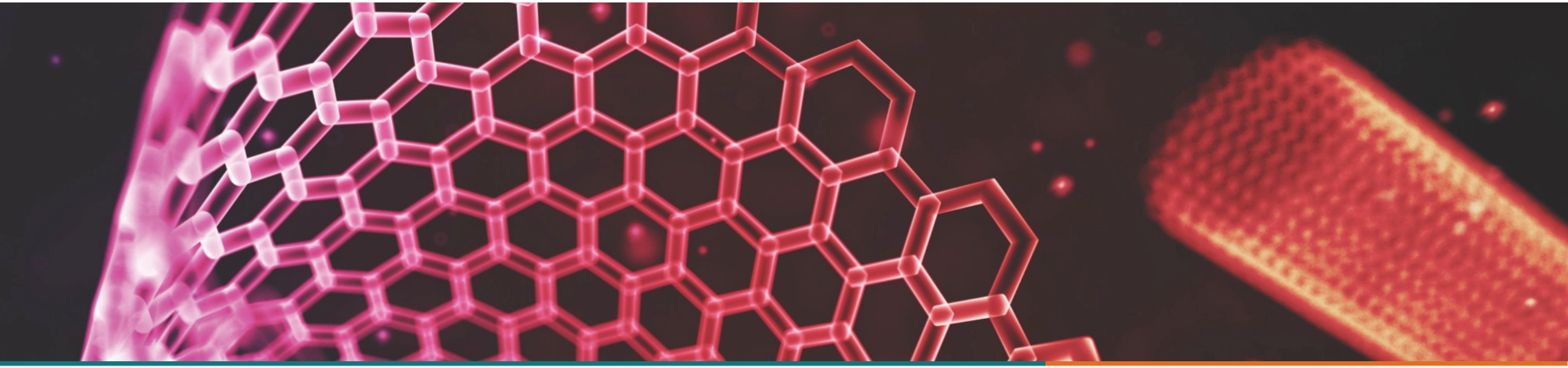
To do this, we need to add the **skip()** in the cursor and specify the number of documents that we want to skip.

```
const movieCollection = await movies();

const movieList = await movieCollection.find({}).skip(10).toArray();

return movieList;
```

We can use limit and skip together:

```
const movieCollection = await movies();

const movieList = await movieCollection.find({}).skip(10).limit(20).toArray();

return movieList;
```

# Returning Only Certain Fields




```
{
        _id: 1,
        title: 'Inception',
        rating: 4.5,
        reviews: [
                {
                _id: '9e6b393c-4678-4fa3-a714-5c3bba0331cc',
                title: 'Really Good',
                comment: 'This movie was so interesting.',
                reviewer: 'Phil',
                rating: 4.5
                },
                {
                _id: '13fa5b13-6d1f-4807-a033-b2a5cd2c3e22',
                title: 'Bad',
                comment: 'This movie is trite.',
                reviewer: 'Agatha',
                rating: 2
                },
                {
                _id: 'ac0fcaf2-8899-4ddb-bec6-eb7f0f283db4',
                title: 'Perfect',
                comment: 'Leo should win an Oscar for this.',
                reviewer: 'Definitely Not Leo',
                rating: 4
                }
        ],
        cast: ['Leonardo DiCaprio','Ellen Page','Ken Watanabe','Joseph Gordon-Levitt','Marion Cotillard','Tom Hardy'
        ],
        info: {
                release: 2015,
                director: 'Christopher Nolan'
        }
}
```

# Returning Certain Fields

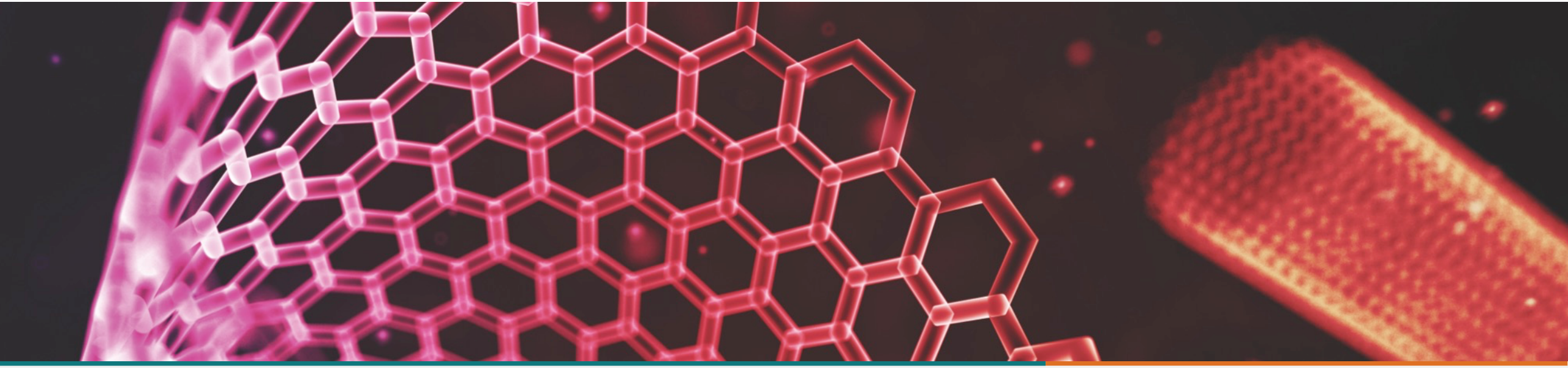
By default, queries in MongoDB return all fields in matching documents. To limit the amount of data that MongoDB sends to applications, you can include a projection document to specify or restrict fields to return.

```
const movieCollection = await movies();

const movieList = await movieCollection

.find({}, { projection: { _id: 0, title: 1, 'info.director': 1, rating: 1, cast: 1 } }).toArray();
return movieList;
```

This will **ONLY** return the fields, title, info.director(a part of a subdocument), rating and the array of cast members.

We can include fields by using the value 1 and exclude fields by using a value of 0

# Sorting



# Sorting

Specifies the order in which the query returns matching documents. You must apply sort() to the cursor before retrieving any documents from the database.

```
getAllMoviesSortedByTitleAsc: async () => {
const movieCollection = await movies();
const movieList = await movieCollection.find({}).sort({ title: 1 }).toArray();
return movieList;
},
getAllMoviesSortedByTitleDec: async () => {
const movieCollection = await movies();
const movieList = await movieCollection.find({}).sort({ title: -1 }).toArray();
return movieList;
}
```

# Sorting

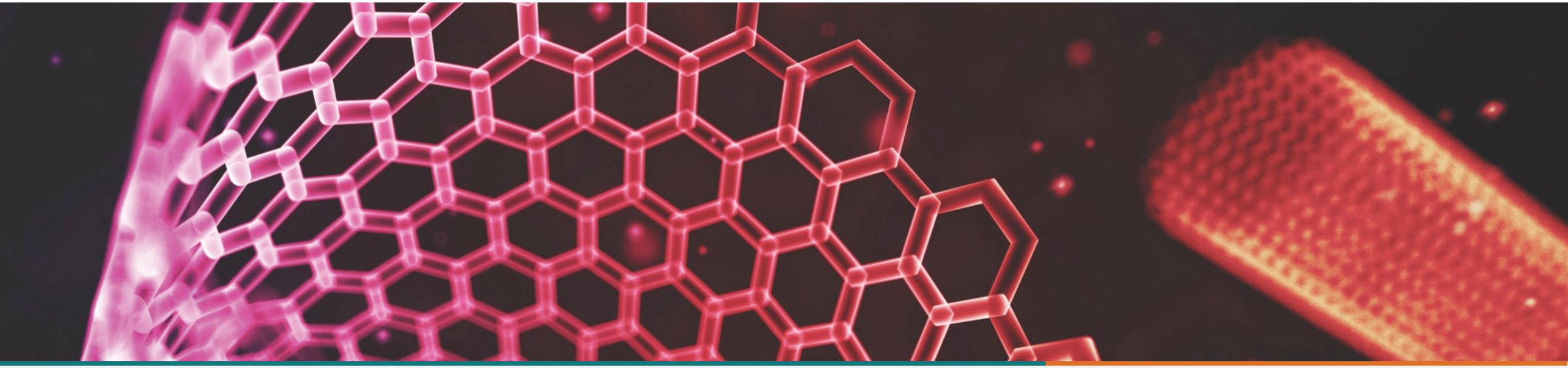

Sorted by Title, then release year

```
const movieCollection = await movies();

const movieList = await movieCollection.find({}).sort({ title: 1, 'info.release': 1 }).toArray();

return movieList;
```

We can also use limit and skip with sort:

```
const movieCollection = await movies();
const movieList = await movieCollection
.find({})
.skip(1)
.limit(2)
.sort({ title: 1, 'info.release': 1 })
.toArray();
return movieList;
```

# API: POST, PUT, PATCH, DELETE



# POST, PUT, PATCH, DELETE

- **POST**
  - The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.
- **PUT**
  - The PUT method replaces all current representations of the target resource with the request payload.
- **PATCH**
  - Similar to PUT, but you can replace portions of the resource instead of the whole resource.
- **DELETE**
  - The DELETE method deletes the specified resource.

Each of these request types can use the following types of data:

- Querystring parameters
- Request bodies
- URL Params
- Headers

# The Request Body

**POST**, **PUT**, **PATCH** and **DELETE** requests can all provide data in a request body.

A request body is a series of bytes transmitted below the headers of an HTTP Request.

We will be submitting a request body in two ways:

- Text that is in a JSON format (modern format of submitting data)
- Text that is in a form data format (traditionally how browsers **POST**)

The request body will be interpreted by our server using the ***`express.json`*** middleware Function that is built into Express

- https://expressjs.com/en/api.html#express.json

# Using Request Body Data

In order to access request body data, we must first apply the ***express.json*** middleware.

This will allow us to add text that is formatted as JSON to a request body, and to have our server parse the JSON and place the object in the ***request.body*** property.

This will allow us to submit data with our POST, PUT, PATCH and DELETE calls and begin interacting with our server.

- https://expressjs.com/en/api.html#express.json

# Using Postman

As we use more methods, such as **POST**, **PUT**, **PATCH** and **DELETE**, it becomes increasingly difficult to test using just your browser, particularly because you cannot directly **PUT**, **PATCH** and **DELETE** from the browser!  The browser only knows a **GET** and **POST** request.

You can use a REST client such as Postman and PAW to test your API calls.

- https://www.getpostman.com/
- https://luckymarmot.com/paw

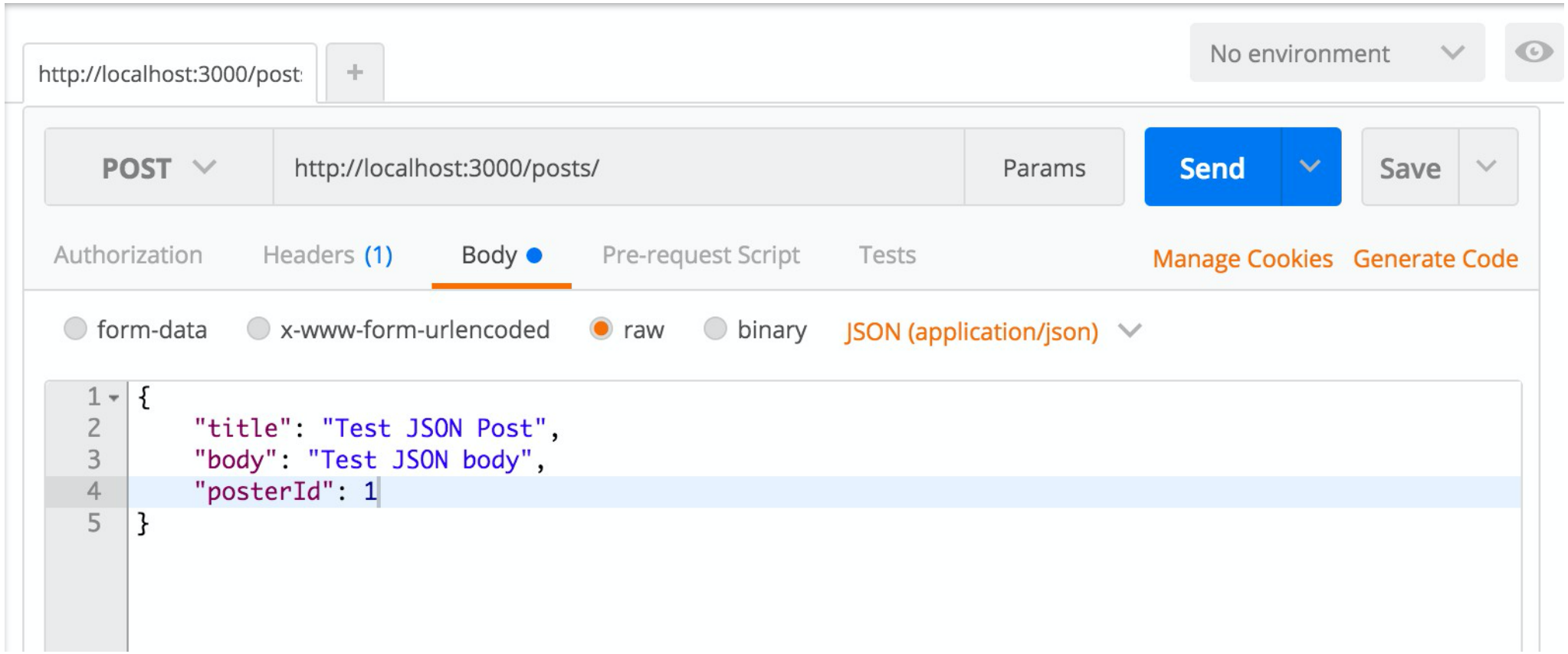
A REST client is a program that will allow you to easily configure and make HTTP Calls to your servers.

# Using Postman to Send JSON Data

In order to use Postman, you need:

- The URL you wish to submit data to
- The request method you wish to use
- Body data
  - You must set the body type to raw
  - You must also set the type to JSON (application/json)

# Adding a Blog Post with Postman



# Using the Data That Was Sent in the Request

We can access the data that was sent in the request's body inside the route, we then call our DB function ***addPost()*** to add the post to the DB :

```
router.post('/', async (req, res) => {
    const blogPostData = req.body;
    if (!blogPostData.title) throw 'You must supply a blog title';
    if (!blogPostData.body) throw 'You must supply a blog body';
    if (!blogPostData.posterId) throw 'You must supply a poster ID';
    try {
        const { title, body, tags, posterId } = blogPostData;
        const newPost = await postData.addPost(title, body, tags, posterId);
        res.json(newPost);
    } catch (e) {
        res.status(500).json({ error: e });
    }
});
```

# Updating Data - PUT

There are two ways to update data; **PUT** and **PATCH** the difference between the two is how the data is updated.  With a **PUT** request, all the fields of the object need to be supplied.  For example. Say we have the following object in our DB that we wanted to update:

```
{
    "_id": "5c7f137d10a5c9c2a7cc87d2",
    "title": "The Case of the Stolen Bone",
    "body": "It was 2015 when it happened. Someone stole the bone, and hid it in a hole outside....",
    "poster": {
        "id": "5c7f12e410a5c9c2a7cc87d1",
        "name": "Max"
    }
}
```

# Updating Data - PATCH

With a **PUT** request, all the fields need to be supplied in the request body, if they are not supplied, you would need to throw an error. You can think of a **PUT** request as a full replacement of the object, so **all the fields in the object need to be supplied in the request body**. With a **PATCH** request, you only have to supply one or more of the fields in the request body. As long as there is at least one field present, then you can proceed with updating just that field in the DB. So just as it sounds, **PATCH** allows you to "patch" the data, only replacing the data that has changed, as opposed the **PUT** request where the new data replaces the old data completely.

# Updating Data – PUT Route

```
router.put('/:id', async (req, res) => {
    const updatedData = req.body;
    if (!updatedData.title || !updatedData.body || !updatedData.posterId || !updatedData.tags)
        throw 'All fields need to be supplied';
    try {
        await postData.getPostById(req.params.id);
    } catch (e) {
        res.status(404).json({ error: 'Post not found' });
        return;
    }

    try {
        const updatedPost = await postData.updatePost(req.params.id, updatedData);
        res.json(updatedPost);
    } catch (e) {
        res.status(500).json({ error: e });
    }
});
```

# Updating Data – PATCH Route

```
router.patch('/:id', async (req, res) => {
    const requestBody = req.body;
    let updatedObject = {};
    try {
        const oldPost = await postData.getPostById(req.params.id);
        if (requestBody.title && requestBody.title !== oldPost.title) updatedObject.title = requestBody.title;
        if (requestBody.body && requestBody.body !== oldPost.body) updatedObject.body = requestBody.body;
        if (requestBody.tags && requestBody.tags !== oldPost.tags) updatedObject.tags = requestBody.tags;
        if (requestBody.posterId && requestBody.posterId !== oldPost.posterId)
            updatedObject.posterId = requestBody.posterId;
    } catch (e) {
        res.status(404).json({ error: 'Post not found' });
        return;
    }

    try {
        const updatedPost = await postData.updatePost(req.params.id, updatedObject);
        res.json(updatedPost);
    } catch (e) {
        res.status(500).json({ error: e });
    }
});
```

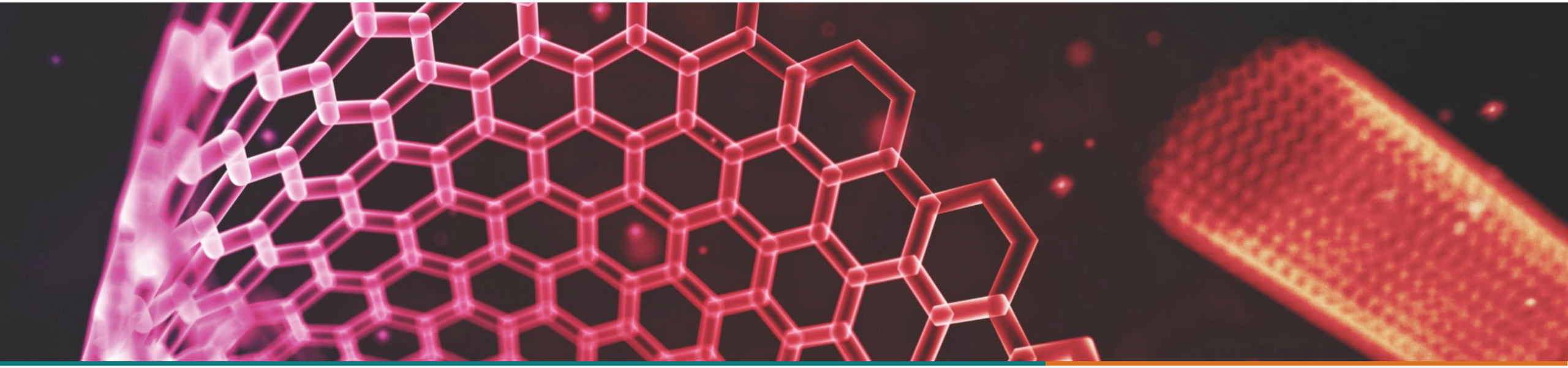
# Deleting Data

Informing your server that you want to delete an entity is extremely easy. Much like **PUT**, you would send a **DELETE** call to a URL that contains the identifier.

That means to delete a blog post with an id of 3 you would **DELETE** to http://localhost:3000/blog/3

```
router.delete('/:id', async (req, res) => {
  try {
    await postData.getPostById(req.params.id);
  } catch (e) {
    res.status(404).json({ error: 'Post not found' });
  return;
  }
  try {
    await postData.removePost(req.params.id);
    res.sendStatus(200);
  } catch (e) {
    res.status(500).json({ error: e });
  }
});
```

# Server-Side Error Checking



# What is Server-Side Validation?

Users will submit errors; it's a fact of life that as a web developer, you will encounter situations where an error is submitted.

There are many types of errors that can occur:

- The user tries to request a resource that does not exist
- The user inputs data that does not make sense (bad arguments/parameters/ querystring data)
- The user is not authenticated
- The input the user provides does not make sense
- The user is attempting to access resources they do not have access to

# Server-Side Error Checking

Whenever input comes from a user, you must check that this input is:

- Actually there!
- Actually the type you want!
  - For example, you may have to change from strings to numbers
- Actually valid!
  - When you write a calculator that you wouldn't let someone divide by 0

**There are two places you will need to perform error checking:**

- Inside of your routes; this will easily catch user submitted errors
- Inside of your data modules; this will allow you to ensure that you don't create bad data.

# Error Handling in an API

While we build out these APIs, error handling is extremely easy!  When you encounter an issue in your API routes, you will:

- Determine what type of error it is (i.e., the user is requesting an object that does not exist) and respond with the proper status code.
- In addition to the failed status code, also send back a JSON object that describes what happened. It can be as simple as having a property called errorMessage with a string describing the error, or an array of all the errors!

# Questions?