Using MySQL to Store API Data

- At its heart, every API is about data, and nearly all APIs store data in some form.

- How an API stores and organizes data has important implications on the API's performance and general usefulness, so it's important to talk about how to do this effectively. This means talking about databases.

- Two broad categories of databases are widely used today: ***relational databases***, in which data is stored in tables of rows and columns, and ***document-oriented databases***, in which data is stored as semi-structured objects much like objects in JSON or XML. In this course, we will study both kinds of database.

- We'll start here with relational databases. There are various relational databases in use today, but MySQL is probably the most popular. With that in mind, we'll focus here on MySQL.

- MySQL is a networked service: it is run by a server process and is accessed by clients over a network. That means to use MySQL for an API, we'll have to have a MySQL server running.

- There are various ways to gain access to a MySQL server, including installing and running MySQL on your own server or using one of many available hosted MySQL services. Here, we'll run a MySQL server using Docker.

- Let's dive in and launch a Docker container running a MySQL server in detached mode. We'll use the official MySQL image to do this:

```
docker run -d --name mysql-server           \
     --network mysql-net                    \
     -p "3306:3306"                         \
     -e "MYSQL_RANDOM_ROOT_PASSWORD=yes"    \
     -e "MYSQL_DATABASE=bookaplace"         \
     -e "MYSQL_USER=bookaplace"             \
```

```
-e "MYSQL_PASSWORD=hunter2"              \
mysql:5
```

- A few things to note about the command we just ran:
  - We're using MySQL version 5, since the Node.js package we'll use to connect to MySQL currently has problems authenticating with MySQL 8.
  - We're assuming a bridged network named `mysql-net` is already created.
  - Port 3306 is the default MySQL port.
  - We're specifying several environment variables to help initialize the MySQL server running in this container:
    - `MYSQL_RANDOM_ROOT_PASSWORD=yes` – this tells the `mysql` image to automatically generate a random password for the MySQL root superuser. Once generated, this password can be accessed via the container's Docker logs, e.g.:

      ```
      $ docker logs mysql-server 2>&1 | grep PASSWORD
      GENERATED ROOT PASSWORD: ...
      ```

      - Instead of generating a random password, you can explicitly set the password using the environment variable `MYSQL_RANDOM_ROOT_PASSWORD`. One of these environment variables *must* be specified.
    - `MYSQL_DATABASE` – this specifies the name of a database to be created when the container starts. One MySQL server can manage many databases. This environment variable is optional.
    - `MYSQL_USER` and `MYSQL_PASSWORD` – these combine to create a new user with a password. The created user is given superuser permissions for the database specified by `MYSQL_DATABASE`.

Connecting to a MySQL server and running queries

- Eventually, we will see how to connect to our MySQL server from our Node.js API server. However, to start out, let's just connect to it using the ***MySQL terminal monitor***, a terminal-based shell from which we can run MySQL commands.

- We can run the terminal monitor using the MySQL Docker image as well, again using MySQL version 5:

```
docker run --rm -it      \
```

```
    --network mysql-net \
    mysql:5             \
         mysql -h mysql-server -u bookaplace -p
```

- The `mysql` command we run here (from the `mysql` image) launches the terminal monitor. The options are:
  - `-h` – specifies the hostname of the MySQL server. Here, we can refer to our MySQL server container by its name, `mysql-server`, since the containers are on the same bridged network.
  - `-u` – specifies the user with which to connect to the server, the `bookaplace` user we created above.
  - `-p` – tells the MySQL server to prompt us to enter the user's password.

- Once connected, we will enter a shell whose command prompt looks like this:

```
mysql>
```

- From this prompt, we can enter queries to the server. Queries are written in SQL (structured query language). For example, we could enter a query to see the current MySQL user and the current date:

```
mysql> SELECT USER(), CURRENT_DATE;
+----------------------+--------------+
| USER()               | CURRENT_DATE |
+----------------------+--------------+
| bookaplace@172.19.0.3 | 2018-04-26   |
+----------------------+--------------+
1 row in set (0.00 sec)
```

- To begin storing data, we need to use a database. We can see what databases are available to us with a `SHOW DATABASES` statement:

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| bookaplace         |
| information_schema |
+--------------------+
```

```
2 rows in set (0.01 sec)
```

- Here, we can see the `bookaplace` database we created when we launched the MySQL Docker container along with a system database `information_schema`.
- To begin to use the `bookaplace` database, we can use a `USE` statement:

```
mysql> USE bookaplace;
```

- Now, all of our data interactions will be with this database. If we wanted to switch databases, we could enter another `USE` statement.
    - As an alternative to running a `USE` statement, you can specify the database on the command line when you start the terminal monitor, e.g.:

```
mysql -h mysql-server -u bookaplace -p bookaplace
```

- Below, we will explore how to interact with our database. Note that a full explanation of how to effectively write MySQL queries is beyond the scope of this course, so the discussion here will be limited.

Adding data to a database

- Let's start to put some data into our database. In MySQL, all data is stored in tables, so our first step will be to create one or more tables. To do this, we need to know the layout of the data that each table will hold, i.e. the identities, data types, and data characteristics of all of the table's columns.

- In this case, we will create a single table to store the lodgings for our Book-a-Place API. Here are the things we'll want to store in this table:
    - **ID** – A unique ID for the lodging.
    - **Name** – The lodging's name.
    - **Description** – A longer textual description of each lodging.
    - **Street address, City, State, and ZIP code** – We will store these as separate fields.
    - **Price** – The price per night for the lodging, in USD.
    - **Owner ID** – The ID of the user who owns the lodging.

- With that list of fields to be stored in the table, we'll start to write a `CREATE TABLE` statement to create the table. This statement will name the table and define each column. We'll write out the statement itself and then make a few

notes on its various parts below:

```
CREATE TABLE lodgings (
  id MEDIUMINT NOT NULL AUTO_INCREMENT,
  name VARCHAR(255) NOT NULL,
  description TEXT,
  street VARCHAR(255) NOT NULL,
  city VARCHAR(255) NOT NULL,
  state CHAR(2) NOT NULL,
  zip CHAR(5) NOT NULL,
  price DECIMAL(10,2) UNSIGNED NOT NULL,
  ownerid CHAR(24) NOT NULL,
  PRIMARY KEY (id),
  INDEX idx_ownerid (ownerid)
);
```

- Some notes on this statement:
    - The `id` column is our primary key for the table, since this will be the main way we access lodgings (e.g. via the `GET /lodgings/{lodgingID}` API endpoint). This could be any kind of data type, but it is convenient to store it as an integer and take advantage of MySQL's `AUTO_INCREMENT` attribute to ensure that each row is assigned a unique ID.
    - We put `NOT NULL` constraints on most columns, since they are required.
    - We store the `price` column as a `DECIMAL` type, which is an "exact" fixed-point number. Here, we use a number with 10 total digits, 2 of which come after the decimal point.
    - We represent the `ownerid` column as a 24-character string to make our lives easier when we implement a users database with MongoDB. We also build a secondary index over the `ownerid` column, since we will often want to query lodgings by owner ID.
        - Note if we were implementing a users table in the same MySQL database instead of in MongoDB (which we will do eventually), it would make sense to place a `FOREIGN KEY` constraint on the `ownerid` column instead of an `INDEX` constraint.

- With our table created, we are ready to begin inserting data into it. We can do this with the `INSERT INTO` statement. We must specify a value for each column

(leaving the value of `id` as NULL to allow the MySQL auto-increment feature to automatically assign a value), e.g.:

```
INSERT INTO lodgings VALUES (
  NULL,
  'My Cool Condo',
  'A nice place to stay, downtown near the riverfront.',
  '123 SW 1st St.',
  'Corvallis', 'OR', '97333',
  128.00,
  '1'
);
```

- We could make run more `INSERT INTO` statements if we wanted to insert more data. The `INSERT INTO` statement actually has a second form that uses a `SET` clause instead of a `VALUES` clause. In this version of `INSERT INTO`, we can specify column values as an unordered, comma-separated list of `name = value` pairs. For example, we could also insert a new lodging like this:

```
INSERT INTO lodgings SET
  id = NULL,
  name = 'My Marvelous Mansion',
  description = 'Big, luxurious, and comfy.',
  street = '7200 NW Grandview Dr.',
  city = 'Corvallis',
  state = 'OR',
  zip = '97330',
  price = 256.00,
  ownerid = '2';
```

Reading, updating and deleting data from a table

Reading data

- To read data from our `lodgings` table, we must use a `SELECT` statement. For example, the statement below reads all rows and columns from the table:

```
SELECT * FROM lodgings;
```

- We can also limit the rows we read to meet certain criteria by adding a `WHERE` clause to the `SELECT` statement. For example, this statement reads only the rows where the value of the `ownerid` column is `'1'`:

```
SELECT * FROM lodgings WHERE ownerid = '1';
```

- To perform pagination, we can add a `LIMIT` clause. It is good practice to accompany this with an `ORDER BY` clause to ensure consistent pagination. For example, this statement reads 10 rows starting with row 20:

```
SELECT * FROM lodgings ORDER BY id LIMIT 20,10;
```

Updating data

- To update data in our `lodgings` table, we use an `UPDATE` statement. The `UPDATE` statement is accompanied by a `WHERE` clause that determines *which* rows to apply the update to. For example, the following statement updates the price of all rows where the value of the `id` column is 2 (of which there is only one, since the value of the `id` column is unique):

```
UPDATE lodgings SET price = 300.00 WHERE id = 2;
```

- Importantly, an `UPDATE` statement can make multiple updates if the `WHERE` clause matches multiple rows. For example, this statement would update the price of *all* rows where `ownerid` is `'2'`:

```
UPDATE lodgings SET price = 200.00 WHERE ownerid = '2';
```

Deleting data

- The `DELETE FROM` statement deletes rows from a table. It works similarly to an `UPDATE` statement, using a `WHERE` clause to determine which rows to delete. For example, this statement deletes the row with `id` 2:

```
DELETE FROM lodgings WHERE id = 2;
```

- Again, as with the `UPDATE` statement, a `DELETE FROM` statement can delete multiple rows, depending on what the `WHERE` clause matches.

JavaScript promises

- Before moving on to incorporate MySQL into our API server, let's take a brief detour to explore JavaScript ***promises***.

- A promise is an object representing the eventual completion or failure of an asynchronous operation. They are used to allow an asynchronous operation (like a MySQL query) to return a value (a promise object) to which callbacks can be attached instead of requiring callbacks to be passed to asynchronous operations, which is the traditional paradigm for obtaining asynchronous results.

- The way promises are implemented allows for cleaner, much more understandable code when dealing with asynchronous operations, especially ones that need to be executed in sequence. Using callbacks in such a situation can result in a code structure often affectionately called the "pyramid of doom."

- To explore this in more detail and to see how using promises can help, let's start to work with a small function that simply waits for a second using JS's built-in `setTimeout()` function and then prints a provided message, calling a callback function to handle the "results" of this asynchronous operation:

```
function waitAndPrint(message, callback) {
  setTimeout(() => {
    console.log(message);
    callback();
  }, 1000);
}
```

    - Note here that we're using ES6's arrow function syntax, which for now you can think of as simply a different way to specify an anonymous function.

- Creating a sequence of delayed messages gives us a small pyramid of doom:

```
waitAndPrint("3", () => {
  waitAndPrint("2", () => {
    waitAndPrint("1", () => {});
```

```
  });
});
```

- Following the logic here is not incredibly difficult, but it is also not straightforward. If becomes more challenging if we want to permit error handling. Say, for example, that we modify our `waitAndPrint()` function to pass an error message into the callback if a simulated error occurs:

```
function waitAndPrint(message, callback) {
  setTimeout(() => {
    if (message === "Error") {
      callback(`We didn't like this string: "${message}"`);
    } else {
      console.log(message);
      callback(null);
    }
  }, 1000);
}
```

- Incorporating error handling into our sequence of calls to `waitAndPrint()` makes our pyramid of doom a bit more disastrous:

```
waitAndPrint("3", (err1) => {
  if (err1) {
    console.log(`== Error: ${err1}`);
  } else {
    waitAndPrint("2", (err2) => {
      if (err2) {
        console.log(`== Error: ${err2}`);
      } else {
        waitAndPrint("1", (err3) => {
          if (err3) {
            console.log(`== Error: ${err3}`);
          }
        });
      }
    });
  }
});
```

- This code is much more difficult to understand. Let's see how promises can help.

- Promises are represented in JS using the `Promise` class. A `Promise` object can exist in one of three states:
    - **Pending** – the initial state of a promise
    - **Resolved** – the state of a promise representing a successful operation
    - **Rejected** – the state of a promise representing a failed operation

- A `Promise` object is used to wrap an asynchronous operation. To do this, the `Promise` constructor is passed an "executor" function that, when called, executes the asynchronous operation. The executor function takes two arguments: a function that resolves the promise and a function that rejects it.

- This looks something like this:

```
let p = new Promise((resolve, reject) => {
  doAsyncOperation((err, results) => {
    if (err) {
      reject(err);
    } else {
      resolve(results);
    }
  });
});
```

- Here, the asynchronous operation itself, `doAsyncOpoeration()`, takes a callback function as an argument to handle errors and successful results. In this case, we provide a callback that rejects the containing `Promise` on an error and resolves it on success.

- When the `Promise` object is constructed, the executor function we provide (`(resolve, reject) => {...}`) is called immediately by the `Promise` framework, which passes the arguments `resolve` and `reject`. This results in the asynchronous operation being invoked.

- Let's turn back to our `waitAndPrint()` situation. In this case, our underlying asynchronous operation is `setTimeout()`. We can promise-ify `waitAndPrint()` by wrapping `setTimeout()` in a `Promise` object, which we

can then return. This will allow us to avoid passing a callback into `waitAndPrint()`. Instead, as we will see we will provide result and error handling functionality when we call `waitAndPrint()`.

- Here's what the promise-ified `waitAndPrint()` function looks like:

```
function waitAndPrint(message) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (message === "Error") {
        reject(`We didn't like this string: "${message}"`);
      } else {
        console.log(message);
        resolve();
      }
    }, 1000);
  });
}
```

- How do we use this promise-ified function? The most common approach is to use the `Promise.then()` method to attach result handling and `Promise.catch()` to attach error handling, e.g.:

```
waitAndPrint("Patience is a virtue").then(() => {
  console.log("... but I don't have the time.");
}).catch((err) => {
  console.log(`== Error: ${err}`);
});
```

- Here, we pass a result-handling function to `.then()` and an error-handling function to `.catch()`. These functions are invoked, respectively, by the `resolve` and `reject` functions associated with the `Promise` in `waitAndPrint()`.

- Combined, these result-handling and error-handling functions are similar to a callback we might have passed to our original `waitAndPrint()` function (the non-promise-ified version). Indeed, the code above does the same thing as this code, using our original `waitAndPrint()` function:

```
waitAndPrint("Patience is a virtue", (err) => {
  if (err) {
    console.log(`== Error: ${err}`);
  } else {
    console.log("... but I don't have the time.");
  }
});
```

- Overall, we don't seem to have gained much here. It's nice to separate result- and error-handling functionality, and it's nice to return something from our asynchronous operation instead of passing it a callback, but our resulting code is much the same as before.

- Promises really shine when used in a sequence of asynchronous operations. In particular, if the result-handling function we pass to `.then()` also returns a promise, we can chain together calls to `.then()` instead of nesting callbacks.

- Here's the promise-ified version of the sequence of calls (plus an extra call) we made above to `waitAndPrint()`:

```
waitAndPrint("3")
  .then(() => { return waitAndPrint("2"); })
  .then(() => { return waitAndPrint("1"); })
  .then(() => { return waitAndPrint("...Blastoff!"); })
  .catch((err) => { console.log(`== Error: ${err}`); });
```

- Notice how much easier this is to understand (and to write) than our pyramid of doom above. This works because `waitAndPrint()` returns a `Promise`. Each successive call to `.then()` simply returns this `Promise`, for which the next call to `.then()` provides result handling.

- Importantly, the call to `.catch()` here provides error handling for every `Promise` in the chain. For example, the following two sequences of calls would both result in calls to the error-handling function passed to `.catch()`:

```
waitAndPrint("3")
  .then(() => { return waitAndPrint("2"); })
  .then(() => { return waitAndPrint("1"); })
  .then(() => { return waitAndPrint("Error"); })
  .catch((err) => { console.log(`== Error: ${err}`); });

waitAndPrint("Error")
  .then(() => { return waitAndPrint("2"); })
  .then(() => { return waitAndPrint("1"); })
  .then(() => { return waitAndPrint("...Blastoff!"); })
  .catch((err) => { console.log(`== Error: ${err}`); });
```

- Promises are the modern way to work with asynchronous operations. They are supported out of the box in recent versions of Node.js (since version 0.12) and are also supported in most modern browsers. Cutting-edge versions of JS include interesting extensions to the promise framework.
    - There is also a `promise` package available on NPM that provides extended promise functionality.

Using MySQL in our API server

- Now that we've seen some of the basics of MySQL and a bit about how to use promises, let's explore how to incorporate these into our API server. To do this, we'll start with the API server code we previously developed (note that this URL points to a specific commit):

  https://github.com/OSU-CS493-Sp18/mysql/tree/9d88b532ba8c2e964628438732422ea97da58bc2

- For now, we can leave our `mysql-server` Docker container running and continue to use the database there.

- The first thing we'll want to do is install the `mysql` package, which will allow us to work with a MySQL database within Node.js:

```
npm install --save mysql
```

- With that dependency installed, we'll incorporate it into our API server code in `server.js`:

```
const mysql = require('mysql');
```

- In order to get database information (host, database name and credentials) into our server process, we'll set environment variables containing this information. We'll read the values from the environment within our server in order to use them to establish a connection to the database:

```
const mysqlHost = process.env.MYSQL_HOST;
const mysqlPort = process.env.MYSQL_PORT || '3306';
const mysqlDB = process.env.MYSQL_DB;
const mysqlUser = process.env.MYSQL_USER;
const mysqlPassword = process.env.MYSQL_PASSWORD;
```

    - Note that the default MySQL port is 3306.

- The `mysql` package can be used to establish a single connection with the database. A better practice is to establish a pool of connections from which we can draw to make requests. We'll use our credentials, etc. to do this:

```
const maxMySQLConnections = 10;
const mysqlPool = mysql.createPool({
  connectionLimit: maxMySQLConnections,
  host: mysqlHost,
  port: mysqlPort,
  database: mysqlDB,
  user: mysqlUser,
  password: mysqlPassword
});
```

- Now, we can use that connection pool to make queries to our database. All queries can now be made using the `mysqlPool.query()` method. We'll see how to incorporate queries into our API endpoints below.

MySQL-ifying the `GET /lodgings` endpoint

- Let's start seeing how to incorporate MySQL queries into our API server's endpoints. We'll start with the `GET /lodgings` endpoint, which returns a paginated list of lodgings from our database.
- This endpoint is an interesting one to start on because of the pagination, which requires us to make 2 sequential queries to the database: one to get the total number of lodgings and one to get the current page of lodgings.

- Let's start here by writing a query to count the number of lodgings. We'll write a separate function to do this. Note that our MySQL queries are going to be asynchronous calls, so we'll wrap them in `Promise` objects, which we'll return from their respective functions:

```
function getLodgingsCount() {
  return new Promise((resolve, reject) => {...});
}
```

- The specific query we'll want to execute to count the lodgings will look something like this:

```
SELECT COUNT(*) FROM lodgings;
```

- To run this query in our server, we'll make a query from our connection pool using the `mysqlPool.query()` method. We can pass different configurations of arguments to this method. We'll use a couple of these configurations as we go on, but all of them take two essential arguments:
    - `queryString` – a string containing a MySQL query to be executed
    - `callback` – a callback function to be called when the query is completed. This function can receive up to three different arguments:
        - `err` – an object representing an error, if one occurred making the query.
        - `results` – the results of the query. In the case of a `SELECT` query, this will be an array of objects, each of which represents one row of results.
        - `fields` – an array containing information about the fields returned by the query. This argument is often not used.

- Because of the way results are returned by this method, we will actually assign a name to our row count in the query, which will be executed like this:

```
mysqlPool.query(
  'SELECT COUNT(*) AS count FROM lodgings',
  function (err, results) {
    ...
  }
);
```

- If our query is successful here, `results` will look like this, where the name `count` comes from the alias we assigned to `COUNT(*)` in the query above:

```
[
  {
    count: ...
  }
]
```

- If there was no error, we can extract `count` and pass it to the promise's `resolve()` function. If we got an error, we can pass that to `reject()` instead:

```
if (err) {
  reject(err);
} else {
  resolve(results[0].count);
}
```

- We can now begin modifying the route for our GET /lodgings endpoint by calling `getLodgingsCount()` and hooking up results handling and error handling:

```
getLodgingsCount()
  .then((count) => {...})
  .catch((err) => {...});
```

- Let's first write the error-handling method passed to `.catch()`. Here, we'll respond with a 500 status, since the client's request was valid, and we'd normally expect to be able to succeed at making the count query to the database:

```
res.status(500).json({
  error: "Error fetching lodgings list.  Try again later."
});
```

- Now let's move to the `.then()` call. Here, we'll have the count passed into `resolve()` from `getLodgingsCount()`. Once we have that count, we'll want to perform pagination calculations and fetch the requested page of lodgings. Let's write a separate function to do this. We'll again return a `Promise` object:

```
function getLodgingsPage(page, totalCount) {
  return new Promise((resolve, reject) => {...});
}
```

- Within the promise's executor function, we'll perform the pagination calculations and make our second query to fetch the requested page of lodgings. Much of the code we wrote to perform our original pagination calculations can be re-used:

```
const numPerPage = 10;
const lastPage = Math.ceil(totalCount / numPerPage);
page = page < 1 ? 1 : page;
page = page > lastPage ? lastPage : page;
const offset = (page - 1) * numPerPage;
```

- The query we'll want to make to fetch a page of lodgings will look like the query we wrote above for paginated lists of lodgings:

```
SELECT * FROM lodgings ORDER BY id LIMIT <offset>,<count>
```

- Here, we'll want to plug the values of `offset` and `numPerPage` into the query string at `<offset>` and `<count>`, respectively. We could do this with a simple string concatenation, but the `mysql` package has a better mechanism for plugging values from variables into a query string.

- In particular, when formulating a query to `mysqlPool.query()`, we can insert a question mark `?` as a placeholder at every location into which we want to plug an outside value. Then, we can pass the values to be plugged into the placeholders

within an array as the second argument to `mysqlPool.query()` (pushing `callback` to the third argument).

- For example, we can execute our paginated query like this:

```
mysqlPool.query(
  'SELECT * FROM lodgings ORDER BY id LIMIT ?,?',
  [offset, numPerPage],
  function (err, results) {
    ...
  }
);
```

- Here, the value of `offset` is plugged into the first `?`, and the value of `numPerPage` is plugged into the second `?`.

- Importantly, any values plugged into placeholders this way are ***escaped***. This means that if those values contain text that could be interpreted as MySQL code, that text is translated into a form that is not executed as MySQL code.

- Any value that could have potentially been provided by the user *must* be escaped to prevent a type of attack known as ***SQL injection***.

- In this case, the values `offset` and `numPerPage` don't come from the user and so do not need to be escaped, but it's convenient to use the placeholder syntax whenever inserting values from variables into a query string, as we've done here.

- Within the callback to our query, we can pass an error, if there was one, to `reject()`. If there was no error, we will pass an object containing all of our pagination information along with the page of lodgings into `resolve()`, since we will want to send all of this back to the client in our response, and only one parameter may be passed to `resolve()`:

```
if (err) {
  reject(err);
} else {
  resolve({
```

```
      lodgings: results,
      pageNumber: page,
      totalPages: lastPage,
      pageSize: numPerPage,
      totalCount: totalCount
    });
}
```

- We can wrap up our `GET /lodgings` endpoint's route by calling `getLodgingsPage()` inside the `.then()` attached to our original `getLodgingsCount()` call, making sure to return the `Promise` it returns and passing in the requested page from the query string (or 1 if a valid page wasn't specified) and the total lodgings count from our call to `getLodgingsCount()`:

```
return getLodgingsPage(
     parseInt(req.query.page) || 1, count
);
```

- Because we return the `Promise` here, error handling for our call to `getLodgingsPage()` will be handled by the `.catch()` we already attached to the call to `getLodgingsCount()`. We can add an additional `.then()` to this promise chain to return the success response to the client:

```
.then((lodgingsPageInfo) => {
  res.status(200).json(lodgingsPageInfo);
});
```

- Importantly, note that the array of results returned by `mysqlPool.query()` actually contains objects of the `RowDataPacket` class. However, these objects are correctly serialized to JSON by `res.json()`, so we don't need to do more with them.

- If our `lodgings` table contained more columns than we wanted to return in the API's response to this endpoint, we could either iterate through the results and extract only the fields we wanted to return, or we could formulate our `SELECT` statement to return only the columns we wanted to return from the API.

MySQL-ifying the `POST /lodgings` endpoint

- Next, let's work hooking our API's `POST /lodgings` endpoint up to our MySQL database. This endpoint creates a new lodging resource and stores it in the database.

- Much of the code we wrote before for this method can stay the same. In particular, we will still want to perform data verification on the POST request body and respond with a 400 error if the request contains a badly-formatted body:

```
if (req.body && req.body.name && req.body.price && ...) {
  ...
} else {
  res.status(400).json({
    error: "Request needs a JSON body with a name, etc."
  });
}
```

- However, once we've ensured that the request body is correctly formatted, we'll want to execute a MySQL query to insert the corresponding lodging into the database. We'll write a function to do this. As before, it will return a `Promise`:

```
function insertNewLodging(lodging) {
  return new Promise((resolve, reject) => {...});
}
```

- Within the promise's executor function, we'll extract the lodging values needed to populate a row in the database, adding in a `null` value for the `id` column:

```
const lodgingValues = {
  id: null,
  name: lodging.name,
  description: lodging.description,
  street: lodging.street,
  city: lodging.city,
  state: lodging.state,
  zip: lodging.zip,
  price: lodging.price,
```

```
  ownerid: lodging.ownerID
};
```

    - Importantly, note that the keys of this object correspond exactly to the names of the columns in the lodgings table.

- With these individual values extracted, we can execute an `INSERT INTO` query to insert the new lodging into the DB:

```
mysqlPool.query(
  'INSERT INTO lodgings SET ?',
  lodgingValues,
  function (err, result) {
    ...
  }
);
```

- In this query, we're combining the special `INSERT INTO table SET ...` syntax of MySQL we saw above with a feature of the `mysql` package's placeholder framework.

- Specifically, if we put placeholder values into a JS object instead of an array, the `mysql` package encodes the object as a list of `key = value` pairs like above.

- In other words, because the `lodgingValues` object's keys directly correspond to the column names of the lodgings table, we can pass that object directly to supply the placeholder values for our query, and the `mysql` package will put it into the proper form to use in the `SET` clause of our `INSERT INTO` statement.

- In the callback to this query, we'll do something similar to what we did in our MySQL query functions above, calling the promise's `reject()` with the `err` object if there was an error making the query, or, if there was no error, using the `insertID` field of the `result` object to pass the ID of the newly-inserted object into `resolve()`:

```
if (err) {
  reject(err);
```

```
} else {
  resolve(results.insertId);
}
```

- With this function written, we can go back to our `POST /lodgings` endpoint's route. There, after we've verified that the request body is correctly formatted, we can make a call to the `insertNewLodging()` function:

```
insertNewLodging(req.body)
  .then((id) => {...})
  .catch((err) => {...});
```

- Inside the `.then()` call here, we know the lodging was successfully inserted into the database, and we can return its ID back to the client:

```
res.status(201).json({ id: id });
```

- Inside the `.catch()` call, we can again respond with a 500 error, since any error in this situation is a server-side error:

```
res.status(500).json({
  error: "Error inserting lodging into DB."
});
```

MySQL-ifying the `GET /lodgings/{lodgingID}` endpoint

- To power our `GET /lodgings/{lodgingID}` endpoint, let's begin a function to perform a database query that selects a lodging based on its ID, returning a `Promise`, as before:

```
function getLodgingByID(lodgingID, callback) {
  return new Promise((resolve, reject) => {...});
}
```

- Inside the promise's executor function, we'll again execute a `SELECT` statement, this time with a `WHERE` clause to select only the lodging with the specified ID. If we got an error, we'll pass that to the promise's `reject()` function. Otherwise, we'll pass the first result (which could be `undefined` if the query didn't match any lodgings) to `resolve()`:

```
mysqlPool.query(
  'SELECT * FROM lodgings WHERE id = ?',
  [ lodgingID ],
  function (err, results) {
    if (err) {
      reject(null);
    } else {
      resolve(results[0]);
    }
  }
);
```

- Back in our `GET /lodgings/{lodgingID}` endpoint's route, we'll call this function with the ID specified, obtained from the route parameters:

```
const lodgingID = parseInt(req.params.lodgingID);
getLodgingByID(lodgingID)
  .then((lodging) => {...})
  .catch((err) => {...});
```

- Inside the `.then()` call here, we need to deal with two different situations: when the query successfully returned a lodging and when the query did not return a lodging because the specified ID did not match anything in the database. In the former case, we'll return a success response, and in the latter, we'll call `next()` to eventually respond with a 404 error:

```
if (lodging) {
  res.status(200).json(lodging);
} else {
  next();
}
```

- Inside the `.catch()` call, we'll again return a 500 status response to the client:

```
res.status(500).json({
  error: "Unable to fetch lodging."
});
```

MySQL-ifying the `PUT /lodgings/{lodgingID}` endpoint

- Our `PUT /lodgings/{lodgingID}` endpoint will combine aspects of the `POST /lodgings` endpoint and the `GET /lodgings/{lodgingID}` endpoint. The pattern we're using to implement these endpoints should be familiar by now, so below is the full code for an `updateLodgingByID()` function to use for this endpoint, returning a `Promise`, as always. This function makes an `UPDATE` query to the database to update a lodging based on its ID:

```
function updateLodgingByID(lodgingID, lodging) {
  return new Promise((resolve, reject) => {
    const lodgingValues = {
      name: lodging.name,
      description: lodging.description,
      street: lodging.street,
      city: lodging.city,
      state: lodging.state,
      zip: lodging.zip,
      price: lodging.price,
      ownerid: lodging.ownerID
    };
    mysqlPool.query(
      'UPDATE lodgings SET ? WHERE id = ?',
      [ lodgingValues, lodgingID ],
      function (err, result) {
        if (err) {
          reject(err);
        } else {
          resolve(result.affectedRows > 0);
        }
      }
    );
  });
```

```
}
```

- This function uses concepts we've seen above. Note here that when calling `resolve()` to handle the success case, we simply pass it a boolean value indicating whether any rows of the table were affected by the query. If no rows were affected, that means that the specified lodging ID was invalid.

- Our API endpoint's route here will again look like a combination of the routes of endpoints we wrote above. Specifically, it must verify the PUT request body, call our query function, and send an appropriate response back to the client based on the results of the query. Here is the code:

```
const lodgingID = parseInt(req.params.lodgingID);
if (req.body && req.body.name && req.body.price && ...) {
  updateLodgingByID(lodgingID, req.body)
    .then((updateSuccessful) => {
      if (updated) {
        res.status(200).json({});
      } else {
        next();
      }
    })
    .catch((err) => {
      res.status(500).json({
        error: "Unable to update lodging."
      });
    });
} else {
  res.status(400).json({
    err: "Request needs a JSON body with a name, etc."
  });
}
```

MySQL-ifying the `DELETE /lodgings/{lodgingID}` endpoint

- The `DELETE /lodgings/{lodgingID}` endpoint will be very similar to the `GET /lodgings/{lodgingID}` endpoint but will need to execute a DELETE query instead of a SELECT query. Below is a function to perform this query that based on a given lodging ID, returning a `Promise`, as always. As above, we use the `affectedRows` property of the query result to determine whether the specified lodging ID was valid and resulted in a lodging being deleted:

```
function deleteLodgingByID(lodgingID, callback) {
  return new Promise((resolve, reject) => {
    mysqlPool.query(
      'DELETE FROM lodgings WHERE id = ?',
      [ lodgingID ],
      function (err, result) {
        if (err) {
          reject(err);
        } else {
          resolve(result.affectedRows > 0);
        }
      }
    );
  });
}
```

- Given this function, the code for our API endpoint's route will look similar to the one for the corresponding GET route:

```
const lodgingID = parseInt(req.params.lodgingID);
```

```
deleteLodgingByID(lodgingID)
  .then((deleteSuccessful) => {
    if (deleted) {
      res.status(204).end();
    } else {
      next();
    }
  })
  .catch((err) => {
    res.status(500).json({
      error: "Unable to delete lodging."
    });
  });
```

MySQL-ifying the `GET /users/{userID}/lodgings` endpoint

- Given the structure of our database, the `GET /users/{userID}/lodgings` endpoint, which is used to fetch all of the lodgings owned by a specific user, will look nearly identical to the `GET /lodgings/{lodgingID}` endpoint.
- In this case, however, we will need to execute a query like the following and return *all* of the results:

```
SELECT * FROM lodgings WHERE ownerid = <userID>
```

- Implementing code for this endpoint is left as an exercise for you.