Using MongoDB to Store API Data

- ***Document-oriented databases*** are becoming increasingly popular. Unlike relational databases, which store highly-structured data in tabular form, document-oriented databases store data as semi-structured objects called ***documents.***

- Like a row in a relational database, a document is a data record, typically representing a single entity of a given type, e.g. a user, a business, etc.

- Documents are typically composed of key-value pairs. The values of these pairs can be simple types, such as strings, numbers, dates, etc., but they can also be structured types such as arrays or even nested documents.

- Further, unlike relational databases, in which each data record (i.e. a row) must obey the schema of the table in which it resides, document-oriented databases are typically schemaless by default, and the documents in a given database may or may not have the same structure.

- And, unlike simple key-value stores, document-oriented databases make use of the internal structure of documents to help optimize queries over the database.

- MongoDB is among the most widely-used document-oriented databases today.

- In MongoDB, documents are stored as BSON, a binary representation of JSON. In fact, it is easiest to think of MongoDB as simply storing JSON documents. Indeed, when working with MongoDB, documents will be represented in the program as JS objects.

- For example, below is what a document containing data for a single person might look like in an API that dealt with photo data. Note that this document contains structured data, including an array of nested sub-documents. For all intents and purposes, this is simply a JS object:

```
{
     personId: "luke",
     name: "Luke Skywalker",
     age: 24,
     bio: "Jedi Knight",
     photos: [
          {
               url: "http://...",
               caption: "..."
          },
          {
               url: "http://...",
               caption: "..."
          },
          ...
     ]
}
```

Running a MongoDB server (with authentication)

- As with MySQL, MongoDB is a networked service, and we'll use the official MongoDB Docker image to run our MongoDB server.

- Note that MongoDB does not have access control enabled by default. However, if we set the environment variables `MONGO_INITDB_ROOT_USERNAME` and `MONGO_INITDB_ROOT_PASSWORD`, this will create a superuser for us and enable authentication before starting the MongoDB server:

```
docker run -d --name mongo-server           \
  --network mongo-net                       \
  -p "27017:27017"                          \
  -e "MONGO_INITDB_ROOT_USERNAME=root"           \
  -e "MONGO_INITDB_ROOT_PASSWORD=hunter2"   \
  mongo:latest
```

- A few additional notes here:
    - We're also assuming a Docker network named `mongo-net` is created.
    - Port 27017 is the default MongoDB port.

    - If we had any scripts (JavaScript files) we wanted to run to perform initialization on our database (e.g. user creation, etc.), we could store those in a directory on our host machine and bind mount that directory as `/docker-entrypoint-initdb.d` in the container to automatically run those scripts.

Using MongoDB shell to connect to our server

- MongoDB has a command line shell similar to the MySQL terminal monitor in which we can run basic JavaScript commands against our server. As with MySQL, we can run the MongoDB shell using the MongoDB Docker image:

```
docker run --rm -it                 \
     --network mongo-net            \
     mongo:latest                   \
     mongo --host mongo-server      \
          --username root           \
          --password hunter2        \
          --authenticationDatabase admin
```

- Here, the `mongo` command launches the MongoDB shell, and we specify the hostname and user credentials for our server. We also need to set the database against which our user is authenticated. When creating a user at server startup time as we did above, this database will be `admin`.

- Running this task will launch us into the MongoDB shell. Our first task will be to use this shell to create a database and less-privileged user we can use for our Book-a-Place API. First, we'll create the database, named `bookaplace`:

```
use bookaplace
```

- Here, the `use` command sets the active database to `bookaplace`, creating the database if it doesn't already exist. As we'll see, many MongoDB commands do something similar.

- Now, we can execute a method on the `bookaplace` database to create a user. All interactions with a database in the MongoDB shell happen through the global `db` object. In this case, we'll call `createUser()` on this object, passing as an argument a document specifying information about the user we're creating:

```
db.createUser({
  user: "bookaplace",
  pwd: "hunter2",
  roles: [ { role: "readWrite", db: "bookaplace" } ]
})
```

- Note that the document we pass here is essentially just a JS object. It lets Mongo know that we want our user to be named `bookaplace`, that their password should be `hunter2`, and that we want them to have read and write permissions on the `bookaplace` database. We'll use this user later when we connect to our MongoDB server from our API server.

- Importantly, the ***authentication database*** for this user is the one that was active at the time we called `createUser()`. In this case, this was `bookaplace`.

CRUD operations in MongoDB

- In MongoDB, documents are stored in ***collections***, where each document in the collection is typically an instance of the same kind of entity. A MongoDB database can contain many collections, each of which represents a different kind of entity.

- For example, in our Book-a-Place application, we might want a collection to store documents with information about Book-a-Place users. We could name this collection `users`. If we did this, we could get access to the collection in the MongoDB shell via the `db` object as `db.people`.

- To obtain an array listing the names of all collections in our database by calling the `db` object's `getCollectionNames()` method:

```
db.getCollectionNames();
```

Creating data with `insertOne()` and `insertMany()`

- To create data entries in a MongoDB database, we use either the `insertOne()` method or the `insertMany()` method on the specific collection in which we want to create the entries.

- For example, we could insert one new entry into our `users` collection like this:

```
db.users.insertOne({
  userID: "luke",
  name: "Luke Skywalker",
  age: 24,
  email: "LukeTheJedi@gmail.com",
  lodgings: []
})
```

- Note that if we try to insert into a collection that doesn't exist, the collection is automatically created before the data inserted. Thus, the `insertOne()` call above would succeed, even if the `users` collection didn't exist yet.

- Importantly, every document in a MongoDB collection must contain a field `_id` whose value uniquely identifies the document within the collection. If we don't specify the `_id` field when calling `insertOne()`, MongoDB automatically generates an `ObjectId` object and assigns it as the value of the document's `_id` field. It is typically easiest to allow MongoDB to automatically assign IDs this way, similar to the way we make MySQL auto-increment a primary key.

- For example, if we used the `find()` method (which we'll learn more about in a bit) to read the data in our `users` collection, we'd see that our newly inserted document was assigned an `_id` field:

```
{
     "_id": ObjectId("5a1f3e8864be89eef867596a"),
     "userID": "luke",
     ...
}
```

- An array can be passed to the `insertMany()` method to insert multiple documents at once, e.g.:

```
db.users.insertMany([
     { userID: "leia", ... },
     { userID: "darth", ... },
     { userID: "rey", ... }
])
```

- Each call to `insertOne()` or to `insertMany()` returns a `WriteResult` object that contains information about the status of the operation, such as the number of documents successfully inserted, e.g.:

```
WriteResult({ "nInserted" : 1 })
```

Reading data using `find()`

- To read data from a MongoDB database, we use the `find()` method on the specific collection from which we want to read.

- To return the complete set of all documents in a collection, we can simply call `find()` with no arguments, e.g.:

```
db.users.find()
```

- This would result in all of the documents in our `users` collection being printed in the MongoDB shell. To print them in a more readable format, we could use MongoDB's pretty-printing functionality:

```
db.users.find().pretty()
```

- Often, however, we will want to perform more precise queries that return only a subset of a collection's documents that match certain criteria. To do this, we can pass an argument to the `find()` method to specify our query.

- One of the most common kinds of query is one based on an equality match on one or more fields. To perform such a query can pass to `find()` an object specifying the equality conditions by which we want to query.

- For example, if we wanted to query our people collection to find all documents whose `userID` field has a value equal to `"luke"`, we could do so like this:

```
db.users.find({ userID: "luke" })
```

- Of course, if we ensure that the `userID` field has a unique value for each document, then the above call to `find()` will return only a single document (or zero documents, if none have a `userID` field equal to `"luke"`).

- We can specify multiple fields in such a query, and only documents matching all of the specified criteria will be returned. For example, the following query returns only documents whose `age` field equals `24` *and* whose `email` field equals `"TheLastJedi@gmail.com"`:

```
db.users.find({
  age: 22,
  email: "TheLastJedi@gmail.com"
})
```

- We can also query nested documents using dot notation. For example, if the documents in our `users` collection each had a field called `address` whose value was a nested sub-document specifying the user's address, we could make a query like the following to find only users whose zip code is 97330:

```
db.users.find({ "address.zipcode": "97330" })
```

    - Importantly, when using dot notation like this, the name of the field needs to be enclosed in quotes, as above.

- MongoDB allows us to make more flexible queries than ones that test only for equality. Specifically, the object we pass as an argument to `find()` can use operators, such as comparisons, to specify various query conditions.

- The typical (with some exceptions) format of a query operator in MongoDB looks like this:

```
{ <field>: { <operator>: <value> } }
```

- For example, if we wanted to make a query over our `users` collection to find all documents representing users whose age was at least 30, we could use the `$gte` (greater than or equal) operator:

```
db.users.find({ age: { $gte: 30 } });
```

- Or, for example, you can specify a logical OR of query conditions using the `$or` operator. The `$or` operator works a little differently than other operators. Specifically, the value passed to the `$or` operator is an *array* of the query conditions to be OR'ed together. For example, the following query would return all documents representing people who are at least 30 *or* whose email is `"TheLastJedi@gmail.com"`:

```
db.users.find({ $or: [
     { age: { $gte: 30 } },
     { email: "TheLastJedi@gmail.com" }
]});
```

- There are many more query operators you can use. You can find a list of them with links to documentation here:

  https://docs.mongodb.com/manual/reference/operator/query/

- Finally, we can sort the results of a call to `find()` using the `sort()` method, which takes as an argument an object specifying the fields by which to sort the returned documents and whether to sort in ascending (using the value `1`) or descending (using the value `-1`) order on each of those fields.

- For example, if we wanted to sort all of the documents in our `users` collection by ascending age, we could do so like this:

```
db.users.find().sort({ age: 1 });
```

- Alternatively, if we wanted to sort all of the documents in our `users` collection by *descending* age, we could do so like this:

```
db.users.find().sort({ age: -1 });
```

- Multiple fields can be specified in the argument to `sort()`. The result will be that documents will first be sorted based on the first specified field. Then documents with equal values of that field will be sorted based on the second specified field, and so forth.

Updating data with `updateOne()`, `updateMany()`, and `replaceOne()`

- We can update data in a MongoDB collection using the `updateOne()` method, which updates at most a single document, the `updateMany()` method, which could potentially update multiple documents, and the `replaceOne()` method, which completely replaces a document with a new value.

- The first two of these methods, `updateOne()` and `updateMany()`, are called the same way. Specifically, they both take two arguments:
    - An object specifying a query. This uses the same structure and syntax as the query object argument passed to `find()`.
    - An object specifying how to update the document(s) that match the query specified as the first argument.

- When two such arguments are passed to the `updateOne()` method, the update specified in the update object is applied to the first document in the collection matching the query object, if any. The `updateMany()` method, on the other hand, applies the specified update to *all* documents matching the query object.

- Many different kinds of update can be performed with `updateOne()` and `updateMany()`. MongoDB provides many update operators to specify these updates.

- One of the simplest update operators is `$set`, which simply replaces the value of a given field with a new value (or creates that field if it doesn't exist, assigning it the specified value). For example, the following call to `updateOne()` will update the first document whose `userID` equals `"luke"` by setting it's `email` field to `"TheReclusiveJedi@gmail.com"`:

```
db.users.updateOne(
     { userID: "luke" },
     { $set: { email: "TheReclusiveJedi@gmail.com" } }
);
```

- Another useful update operator is `$inc`, which increments the value of one or more fields. For example, the following call to `updateMany()` will update all of the documents in our `users` collection, incrementing each person's age by 1:

```
db.users.updateMany(
      {},
      { $inc: { age: 1 } }
);
```

- An element can be added to an array within a document using the `$push` operator. For example, the following call to `updateOne()` will add one lodging ID to the end of the `lodgings` array of the first document (if any) whose `userID` field is `"darth"`:

```
db.users.updateOne(
      { userID: "darth" },
      { $push: { lodgings: 7 } }
);
```

- Again, there are many more update operators available in MongoDB. You can find a list of them with links to documentation here:

  https://docs.mongodb.com/manual/reference/operator/update/

- The `replaceOne()` method does not take an update object as its second argument but instead takes a replacement document. If the query argument passed as the first argument to `replaceOne()` matches any documents in the database, the first such document is completely replaced with the replacement document passed as the second argument.
  - Importantly, if a document is replaced using `replaceOne()`, its `_id` field isn't changed. This field can't be changed once set for a given document.

Deleting data with `deleteOne()` and `deleteMany()`

- Finally, documents can be deleted from a collection in MongoDB using the `deleteOne()` and `deleteMany()` methods.

- These methods both take as an argument an object specifying a query with the same structure and syntax as the query objects passed to `find()` and `updateOne()`/`updateMany()` methods. `deleteOne()` deletes the first documents in the collection matching the specified query, and `deleteMany()` deletes all such matching documents.

- For example, this call to `deleteOne()` deletes the first document in our `users` collection whose `userID` field is `"darth"`:

```
db.people.userID({ personId: "darth" });
```

Indexing a collection

- As with a MySQL table, a MongoDB collection can be indexed on specific document fields to allow for faster queries on those fields.

- The simplest way to create an index is to call the `createIndex()` method on the collection to be indexed. The first argument to this method is an object specifying the field on which to index and what type of index (ascending sort or descending sort) to create.

- For example, the following call creates an ascending index on our `users` collection over the `userID` field:

```
db.users.createIndex({ userID: 1 })
```

- For fields for which we know each document will have a unique value, we can create a unique index, which enforces uniqueness for the indexed fields. To do this, we can pass an option in the second argument to `createIndex()`:

```
db.users.createIndex({ userID: 1 }, { unique: true })
```

- By default, every collection has a unique index on the `_id` field.

- There are many different ways to index a collection in MongoDB. The documentation goes into more detail about this:

  https://docs.mongodb.com/manual/indexes/

Accessing a MongoDB database from a Node.js app

- Now that we have an idea how to use a MongoDB database to store and access data, we want to be able to incorporate data from a MongoDB database into our API server.

- We can do this easily using the MongoDB Node.js driver, which we can install using npm:

  ```
  npm install --save mongodb
  ```

- Once we have the MongoDB driver installed, we can import it into our server-side code. Importantly, the MongoDB driver contains various classes we can use in our code. Here, we are specifically interested in the `MongoClient` class, which will allow us to make queries to an existing MongoDB database. Thus, we'll grab that class directly when we import the driver:

  ```
  const MongoClient = require('mongodb').MongoClient;
  ```

- We can use the `MongoClient` class to connect to our database. To do this, we'll need a URL that points to the database. This URL must have the following form:

  ```
  mongodb://<username>:<password>@<hostName>:<port>/<database>
  ```

- Most parts of this URL should be straightforward in meaning. Note that the `<database>` part of the URL indicates the *authentication database* for the specified user.

- We'll need specific values for the username, password, hostname, etc. in order to construct such a URL. It is best practice to read such values in from the environment. Doing so helps us avoid hard-coding these values into our code (and thus potentially saving them in a public version control repository like GitHub). It also gives us flexibility to easily change these values, e.g. switching between a development database and a production database.

- In order to read these values from the environment, we can do something like this (27017 is the default port on which a MongoDB server runs):

  ```
  const mongoHost = process.env.MONGO_HOST;
  const mongoPort = process.env.MONGO_PORT || 27017;
  const mongoUser = process.env.MONGO_USER;
  const mongoPassword = process.env.MONGO_PASSWORD;
  ```

```
const mongoDBName = process.env.MONGO_DB;
```

- Assuming these values are set appropriately in the environment when we start our server, we can use them like this to generate our Mongo URL:

```
const mongoURL =
`mongodb://${mongoUser}:${mongoPassword}@${mongoHost}:${mon
goPort}/${mongoDBName}`;
```

- Once we have our URL, we can use the `MongoClient.connect()` method to connect to our database. There are some considerations we must make here:
    - If any routes or other middleware in our server depend on our MongoDB database, we want to ensure that our database connection was successful before allowing those routes or middleware functions to be called.
    - `MongoClient.connect()` is an asynchronous method. It takes as an argument a callback function, and a reference to the connected client is passed as an argument to the callback after the connection is successfully made. This can be used to get a reference to a database.

- Given these considerations, the typical approach to making a database connection with `MongoClient.connect()` is to define a global variable in which we'll store the database reference passed to the callback of `MongoClient.connect()`. Because this variable is at global scope, we'll be able to access it within our routes and other middleware functions. In addition, we'll start our own web application's server from *within* the callback to `MongoClient.connect()` to ensure that the database connection is successfully made before our server starts:

```
let mongoDB = null;
...
// Routes and middleware are defined here.
...
MongoClient.connect(mongoURL, function (err, client) {
  if (err) {
    throw err;
  }
  mongoDB = client.db(mongoDBName);
  app.listen(port, function () {
    console.log("== Server listening on port", port);
```

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

- Now, our server will only start if a successful connection is made to our MongoDB database. Thus, we can feel safe to incorporate calls to our database into our routes and middleware functions.

- Importantly, the connection created with `MongoClient.connect()` has some nice features, like automatic reconnection upon an error. Because of this, it is recommended that an application call `MongoClient.connect()` only once and then reuse the db object that's provided by the connected client.

Inserting data into MongoDB within Node.js

- Now that we have a connection to our database created, let's start using it by creating an API endpoint to insert a new user into our `users` collection. We'll do the insertion within a new function that takes a valid object representing the user to be inserted. The first thing we'll do here is extract the valid fields from this user object (adding an empty array representing the user's lodgings owned):

```
function insertNewUser(user) {
  const userValues = {
    userID: user.userID,
    name: user.name,
    email: user.email,
    lodgings: []
  };
  ...
}
```

- To perform an operation on a MongoDB collection from using the Node.js `mongodb` driver, we first need to use our global database object to grab a reference to that collection:

```
const usersCollection = mongoDB.collection('users');
```

- We can now use this reference to our `users` collection to insert an element using the `insertOne()` method.

- Importantly, the Node.js versions of all of the MongoDB CRUD methods we examined above already return a `Promise` object.

- In this case, however, we want to keep all of our MongoDB-related logic for new user insertion contained within this single function, so we'll actually attach a `.then()` clause to this promise that we'll use to extract the `_id` field of the newly-inserted user, which we'll return to the caller wrapped in a second promise:

```
return usersCollection.insertOne(userValues)
  .then((result) => {
    return Promise.resolve(result.insertedId);
  });
```

- Here, the function `Promise.resolve()` simply converts a value into a `Promise` object. The following two statements are essentially equivalent:

```
let p = Promise.resolve(value);
let p = new Promise((resolve) => { resolve(value); });
```

- We'll postpone any error handling for our query until later.

- With that function written, we can begin writing our API endpoint for inserting a new user. Here's what the endpoint's route definition will look like:

```
app.post('/users', function (req, res) {
  ...
});
```

- Within this route, we'll assume we have a `validateUserObject()` function available to validate the request body the client sends to this endpoint. If the client sends an invalid body, we'll send back a 400 response:

```
if (validateUserObject(req.body)) {
  ...
} else {
  res.status(400).json({
    error: "Request body does not contain valid user data."
  });
```

```
}
```

- If the request body is valid, we can call the function we just wrote to insert a new user, attaching `.then()` and `.catch()` clauses:
```
insertNewUser(req.body)
  .then((id) => {...})
  .catch((err) => {...});
```

- If the insertion was successful (in the `.then()` clause), we'll return a success response to the client:

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

- Otherwise, in the `.catch()` clause, we'll return a server error:

```
res.status(500).json({
  error: "Error creating new user."
});
```

Fetching data from MongoDB within Node.js

- Next, let's work on implementing an API endpoint to fetch a user's data based on ID. We'll implement this endpoint so that users can be queried by either the `userID` field or the `_id` field. We'll start as always by writing a function to perform our query:

```
function getUserByID(id) {
  ...
}
```

- Within this function, as before, we'll use our global database object to grab a reference to our `users` collection:

```
const usersCollection = mongoDB.collection('users');
```

- Next, we'll figure out whether we want to make our query on the `userID` field or the `_id` field. We'll assume if the value of `id` is a valid MongoDB `ObjectID` that we should query based on the `_id` field. Otherwise, we'll query based on `userID`. To check whether a value is a valid `ObjectID`, we'll first need to

import the `ObjectID` class from the `mongodb` package:

```
const ObjectID = require('mongodb').ObjectID;
```

- Now, we can use the static method `ObjectID.isValid()` to check the value passed in `id`. We'll write a new function to generate the correct query according to that value:

```
function generateUserIDQuery(userID) {
  if (ObjectID.isValid(userID)) {
    return { '_id': new ObjectID(userID) };
  } else {
    return { 'userID': userID };
  }
}
```

- We can now generate our query using this new function and execute it using the MongoDB `find()` method. Note that this method returns a cursor object, which we'll convert to an array by calling its `toArray()` method. As before, we'll attach a `.then()` clause to the promise returned by `toArray()` within which we'll extract the value of the first element in the array (which will be undefined if the requested user doesn't exist):

```
const query = generateUserIDQuery(userID);
return usersCollection.find(query).toArray()
  .then((results) => {
    return Promise.resolve(results[0]);
  });
```

- Now we're ready to create our API endpoint for fetching a user by their ID. Here's what the route definition will look like:

```
app.get('/users/:userID', function (req, res, next) {
  ...
});
```

- Within this route, we'll simply initiate a call to the `getUserByID()` function we just wrote to try to fetch the user specified in the requested URL path:

```
getUserByID(req.params.userID)
  .then((user) => {...})
  .catch((err) => {...});
```

- Inside the `.then()` clause here, we need to handle two different cases: if a user corresponding to the requested ID was successfully fetched and if one was not successfully fetched. In the former case, we want to send the user data back in a success response. In the latter, we'll return a 404 error (using `next()`):

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

- Inside the `.catch()` clause, we'll simply respond with a 500 error, as always:

```
res.status(500).json({
  error: "Error fetching user."
});
```

Updating data in MongoDB from Node.js (and coordinating with MySQL)

- Finally, let's update our application to add the ID of each newly-inserted lodging into the `lodgings` array for that user in our MongoDB `users` collection. This will be particularly interesting because it will involve coordination between MongoDB and MySQL.

- Let's start by writing a function that updates a user's entry in our `users` collection to push a lodging ID into the user's `lodgings` array. This function will begin much like our function to fetch a user, generating an object to query a user based on the specified user ID:

```
function addLodgingToUser(lodgingID, userID) {
  const usersCollection = mongoDB.collection('users');
  const query = generateUserIDQuery(userID);
  ...
```

```
}
```

- After we have our query object, we can make a call to `updateOne()` that uses the `$push` operator to push the specified lodging ID into the specified user's `lodgings` array. As before, we'll attach a `.then()` clause to the returned promise to return the *lodging* ID after a successful update (we'll see why later):

```
return usersCollection.updateOne(
  query,
  { $push: { lodgings: lodgingID } }
).then(() => { return Promise.resolve(lodgingID); });
```

- Given this function, we can now start to update the already-existing API endpoint in which we insert a new lodging into our database:

```
app.post('/lodgings', function (req, res, next) {
  ...
});
```

- Within this endpoint, our MySQL database and MongoDB database will interact. Specifically, now that we'll be inserting data into multiple places we'll have to modify our endpoint to perform the following sequence of actions if the request body represents a valid lodging object:
    - Make sure the user specified by the `ownerID` field of the request body represents a valid user. We don't want to insert a lodging if it isn't tied to a valid user.
        - If the request body does not specify a valid user, we'll return a response with a 400 error status.
    - If the specified user is valid, we'll insert the lodging into our MySQL database. This will generate an ID for the lodging.
    - In addition to inserting the lodging into our MySQL database, we'll call the `addLodgingToUser()` function we just wrote to add the newly-generated lodging ID to the `lodgings` array of the specified user within our MongoDB database.
    - If all of this succeeds, we'll return a success response to the client containing the ID of the newly-created lodging (as before).

- Our first step will be to verify that the request body specifies a valid user in `ownerID`. We can use our already-written `getUserByID()` function to do this. Specifically, if this function returns a non-null user, then we know the specified user exists. We'll make this call after we've validated the request body:

```
if (validateLodgingObject(req.body)) {
  getUserByID(req.body.ownerID)
    .then((user) => {...})
    .catch((err) => {...});
}
```

- Now, we want to call `insertNewLodging()` *only* if the call to `getUserByID()` returned a valid user. Thus, we'll move this call *inside* the `.then()` clause of our call to `getUserByID()`, wrapped within a check on the returned user.:

```
if (user) {
  return insertNewLodging(req.body);
} else {
  ...
}
```

- If the user is *not* valid, we'll return a 400 error to the client. However, we need to be careful how we do this. Specifically, we are returning a promise (returned by `insertNewLodging()`) from the if statement above, suggesting that we're going to attach another `.then()` clause into this promise chain.

- It's important to recognize that this additional `.then()` clause will be executed even if we return a response before it. If this second `.then()` clause also returns a response, this could result in an error, since you can't send two responses to the same request.

- The upshot here is that we'll want to bypass the second `.then()` clause if the specified user ID is invalid. We'll reserve this second `.then()` clause for returning a success response.

- To bypass the second `.then()` clause, we can simply use `Promise.reject()` (which is similar to `Promise.reject()`, which we saw above) to wrap an error value that we'll later handle in a `.catch()` clause. Here, we'll simply wrap the value 400, which will indicate that the `.catch()` clause should send a 400 error:

```
... else {
  return Promise.reject(400);
}
```

- Now, we can move on to implement our `.catch()` clause. Here, we'll have to differentiate between errors. Specifically, if we get an error value of 400 from our `.then()` clause above, this is an indication that we need to respond with a 400 error. Otherwise, we'll respond with a 500 error

```
if (err === 400) {
  res.status(400).json({
    error: `Invalid owner ID: (${req.body.ownerID}).`
  });
} else {
  res.status(500).json({
    error: "Error inserting lodging into DB."
  });
}
```

- Now, we must modify our `insertNewLodging()` function to do two things:
  - Insert the specified lodging into our MySQL database, as before.
  - Update the user entry in our MongoDB to add the ID of the lodging to the `lodgings` array of the specified owner.

- This will not take much work. Specifically, we can simply attach a `.then()` clause to the promise we created in `insertNewLodging()`. Inside this clause, we can call the `addLodgingToUser()` function we wrote above and return the promise *it* generates:

```
.then((lodgingID) => {
  return addLodgingToUser(lodgingID, lodging.ownerID);
});
```

- This completes our work here. We can leave the rest of the endpoint's code in tact. Specifically, if the call to `insertNewLodging()` succeeds, we will still return a success response containing the lodging ID (which, remember, we returned from `addLodgingToUser()`).

- One important thing to note here: there is a chance that, for some reason, our call to insert the lodging into our MySQL database could succeed, but for some reason, our call to insert that lodging's ID under the appropriate user within our MongoDB database could fail. In this case, our databases would be in an inconsistent state, with one of them containing the lodging while the other didn't.
- In situations like this in a production environment, we would typically add checks to insure that our databases remained consistent even if some database operations failed. Here, we are not performing such checks.

- This issue arises even when not dealing with two separate databases. For example, we would need to ensure consistency through two related writes into a single MySQL database.

Document-oriented DBs vs. relational DBs

- Now that we're considering a second type of database, it makes some sense to explore the differences between relational databases and document-oriented ones and to consider when one might choose one over the other.

- One of the major differences between document-oriented databases like MongoDB and relational databases like MySQL is that relational databases require data to abide by a ***schema***, i.e. a precise specification of the structure of the data, including all data fields and their data types.

- Document-oriented databases, on the other hand, are typically schema-less, meaning that each document in the database could potentially have a different structure, with different fields and/or different data types.

- For this reason, document-oriented databases are typically better suited to situations where the structure of the data is unknown or may change over time.

- In particular, no changes need to be made to a document-oriented database to allow records with a different structure to be stored, whereas changing the structure of records in a relational database can be a costly operation.

- Document-oriented databases are also excellent for storing hierarchical content, permitting a more natural, object-oriented approach to data storage.

- For example, a blog site's API could store each blog post as a document with the comments associated with that post embedded as an array of subdocuments. Here's what a MongoDB document for such an application might look like:

```
{
  title: "...",
  dateTime: "...",
  content: "...",
  authorName: "...",
  category: "...",
  comments: [
    {
      title: "...",
      dateTime: "...",
      content: "...",
      authorName: "...",
    },
    ...
  ]
}
```

- In a relational database, comments would have to be stored in a separate table from blog posts, breaking the natural subordinate relationship between the two and requiring either a JOIN or multiple SELECT statements to fetch all of the relevant data for a given blog post.

- Document-oriented databases used to be poorly-suited for cases where parts of a data hierarchy were repeated or cross-referenced.

- For example, in the document above, we are simply storing the name of the author of each post and each comment. Incorporating information about post and comment authors, such as username, display name, avatar image URL, etc. complicates things slightly.

- In this case we would have to do one of two things:
    - Store all of the user's data along with each post and comment they write.

    - Maintain a separate collection of users where information about each user is stored, and store a reference to a user from this collection along with every blog post and comment they write.

- The first option could potentially result in a huge amount of replicated data. For example, if a user writes 10 blog posts and 100 comments, each post and comment would have its own separate copy of the user's data. What happens in this case if the user updates their display name? We'd have no choice but to update every blog post and comment written by that user with the new name.

- The second option used present its own problems for document-oriented databases like MongoDB. In particular, in these databases it used to be difficult to perform JOIN-like operations to fetch relevant data from multiple collections at once (e.g. posts with comments from a posts collection and author information from a users collection). Instead, multiple queries were required.

- However, recent versions of MongoDB introduce features like the $lookup operation that make performing this kind of cross-referencing easier.

- Indeed, in recent years, MongoDB has caught up to or exceeded MySQL in many ways. One area in which MySQL still excels is ***transactional operations***.

- A transactional operation is actually a group of operations that are treated as though they are a single, atomic operation. If one operation anywhere in a transaction fails, the entire transaction fails, and the database is left in its original state, before the transaction started to execute.

- Transactions are well-suited for applications like bank software, where a common operation might be to deduct a certain amount from one account and to add that same amount to a different account. Here, transactions can help ensure the consistency of account balances.

- MongoDB does not support transactions, though support appears to be coming soon.