MongoDB

- We have nearly all of the pieces of a modern web application in place. Our last step towards implementing a complete application is to store its data on the back end and to use that data to generate pages that are sent to clients in response to their requests.

- We're already equipped to do this in a very rudimentary way by storing application data in a file on the back end, e.g. using the `fs` module. However, we want our data storage to be more robust than this, for a few reasons:
    - **We want to be able to easily interact with and update our application's data.** This is hard to do when the data is stored in files. For example, if all of our data is stored in a single JSON file, it's hard to update one or two fields within one or two objects in the data without rewriting the entire file.
    - **We want to be able to easily query the data.** Sometimes the queries we want to make are complex, asking only for those pieces of data that meet certain specified criteria (e.g. people over a certain age, businesses within a given radius of a specified location, etc.). Making these queries can be very challenging when data is stored in a simple file.
    - **We want our application to scale easily.** As our application grows, it will need to store more and more data on the back end. Just imagine how much data is stored on Facebook's servers, for example. Even at a fraction of Facebook's size, storing data in files makes it very difficult to grow an application.

- For these reasons and others, data for a web application is typically stored in a database that makes it easy to do the things we just listed.

- In general, there are two broad categories into which we can divide the databases that are typically used to power a modern web application: ***relational databases*** and ***document-based databases***.

- Relational databases, such as MySQL, store data in tables, where the rows of a table represent one instance of a specific kind of entity, and the columns of that row store the different fields of data associated with that instance.

- For example, if we wanted to use a relational database to power an application that displayed pages of photos of people, with one page to represent each specific person for whom photos were available, we might divide our data into two tables, one to store information about the people themselves (e.g. name, age, biography, etc.) and one to store information about the photos (e.g. the URL of the photo, a caption, etc.):

people

personId	name	age	bio
luke	Luke Skywalker	24	Jedi Knight
leia	Leia Organa	24	Princess and General
han	Han Solo	29	Pilot/Bandit
...	...	...	...

photos

photoId	personId	url	caption
1	luke	http://...	...
2	leia	http://...	...
3	leia	http://...	...
4	luke	http://...	...
5	han	http://...	...
...	...	...	...

- We could then construct queries on these tables to read data, insert data, update data, etc. in order to power our application.

- Document-based databases, on the other hand, store collections of ***documents***, where each document is a representation of a single instance of some kind of entity. In many document-based databases, such as MongoDB, a document is a collection of key:value pairs roughly equivalent to a JavaScript object.

- Importantly, the data in a document can be structured, with nested arrays and subdocuments.

- For example, if we wanted to use a document-based database to store the data for the application described above, our database might just contain one collection, where each document in the collection represented data about one person, including an array containing sub-documents representing photos of that person:

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

{
     personId: "leia",
     name: "Leia Organa",
     age: 24,
     bio: "Princess and General",
     photos: [...]
}
...
```

- Again, to power our application, we would formulate queries on this document collection to read and interact with the data stored there.

- Both relational databases and document-based databases have pros and cons, and which kind of database you choose (not to mention the specific database implementation you choose) will depend on your application, the the specific data you need to store, and the ways you want to be able to interact with your data.

- Here, we will restrict our focus to document-based databases, since working with them is somewhat conceptually simpler than working with relational databases.

- In fact, we will focus solely on MongoDB, which is a document-based database that is one of the most widely-used for powering web applications.

- MongoDB is a network service. Specifically, it runs behind a server process, and operations on the database are executed using commands issued by clients to the server over the network.

- Thus, to use MongoDB, you need two things:
    - A running MongoDB server.
        - You can host your own server, e.g. using tools from mongodb.com or a platform like Docker.
        - You can use a third-party MongoDB hosting service, such as mongodb.com or mlab.com.
    - A MongoDB client.

- Learning how to host a MongoDB server is beyond the scope of this course, so we'll assume you have access to a running MongoDB server and have a database set up there, and we'll focus on using a MongoDB client to interact with that server.

- We'll specifically look at two different MongoDB clients:
    - The MongoDB shell, which is a terminal-like interactive interface in which basic JavaScript commands can be run to interact with a MongoDB database.
    - The MongoDB Node.js driver, which makes it easy to interact with a MongoDB database from a Node.js application.

- We'll start by exploring how to interact with a MongoDB using the MongoDB shell.

- Assuming you have the MongoDB shell installed on your system, you can connect to a running MongoDB server using the `mongo` command from your favorite terminal application (or `mongo.exe` from Windows Shell).

- For example, from a Unix system, you could run a command like this, specifying your username and password, to connect to a MongoDB database called `myMongoDB` running on the host `classmongo.engr.oregonstate.edu`:

```
mongo -u <username> -p <password> \
     classmongo.engr.oregonstate.edu/myMongoDB
```

CRUD operations in MongoDB shell

- The four basic database operations are creating, reading, updating, and deleting data, which are often referred to collectively as the CRUD operations. We'll first look at how to perform these methods in the MongoDB shell.

- In the MongoDB shell, we can run basic JavaScript commands. To perform CRUD operations on our database (or to interact with it in other ways), the MongoDB shell provides a global object called `db`, which represents the entire database. Most of the commands we run in the MongoDB shell will call methods on this `db` object.

- Importantly, 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 example application above, we used one collection to store documents with information about people. We might name this collection `people`. If we did, 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 `people` collection like this:

```
db.people.insertOne({
     personId: "darth",
     name: "Darth Vader",
     age: 53,
     bio: "Sith Lord",
     photos: []
});
```

- Note that if we try in the MongoDB shell to insert into a collection that doesn't exist, that collection is automatically created and the data inserted into it. Thus, the method call above would succeed, even if the `people` collection didn't exist in our database before the method call.

- 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()`, the MongoDB shell automatically generates an `ObjectId` object and assigns it as the value of the document's `_id` field.

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

```
{
     "_id": ObjectId("5a1f3e8864be89eef867596a"),
     "personId": "darth",
     ...
}
```

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

```
db.people.insertMany([
     { personId: "r2d2", ... },
     { personId: "c3po", ... },
     { personId: "chewie", ... }
]);
```

- 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.people.find();
```

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

```
db.people.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 `personId` field has a value equal to `"luke"`, we could do so like this:

```
db.people.find({ perdonId: "luke" });
```

- Of course, if we ensure that the `personId` 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 `personId` 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 `bio` field equals `"Princess and General"`:

```
db.people.find({ age: 24, bio: "Princess and General" });
```

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

```
db.people.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 people collection to find all documents representing people whose age was at least 30, we could use the `$gte` (greater than or equal) operator:

```
db.people.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 bio is `"Princess and General"`:

```
db.people.find({ $or: [
     { age: { $gte: 30 } },
     { bio: "Jedi Knight" }
]});
```

- 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 `people` collection by ascending age, we could do so like this:

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

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

```
db.people.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()` and `updateMany()`

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

- Both of these methods 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 `update()` will update the first document whose `personId` equals `"luke"` by setting it's `bio` field to `"Reclusive Jedi Knight"`:

```
db.people.updateOne(
     { personId: "luke" },
     { $set: { bio: "Reclusive Jedi Knight" } }
);
```

- 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 `people` collection, incrementing each person's age by 1:

```
db.people.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 `update()` will add one photo object to the end of the `photos` array of the first document (if any) whose `personId` field is `"darth"`:

```
db.people.updateOne(
      { personId: "darth" },
      { $push: { photos: {
            photoURL: "http://...",
            caption: "..."
      }}
);
```

- 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/

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 `people` collection whose `personId` field is `"chewie"`:

```
db.people.deleteOne({ personId: "chewie" });
```

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 web applications.

- In a web application, database access is usually done from the server-side. There are two typical patterns for this access:
    - When a client requests a page in a web application for which content is dynamically generated, the server fetches the appropriate data from the database and then uses that data to generate an HTML page with which to respond to the client.
    - When a client makes an asynchronous call to an API route on a server, the server creates/reads/updates/deletes data from the database as needed. Once the database operation is complete, a response is sent back to the client indicating whether the operation was successful. If appropriate, the response includes a body containing data fetched from the database to satisfy the request.

- Given this paradigm, we will need to incorporate calls to access our MongoDB database into our server-side code. 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:

```
var 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>
```

- 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.

- Remember, environment variables are values that are set in the terminal environment in which a process is executed. Different terminal shells have different methods for setting environment variables. For example, in Bash, you can set environment variables using the `export` command, e.g.:

```
export MONGO_HOST="classmongo.engr.oregonstate.edu"
export MONGO_USER="cs290_hessro"
export MONGO_PASSWORD="hunter2"
export MONGO_DB_NAME="cs290_hessro"
```

- In order to read these values from the environment in our server code, we can do something like this:

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

    - Note that 27017 is the default port on which a MongoDB server runs.

- 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:

```
var mongoURL =
     'mongodb://' + mongoUser + ':' + mongoPassword + '@' +
     mongoHost + ':' + mongoPort + '/' + 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 when calling this method:
    - If any routes or other middleware in our server depend on our MongoDB database, we want to ensure that we have a successful connection to that database before allowing any of those routes or middleware functions to be called.
    - `MongoClient.connect()` is an asynchronous method. It takes as an argument a callback function, and if the connection to the database is successful, a reference to the connected client is passed as an argument to the callback after the connection to it is 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:

```
var mongoDBDatabase;
...
// Routes and middleware are defined here.
...
MongoClient.connect(mongoURL, function (err, client) {
  if (err) {
    throw err;
  }
  db = mongoDBDatabase = client.db(mongoDBName);
  app.listen(3000, function () {
    console.log("== Server listening on port 3000");
  });
});
```

- 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.

- The MongoDB driver allows us to call the same methods we saw above when we interacted with our database from the MongoDB shell: `find()`, `insertOne()`/`insertMany()`, `updateOne()`/`updateMany()`, and `deleteOne()`/`deleteMany()`.

- Now that we have an open connection to our database stored in `db`, we can start to explore how to implement the two different patterns of server-side database access we outlined above.

Using data from a MongoDB database to render a page

- Let's first look at how to use data from our database to dynamically render a page in response to a request from a client. We'll look at two different examples of this pattern of access within the context of the web application we described above, where we're serving pages of photos of people.
    - For these examples, we'll assume the data in our database is structured the same as in our discussion above about the CRUD operations and the MongoDB shell.

- As a first example of how to use data from our database to dynamically render a page, let's say we want our application to have a page that lists all of the people for whom we have photos. The route for this page might look something like this:

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

- Within this route's middleware function, we want to make a query to our database to obtain all of the documents in our `people` collection. To do this, we must first grab a reference to the `people` collection:

```
var people = db.collection('people');
```

- Once we have this reference to the `people` collection, we can call `find()` on it, passing an empty query object as the argument in order to fetch all of the documents in the collection. Importantly, the `find()` method returns a ***cursor***, which is an object that facilitates traversal over the returned documents:

```
var peopleCursor = collection.find({});
```

- There are various ways to get documents from the cursor object. A simple approach is to simply call `toArray()` on the cursor object to return all of the documents represented by the cursor into a simple array. The `toArray()` method takes as an argument a callback function that itself must take two arguments: one to hold a reference to an error if one occurred while accessing the database and one to old the array of documents if it was successfully fetched.

- Within the callback we pass to `toArray()`, we'll first check if there was an error accessing the database. If so, we'll respond with a 500 status to the client to indicate that there was a server-side error. Otherwise, we'll use the returned array of documents to render our people page:

```
peopleCursor.toArray(function (err, peopleDocs) {
  if (err) {
    res.status(500).send("Error fetching people from DB.");
  } else {
    /*
     * Use documents in peopleDocs to construct arguments
     * to our view template and then use res.render() to
     * render the page with the template and its arguments.
     */
    ...
  }
});
```

- Note that, if our query is successful, `peopleDocs` above will be an array containing the documents fetched. If the documents in our people collection look like they did above, `peopleDocs` will look something like this:

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

```
  {
    _id: ...,
    personId: "leia",
    name: "Leia Organa",
    ...
  },
  ...
]
```

- Next, we can use data from our database to render a page of photos for a single person. Our route for this page might look something like this, taking as a route parameter the ID of the person whose page of photos is to be rendered:

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

- The first thing we'll want to do in the middleware function for this route is to again grab a reference to our `people` collection and to call `find()` on that collection. This time when we call `find()`, however, we'll pass a query object formulated to grab the document corresponding to the `personId` specified in the request path:

```
var people = db.collection('people');
var peopleCursor = collection.find({
  personId: req.params.personId
});
```

- There are a few different scenarios that can arise here:
    - The `personId` specified in the request path did not match anyone in our `people` collection. In this case, we'll want to respond to the client with a 404 error, letting them know we don't have a page of photos for the requested person.
    - The `personId` specified in the request path *does* match someone in our `people` collection. In this case, we'll want to use that person's document to render their page of photos.

- In this case, we are interested in at most one document, and our database should not have more than one document matching the specified `personId`. Thus, we don't need to worry about converting the database results into an array using `toArray()`.

- Instead, we can use the cursor's `next()` method to simply get the first (and hopefully only) document in the query results. This method also takes a callback function as its argument, and the callback is passed an error (which will be `null` if there was no error reading the database) and a single object representing a document that matched our query (which will be `null` if no documents matched the query).

- We can use these callback arguments similar to the way we used them above, except this time, if the query returned no documents, we'll call `next()` to move to the next middleware function, assuming here that our server is set up so that this will result in a 404 error being sent back to the client:

```
peopleCursor.next(function (err, personDoc) {
  if (err) {
    res.status(500).send("Error fetching person from DB.");
  } else if (!personDoc) {
    next();
  } else {
    /*
     * Use the personDoc object to construct arguments
     * to our view template and then use res.render() to
     * render the page with the template and its arguments.
     */
    ...
  }
});
```

Updating a MongoDB database within an API route

- Finally, let's explore how to incorporate MongoDB access into an API route, i.e. a route that doesn't render a page but rather reacts to an asynchronous request made by the client by creating/reading/updating/deleting data in the database.

- We'll stick to the context of our photos application here. In particular, we'll now implement a route that will allow the user to make a POST request to our server in order to store a new photo for a specified person. The route will look something like this:

```
app.post('/people/:personId/addPhoto',
  function (req, res, next) {...});
```

- Specifically, the client will need to specify in the request path the `personId` of the person for whom a new photo is to be added. Information about the photo itself will be encoded in the request body.

- We'll specifically assume that the application is set up to generate JSON bodies for requests made to this route, where the JSON encodes an object like this:

```
{
  photoURL: "http://...",
  caption: "..."
}
```

- We'll also assume that the server process is already using the `body-parser` package to parse JSON request bodies.

- With all of this set up, we'll be able to expect requests to our POST route to have the photo's URL and caption available in `req.body.photoURL` and `req.body.caption`. The very first thing we'll do in our route's middleware function is make sure the request specifies at least a photo URL, which we'll consider the minimum amount of information for a photo. If the request body doesn't specify a photo URL, we'll respond to the client with a 400 status, indicating that the request was not correctly formed:

```
if (req.body && req.body.photoURL) {
  ...
} else {
  res.status(400).send("Request must specify photo URL.");
}
```

- Within the `if` block, if the client has in fact specified a photo URL in the request body, we'll want to make a call to our database to update the document corresponding to the `personId` specified in the request path. Specifically, we'll want to take the photo information encoded in the request body, create an object to represent that photo, and append that object to that document's `photos` array. We'll start by simply creating a simple object to represent the photo:

```
var photoObj = {
  photoURL: req.body.photoURL,
  caption: req.body.caption
};
```

- Now we're ready to update our database. As before, we'll need to grab a reference to the `people` collection in the database:

```
var people = db.collection('people');
```

- This time, instead of calling `find()` on our `people` collection, we'll call `updateOne()` to update a single document (the one matching the specified `personId`).

- In the Node.js MongoDB driver, `updateOne()` takes three arguments instead of two, as in the MongoDB shell. The first two arguments are the same as in the MongoDB shell: a query object specifying which documents should be updated and an update object specifying how to update those documents. The third argument must be a callback function, which will be called when the database update completes (or ends with an error):

```
people.updateOne(
  { personId: req.params.personId },
  { $push: { photos: photoObj }},
  function (err, result) {...}
);
```

- As above, the callback function is passed two arguments. The first will hold a reference to any error that occurred while trying to execute the update operation and will be `null` if no error occurred. The second argument will, if the update operation succeeded, hold a reference to an object containing information about the status of the operation (e.g. how many documents were updated).

- Here, we will simply respond to the client with a 500 status if there was any error executing the update operation. If the update operation was successful (whether or not it actually found any documents to update), we'll respond with a 200 status:

```
if (err) {
  res.status(500).send("Error inserting photo into DB.");
} else {
  res.status(200).send("Success.");
}
```

- With all of this in place, the client will be able to use this route to update the data stored in the database, so that when users request the photos page for the person for whom the photo was added, the new photo will be included.

- A similar pattern could be followed to perform many different kinds of database interaction, inserting new data into the database, updating different pieces of that data, and even removing data from the database, and these interactions can be triggered in response to user interactions within the client-side code.