Unit 10 – Databases

MongoDB – Non-relational Database

1. Nesting in MongoDB vs Joins in SQL
2. MongoDB stores everything as JSON
   1. Rather than tables, we have **collections** of documents.
      1. Think of a collection as a blob of stuff.
      2. Of which each document is a JS object
      3. Which store both arrays and objects
      4. Nesting arbitrarily deep.
      5. Emphasizes speed
      6. Cannot handle complex relationships like SQL.
         1. SQL is a relational database, meant for this purpose.
3. One-to-Many relationships in SQL…
   1. 2 tables with employees (with department\_id)
   2. And departments with id
   3. Would need to be joined.
4. One-to-Many in MongoDB
   1. Have a collection called “departments”
   2. With an object called “HR” in the title
   3. And an array of employees.
   4. Storing an array of items –
   5. This kind of embedding is MongoDB’s solution to foreign-key relationships.
   6. Flexibility in field properties and nesting, just like JSON.
5. Mongoose object-data modeling
   1. Used in conjunction with MongoDB.
   2. Mongoose library has the ability to enforce a schema
      1. Schemas are required in SQL
      2. Schema definitions and types can be enforced. Optional.
      3. Creating a schema instance and then creating a model to perform queries is standard practice…
         1. Each schema defines the shape/property/data types of each document.
   3. A **model** is what we use to interface with the database in Mongoose.
      1. This is how we update data, read data, write data, etc.
      2. This is a concrete instantiation of data that follows the schema.
      3. This is created with the “.model” method on the “mongoose” object.

const Name = mongoose.model(‘name’, schema)

* + 1. Javascript will use this model that we’ve created in order to communicate with MongoDB.

1. Mongoose Queries
   1. We can use a callback at the end of the query to handle errors.
   2. You can also use a promise approach.
2. CRUD in mongoose.
   1. These are just functions.
   2. **Find** or **findOne**
      * 1. *(conditions, [projection], [options], callback)*
        2. *(conditions, [projection], [options]).then(….)*
        3. Conditions – specifies conditions to match
        4. Projection – specifies which fields of document to return
        5. Options – sort, limit, skip
        6. **Make sure that you have error catches if nothing is returned!!**
        7. *Callback (err, returnedResults) => {…}*
      1. .find() returns **an array**. It will return an empty array – so you can check for array.length;
      2. findOne() **returns “null” if nothing is returned.**
   3. **Model.prototype.save**  **or Model.create**
      1. Define the object with the data and “create” or “save”.
      2. “save” method can only make one document at a time.
      3. “create” method can make multiple documents at a time by passing in an array of objects.
      4. Just be careful of syntax.
   4. Delete
      1. **findOneAndDelete**
         1. returns the document to you before you delete.
      2. **deleteOne**
         1. You’ll have to run a separate query to make sure you’re deleting the right thing.
      3. **remove**
      4. **deleteMany**
   5. Update
      1. **findOneAndUpdate**
         1. finds, updates, and returns **the old version of the document!!**
         2. “result” from findOneAndUpdate will show you the old version.
         3. To return the updated doc, we pass in {new: true} into the options object parameter.
      2. **updateOne**
      3. **updateMany**
      4. *Option parameter for updating -* **upsert**: **true**
         * 1. if this document exists, “update” it;
           2. if it doesn’t, “insert” it.
   6. Mongoose allows RegExp.
   7. Pre/post middleware
      1. For example, run a function “pre” ‘save’ to hash the password that was inputted.
3. MongoDB vs SQL
   1. We’ll have to make a decision about which we should use.
   2. When data is deeply nested and we have to find a value within it, that’s not so bad in MongoDB.
   3. But if we have to use a deeply nested parameter as a query, then we would be in a bit of trouble. To query for a deeply nested parameter and use that to return top-level information, we would have to go through every single
   4. SQL is still pretty much top dog – remember that!
   5. SQL’s strength is guaranteed referential integrity
      1. This is critical for datasets
      2. You don’t want to run complex app logic to fix consistency
      3. However, it’s a little slow.
   6. MongoDB can be used for certain parts of the application, but it can be unwieldy to serve as a main DB.
   7. Sometimes, it’s quite appropriate.
      1. E-commerce would be a “canonical” instance of MongoDB usage.
      2. A bunch of toys have disparate attributes. They are not particularly related to one another.
      3. So the storage of a lot of unrelated information can be quite useful.
   8. So we can do something like this:
      1. Metadata is stored in MongoDB which specifies what fields we want.
      2. And MongoDB generates an ID.
      3. And then we can have a general SQL database that points to the details by that mongoDB ID.
      4. They cannot communicate with one another, but we would use our server to query one and the other together.