**Mongoose Introduction**

* Mongoose is an Object Data Modeling (ODM) library that allows MongoDB to be used in a Node.js application

**Installation and Connection**

* To use mongoose, install it in via : ‘npm install mongoose’
* We can then require that mongoose library and store the returned object in a constant called ‘mongoose’
* We can then use the ‘mongoose’ constant’s ‘connect’ method to connect to our database. This ‘connect’ method takes in three agreements.
  + The first parameter is the url to our database. To use the locally installed version of mongodb as our database, we use this url: mongodb://localhost/INSERT\_DATABASE\_NAME. Recall that in mongodb, if there is no database with that name, adding data to it will automatically create the database.
  + The second optional parameter is a callback function which gets executed when we successfully connect to the database.
  + The third optional parameter is a callback function which gets executed when we fail connect to the database. This callback function takes the error as a parameter.
* Even though it may take a while to connect to the database, if we start interacting with mongodb via mongoose, it’ll queue up all the commands we make and only make those commands after we connect.
* Text

  Description automatically generated

**Three Main concepts**

* There are three main concepts to understand in mongoose: schema, model, query
* A schema defines what the structure of our data looks like. So, if we have a user object, we’re going to have a user schema that says a user must have a ‘name’, ‘email’, etc.
* A model is the schema in an actual form that can be used. So a model could be like an individual user object from the database that we can interact with
* A query is a command we make against the mongodb database

**Schema/Models**

* We generally create a new file for each Schema.
* In our example, we will create a User.js schema file
* First, we will require mongoose and store it in a ‘mongoose’ constant
* 
* To create a Schema, we use ‘new mongoose.Schema()’.
* This ‘Schema’ method takes in an object will all the different options we need for our schema.
* The object we pass in will have key-value pairs where the key represents the field in our mongodb object in our database and the value will be the type.
* Graphical user interface, text

  Description automatically generated
* This ‘Schema’ method returns a schema object that we can store in a constant, which we will call ‘userSchema’.
* Now that we created a schema for our users, we can create a model via the ‘mongoose’ constant’s ‘model’ method. This ‘model’ method takes in two parameters. The first is the name of our model (this will be the collection name inside of mongodb). The second parameter is the schema object.
* We can then export this model to be used in other files.
* 
* Thus, our User.js file could look like the following:
* Graphical user interface, text

  Description automatically generated

**Creating and Saving Models**

* To use this User model in other files, we can import it as shown below. Notice that this User model object has methods that which are mongoDB methods such as ‘find’.
* Graphical user interface, text, application, chat or text message

  Description automatically generated
* To create a new user and save it to the database, we can do the following:
* Text

  Description automatically generated
* Notice in line 6, we created a local user object. Also notice this user doesn’t need to have all the fields we specified in the userSchema.
* In line 7, we save the user to the database by using the ‘save()’ method which is asynchronous. We then log the saved user to the console.
* The above code snippet would create the following console output:
* Text

  Description automatically generated
* Notice the auto-generated ‘\_id’ field which is what mongoDB normally.
* Notice the ‘\_\_v’ field which mongoose automatically generates. This field is used to keep track of versioning and is used internally by mongoose, so we don’t need to worry about it.
* There is another way to create a user and saving them to the database. This is done via the ‘create’ method which does the exact same thing as creating a user via ‘new User()’ and then saving to the database via ‘save()’.
* Thus, to create a new user and save it to the database, we can also do the following:
* Text

  Description automatically generated
* Output:
* Text

  Description automatically generated

**Updating Models**

* To update models, we can do the following:
* Text

  Description automatically generated
* Output:
* Text

  Description automatically generated
* The first object that is logged is the object we initially created on line 6 which only has a ‘name’ property.
* Notice to change the value of an existing field, we can say user.name = ‘Kyle’ as shown on line 7.
* Notice we can add a new ‘age’ field via user.age = 23 as shown on line 8. We can create a this new ‘age’ field since we already defined this field in the Schema.
* Notice we cannot add a new ‘gender’ field via user.age = 23 as shown on line 9. We cannot create a this new ‘gender’ field since we did not define this field in the Schema.
* We then log the user to the console to get the second object which has a ‘name’ and ‘age’ property.
* We then save the user to the database in line 12.

**Schema Types**

* There are many different data types that Schema field can be.
* Text

  Description automatically generated
* In the above example, the bestFriend field references another user. Thus, we make it have a data type of an object id via mongoose.SchemaTypes.ObjectId
* If instead of hobbies : [String], we had hobbies: [], we could have an array of anything.
* Notice that the address field is a nested object.
* Now, we can create a new User as show below in server.js:
* Text

  Description automatically generated
* Output:
* Text

  Description automatically generated
* Recall that the address field is a nested object. We could have a nested object (as shown above), or we can create a new schema.
* Text

  Description automatically generated
* Now, we can create a new user as before and we would get the following logged to the console:
* Text

  Description automatically generated
* Notice that the address field in the console has an \_id field. This is because we created a new Schema for the address field and mongoDB auto-generates this ‘\_id’ field.

**Schema Validation**

* Since we defined datatypes in our User schema, when we create a new user, the parameters must be valid and should follow the Schema. Invalid parameters could look like the following (we have a string for the ‘age’ property when it should be a number):
* Text

  Description automatically generated
* This would log the following to the console:
* 
* The above was an example of type validation, but there are other types of validation
* Text

  Description automatically generated
* Note that for the ‘email’ field, we have lowercase: true. This does not mean the email entered must be lowercase. Rather it means when an email is entered, it will be converted to lowercase. A similar thing applies to uppercase : true.
* Note that for the ‘createdAt’ field, we have a default value which is a function. Whenever, we create a user, this function is executed. Note that we can’t use default : Date.now() since that will return the date when the Schema is created, not when the user instance is created.
* Note that the ‘createdAt’ field also has immutable : true which means that this ‘createdAt’ field can never be changed, even if we try to change it later via ‘user.createdAt = getNewDate()’
* Note that for the ‘age’ field, we have a validate object.
  + This validate object has a ‘validator’ field which is a function. This validator function allows us to create custom validation. This validator function takes in a value and return true/false indicating if the value is valid or not. In the above example, the validator function only accept ages that are even.
  + This validate object has a ‘message’ field which is a function. This function is executed when the validator function returns false. This function takes in a props object and we can get the value via props.value. In the above example, when a user enters an age such that is odd, we return a message saying the age is not an even number.
* One issue with validation is that validation only runs when we execute the create() or save() methods. Thus, mongoose’s built-in update methods don’t go through validation. As a result, we shouldn’t use those update methods and instead use the save() method.
* Note: Mongoose now support the validation of the fields of $set and $unset operators when you include the runValidators: true option in the update call. (but this should still be avoided)

**Query**

* Finding document by id
  + COLLECTION\_NAME.findById(“INSERT\_ID”) gives the document with an id of INSERT\_ID
  + Ex:
    - A screenshot of a computer

      Description automatically generated with medium confidence
    - Text

      Description automatically generated
* Normal mongodb queries (such as find(), exists(), findOne(), etc) work too
  + Ex:
    - Text

      Description automatically generated
    - Text

      Description automatically generated
* In mongodb, the find() method’s syntax is complicated so mongoose has queries to replace that syntax as shown below.
* Ex:
  + Text

    Description automatically generated
  + In the above query, we are saying return an array of users where each user has a name equal to ‘Kyle’ and their ‘age’ is greater than 13 and less than 27, and only take the first 2 users
  + Output:
  + Text

    Description automatically generated
* Populate
  + Suppose we have these two documents in our Users collectionText

    Description automatically generated
  + Notice user2 is user1’s bestFriend.
  + If we query for user1 and log it to the console, we see the following:
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Text

    Description automatically generated
  + Instead of seeing the id of the best friend, suppose we want to see the document associated with that id.
  + To do so, we go to the schema and add a ‘ref’ field to the bestFriend ‘field’. The value of this ‘ref’ field is the name of the collection that the id belongs to.
  + Text

    Description automatically generated
  + Now, when we query for user1, we chain on the ‘popualte’ method. This ‘populate’ method takes in the field we want to populate as the parameter. In our case, the field we want to populate is ‘bestFriend’.
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Now when we run our code, we see the following:
  + Text

    Description automatically generated
  + Notice in the ‘bestFriend’ field, we no longer just see the id, but we see that this field is populated with the document data that the id referenced.

**Schema Methods/Virtuals**

* After we define our schema, we can add methods onto the schema.
* To create a method that will be accessible to each instance of our model, we can say userSchema.method.INSERT\_FUNCTION\_NAME = FUNCTION. Note that we cannot use an arrow function since we will use the ‘this’ keyword to reference the induvial instance we are working with. These methods are very useful when we want to do a things related to our models and we don’t want to define that code everywhere, we can just define it on the model itself.
  + Ex:
  + Suppose we create a sayHi function that all instances of our User model will have access to.
  + Text

    Description automatically generated
  + We can now call this sayHi method on a user model instance as shown below:
  + Text

    Description automatically generated
  + Output:
  + 
* To create a method that will be available on the model itself, we can say userSchema.statics.INSERT\_FUNCTION\_NAME = FUNCTION. Note that we cannot use an arrow function since we will use the ‘this’ keyword to reference the induvial instance we are working with.
  + Ex:
  + Suppose we create a findByName function that our User model itself will have access to.
  + Text

    Description automatically generated with medium confidence
  + We can now call this findByName method on the User model as shown below:
  + Text

    Description automatically generated
  + Output:
  + Text

    Description automatically generated
* To create a method that we can chain onto a query (a query is when we use find() or where()), we can say userSchema.query. INSERT\_FUNCTION\_NAME = FUNCTION. Note that we cannot use an arrow function since we will use the ‘this’ keyword to reference the induvial instance we are working with.
  + Ex:
  + Suppose we want to create a byName method that we can chain onto a User model query.
  + Graphical user interface, text

    Description automatically generated
  + We can now chain this byName method onto a User model query as shown below:
  + Text

    Description automatically generated
  + Notice that we can also chain on other methods such as limit() or sort()
  + Output:
  + Text

    Description automatically generated
* A virtual is a property that is not defined in the Schema itself but is a property we virtually create. We can access this virtual from model instances.
* To create a virtual, we can say userSchema.virtual(‘INSERT\_VIRTUAL\_PROPERTY\_NAME’).get(FUNCTION). Note that we cannot use an arrow function since we will use the ‘this’ keyword to reference the induvial instance we are working with.
  + Ex:
  + Suppose we want to create a ‘namedEmail’ virtual which we can then access from user model instances.
  + Text

    Description automatically generated
  + We can access this ‘namedEmail’ virtual from a user model instance as shown below:
  + 
  + Output:
  + 
  + Notice ‘namedEmail’ is not actually a property in the schema and is not saved in the database. But the way this property makes it seem like it is, hence the name virtual.