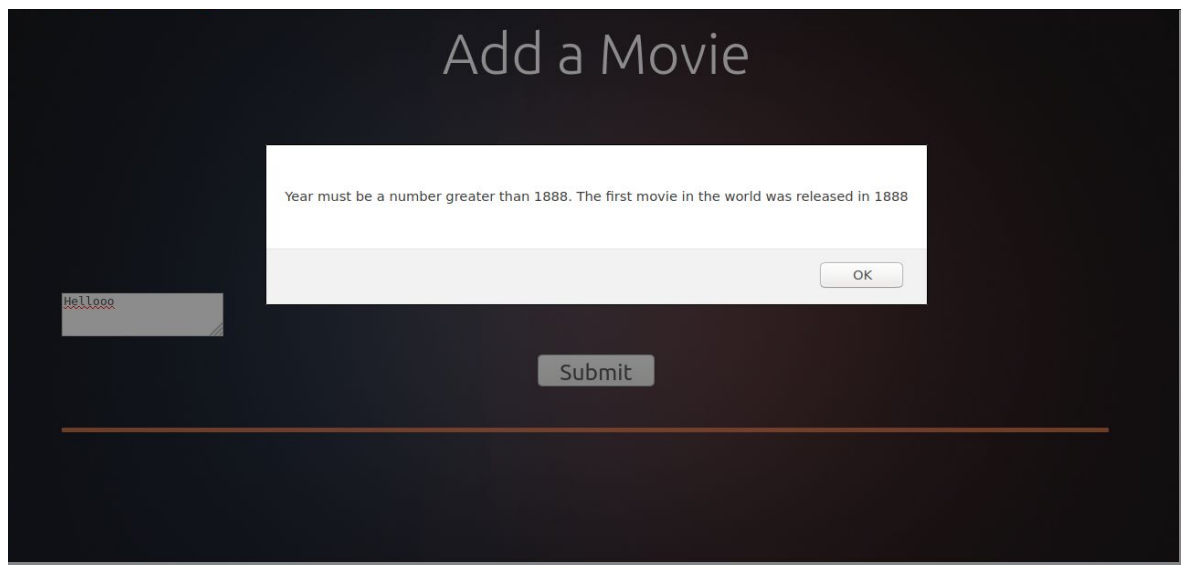


Software Testing

While software can be tested manually by humans, who can try to give input and see if it produces the desired output, using automated testing tools makes it easier to continuously check applications as small and small edits are being made to it. Through this tutorial, we will look at various types of testing. We will begin with a small section on **validating data** before it goes into the database using Mongoose validators, and then we will look at API testing of the backend endpoints. In another upcoming sheet, front-end testing and end-to-end testing will be explained. We will continue using the [sample movies application](#) (Github) to explain all of these.

Database Validation

While this is not testing exactly, validating (or checking) values before pushing them into the database is often good practice. For example in the MERN sample application that has been used in previous tutorials, you will notice there's an add movie functionality that prevents you from adding movies which have a year of release before 1888 (because that's when the first movie was released).



This check is just in the front-end though - the React app makes sure that the year of release > 1888 before sending it to the backend. If we send a POST request to `/api/movie/add` with an impossible year of release (like, say 1887) it would get added to the database without any errors. Checking the data ONLY in the front-end can cause issues if your API is used outside of that front-end (say by a different app). We could probably add code in the backend endpoint to check this before pushing the data in, which is shown in the next screenshot. The lines 27-29 in the code could achieve this.

```

~/mern-crud/routes/api/movie.js (examples) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
server.js x movie.js x
18
19     } else {
20         res.status(401).send("Unauthorized")
21     }
22 });
23
24 app.post("/api/movie/add", (req, res) => {
25     // add a movie to the database
26     var movie_object = {title: req.body.title, desc: req.body.desc, year: req.body.year};
27     if (req.body.year < 1888) {
28         res.send({status: false, error: "There are no movies before 1888."});
29     }
30     else if (req.isAuthenticated()) {
31         Movie.create(movie_object, (err, obj) => {
32             if (err) {
33                 console.log(err);
34                 res.send({status: false, error: err});

```

Another way of achieving the same functionality would be adding a validator in the Mongoose schema for the Movies object. Right now, our movies schema looks something like this. We will be editing Line #13 to add two conditions: (1) the first movie was released in 1888, (2) only movies released so far can be stored in the database, i.e. the year cannot be greater than 2000

```

5  /**
6  var mongoose = require("mongoose");
7  var Schema = mongoose.Schema;
8
9  // Creates schema for movie with required attributes and da
10 var MovieSchema = new Schema({
11     title: {type: String, required: true, unique: true},
12     desc: {type: String, required: false},
13     year: {type: Number, required: true, unique: false},
14     cast: [{type: String}], //This attribute is an array whi
15 });
16
17 const Movie = mongoose.model("Movie", MovieSchema);
18 // Below code used for exporting the Schemas
19 // Used when we need to import the below Schemas in another
20 module.exports = {Movie: Movie};
21

```

The 'year' attribute's description would look something like this after adding these two changes:

```
year: {
  type: Number,
  required: true,
  unique: false,
  min: 1888,
  max: 2020
}
```

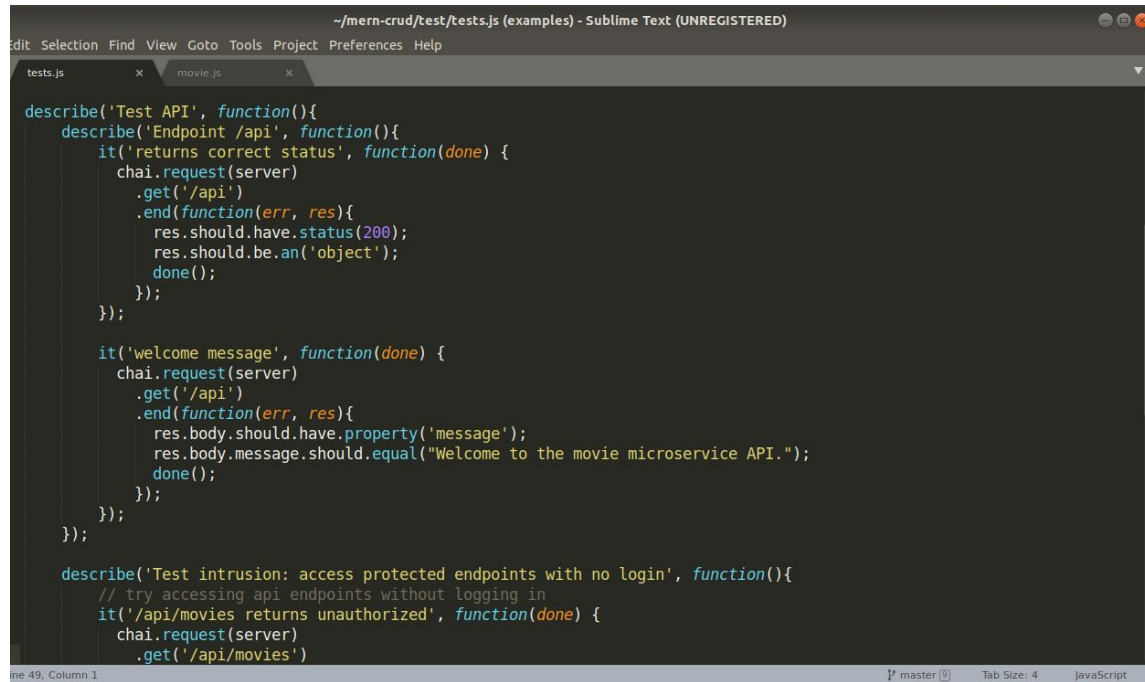
Min and max are two validators available for numeric type data. You can read more about validators in the Mongoose documentation¹. When you try adding a new movie with year, say 2021, using the add movie React form, you'll notice this error on your terminal (Express server), which shows you that the database has rejected the entry. Reloading the all movies page will also confirm that the erroneous movie hasn't been added to the database.

```
at formatWithOptions (internal/util/inspect.js:1093:40)
at Object.Console.<computed> (internal/console/constructor.js:272:10)
at Object.log (internal/console/constructor.js:282:61)
at /home/rohit/mern-crud/routes/api/movie.js:33:23
at /home/rohit/mern-crud/node_modules/mongoose/lib/utils.js:268:11
at /home/rohit/mern-crud/node_modules/mongoose/lib/model.js:4810:21
at /home/rohit/mern-crud/node_modules/mongoose/lib/model.js:3101:16
at /home/rohit/mern-crud/node_modules/mongoose/lib/model.js:3116:18
at callbackWrapper (/home/rohit/mern-crud/node_modules/mongoose/lib/model.js:3070:20)
at /home/rohit/mern-crud/node_modules/mongoose/lib/model.js:4791:16
at /home/rohit/mern-crud/node_modules/mongoose/lib/utils.js:268:11
at /home/rohit/mern-crud/node_modules/mongoose/lib/model.js:4810:21
at /home/rohit/mern-crud/node_modules/mongoose/lib/model.js:485:16 {
  errors: {
    year: MongooseError [ValidatorError]: Path 'year' (2021) is more than maximum allowed value (2020).
      at new ValidatorError (/home/rohit/mern-crud/node_modules/mongoose/lib/error/validator.js:29:11)
      at validate (/home/rohit/mern-crud/node_modules/mongoose/lib/schematype.js:1061:13)
      at /home/rohit/mern-crud/node_modules/mongoose/lib/schematype.js:1115:11
      at Array.forEach (<anonymous>)
      at SchemaNumber.SchemaType.doValidate (/home/rohit/mern-crud/node_modules/mongoose/lib/schematype.js:1070:14)
      at /home/rohit/mern-crud/node_modules/mongoose/lib/document.js:2323:9
      at processTicksAndRejections (internal/process/task_queues.js:76:11) {
        message: 'Path 'year' (2021) is more than maximum allowed value (2020).',
        name: 'ValidatorError',
        properties: [Object],
        kind: 'max',
        path: 'year',
        value: 2021,
        reason: undefined,
        [Symbol(mongoose:validatorError)]: true
      }
  },
  _message: 'Movie validation failed',
  name: 'ValidationError'
}
```

¹ [Mongoose v5.9.6: Validation](#)

Backend API Testing With Mocha and Chai

Two sample endpoint tests using Mocha and Chai are shown in the repository. Open the file `test/tests.js` to see them. For your convenience, here's the relevant code snippet:



```
describe('Test API', function(){
  describe('Endpoint /api', function(){
    it('returns correct status', function(done) {
      chai.request(server)
        .get('/api')
        .end(function(err, res){
          res.should.have.status(200);
          res.should.be.an('object');
          done();
        });
    });

    it('welcome message', function(done) {
      chai.request(server)
        .get('/api')
        .end(function(err, res){
          res.body.should.have.property('message');
          res.body.message.should.equal("Welcome to the movie microservice API.");
          done();
        });
    });
  });
});

describe('Test intrusion: access protected endpoints with no login', function(){
  // try accessing api endpoints without logging in
  it('/api/movies returns unauthorized', function(done) {
    chai.request(server)
      .get('/api/movies')

```

In the first few lines of the file, we import `chai` and `chai-http`, and set up the server that the tests should use. The next few lines show the basic formats of tests, with an example on basic arithmetic. Then, we create a bunch of test case to test the API endpoints. To make assertions, we use the “should” function - you can read up on the various ways “should” can be used in the [ChaiJS Documentation](#). To run the tests, I've added a “test” script to the package json that looks like this:

```
"test": "NODE_ENV=test mocha --exit test/**/*.js"
```

Note: The `--exit` in the command is to force the test script to end after all the tests are done.

This [medium article](#) does a good job at explaining what `describe()`, `it()`, and other functions used in the code mean. In the code sample provided, we do not test authentication-protected endpoints much - we just tested one endpoint to verify that it's not sending data if the user is logged out. If you want to test authentication-protected endpoints thoroughly, you can use the `beforeAll()` function to log in before starting the tests, and then use that token while making the tests. ([this article](#) shows code that can achieve that - they are, however, using Jest and Supertest. The overall structure of the code will look the same for Mocha and Chai as well though).

To summarise, if you want to unit-test API endpoints in your repository:

1. npm install chai chai-http (these are the required libraries)
2. Add `"test": "NODE_ENV=test mocha --exit test/**/*.js"` to your "scripts" in package.json
3. Write the unit tests - for sample code, see the repository linked to this tutorial, or any of the linked Medium articles/documentation.
4. Run the tests using "npm test". You'll get a summary of the test results on the console:

```
rohit@ardula-Inspiron-3558:~/mern-crud$ npm test

> movie-crud@1.0.0 test /home/rohit/mern-crud
> NODE_ENV=test mocha --exit test/**/*.js

mongodb://localhost:27017/movies-crud

Basic Math
  #indexOf()
    ✓ should return -1 when the value is not present
  Addition
    ✓ should return 3 when 1, 2 is added

Test API
  Endpoint /api
    ✓ returns correct status
    ✓ welcome message
  Test intrusion: access protected endpoints with no login
    ✓ /api/movies returns unauthorized

5 passing (69ms)
```

5. Try committing your code to Github. If you have followed the previous tutorial on Continuous Integration and Deployment, you can see that the application is tested and then redeployed. (green tick mark beside the timestamp)



6. You're in business. You can go back and start adding more endpoint tests until you've covered everything. Now, if you make changes to any of the endpoints, these tests will hopefully catch anything that's broken before an app is deployed.