← Mongoose Crud Tutorial | Mongoose Relationships Tutorial →

Mongoose Validation Examples

Just like all other frameworks, Mongoose provides a way to validate data before you save that data to a database. Data validation is important to make sure that “bad” data does not get persisted in your application. A benefit of using Mongoose when inserting data into MongoDB is its built-in support for data schemas, and the automatic validation of data when it is persisted. You would not get this without Mongoose. Mongoose’s validators are easy to configure. When defining the schema, a developer can add extra options to the property that should be validated. Let’s look at some basic examples of validation in Mongoose now.

Getting Started With `required`

Right now we have a schema in Mongoose which has no validation. In other words, all the properties defined below are optional. If you provide each property when creating a document, great! If not, that’s great too!

```
const gameSchema = new mongoose.Schema({
    title: String,
    publisher: String,
    tags: [String],
    date: { type: Date, default: Date.now },
    onSale: Boolean,
    price: Number
});
```

Of course it is almost always necessary to Validate any data you want to persist. We can modify the schema to add validation like so. In the snippet below, we are making the `title` of the game mandatory. It is no longer optional. We can do this with the `required` property.

Douglas Crockford The Good Parts Examples
Laravel RESTful Controllers
PHP String Helper Functions
Laravel File Structure
How To Toggle Exception Handling
VueJS Subnet Calculator
How To Refactor Code To A Dedicated Class
How To Use WordPress Excerpts
Axios Powered VueJS Form Component
Vi Editor Tutorial For Beginners
Linux Keyboard Tricks
Mentions And Notifications
JSON Web Token Authentication With Node.js

```
const gameSchema = new mongoose.Schema({
    title: { type: String, required: true },
    publisher: String,
    tags: [String],
    date: { type: Date, default: Date.now },
    onSale: Boolean,
    price: Number
});
```

With our validation in place for the title of the game, let's try to save a game to the database without specifying a title.

```
const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost/mongo-games')
    .then(() => console.log('Now connected to MongoDB!
    .catch(err => console.error('Something went wrong',

const gameSchema = new mongoose.Schema({
    title: { type: String, required: true },
    publisher: String,
    tags: [String],
    date: { type: Date, default: Date.now },
    onSale: Boolean,
    price: Number
});

const Game = mongoose.model('Game', gameSchema);

async function saveGame() {
    const game = new Game({
        publisher: "Nintendo",
        tags: ["adventure", "action"],
        onSale: false,
        price: 59.99,
    });

    const result = await game.save();
    console.log(result);
}

saveGame();
```

We can run index.js using `node index.js` at the terminal to test this out.

Top 12 Websites for Twi Bootstrap

How To Make HTTP Requests In Angular Using Observables

Most Useful JavaScript Array Functions

How To Use An Interface In Angular

Refactor The Laravel Regex Tool To Use Repositories and Dependency Injection

C# Relationships Between Classes

Moving From Routes to Controllers in Laravel

Getting Started With Python 3

Laravel Vue Component Example

Mongoose Validation Examples

What Is The CSS Box Model?

Vue.js Tutorial

The Top 17 Most Popular PHP Array Functions

Html Tutorials For Beginners

Getting Your Database Set Up For Use With Laravel

Reference Types And Value Types In C#

```
mongo-crud $node index.js
(node:10176) UnhandledPromiseRejectionWarning: Unhandled
promise rejection
(rejection id: 2): ValidationError: Game validation failed:
title: Path title
is required. (node:10176) [DEP0018] DeprecationWarning:
Unhandled promise rejections
are deprecated. In the future, promise rejections that are
not handled will
terminate the Node.js process with a non-zero exit code.
```

Interesting. We get a lot of error information about an unhandled promise rejection. We can fix this by updating our logic in the `saveGame()` function. The good thing however is that within the information listed we do see that the validation worked as *Game validation failed: title: Path* `title` *is required* tells us so. Let's update the code to handle the promise correctly by implementing a try/catch block.

```
async function saveGame() {
    const game = new Game({
        publisher: "Nintendo",
        tags: ["adventure", "action"],
        onSale: false,
        price: 59.99,
    });

    try {
        const result = await game.save();
        console.log(result);
    } catch (err) {
        console.log(err.message)
    }
}

saveGame();
```

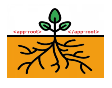
Running the index.js file now give us an easier to read message.

```
mongo-crud $node index.js
Game validation failed: title: Path title is required.
```

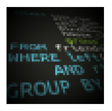
Great! Validation is working. It is important to note that this example of validation in Mongoose is just that, validation in Mongoose. This has nothing to do with data validation at the database, or MongoDb level. Another thing to note is that this type of validation in Mongoose is complimentary to a validation package like Joi which we used in the **node rest api tutorial**. By using both validation at the REST layer and the Mongoose layer, you can ensure that faulty documents will not be persisted to the database.

More About Built In Validators

How To Select Raw SQL Using Eloquent


Angular Root Component Tutorial

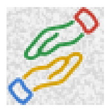
Install MongoDB With Compass On Windows


MySQL Group By Having Limit Offset and More!

Linux Files And Directories Commands

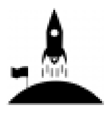
How To Add Routes and Models To Node Rest API

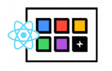
How To Send Email To New Users


Laravel String Helpers

MySQL Tutorials For Beginners

Twitter Bootstrap Classes

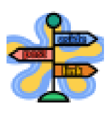

Encapsulation For Hackers


Introduction To React.js

Laravel Collections Tutorial

Learning About Regular Expressions

What Is Twitter Bootstrap?


WordPress Post Formats

What Is A WordPress Archive Page?

Twitter Bootstrap Tutorials

In the section above we saw how to use the `required` property to make it mandatory that a user provide the title of a game when persisting to the database. You can also use a function with `required` to conditionally require something. In the snippet below, we are saying that if the game is on sale, then the price is required.

```
const gameSchema = new mongoose.Schema({
  title: { type: String, required: true },
  publisher: String,
  tags: [String],
  date: { type: Date, default: Date.now },
  onSale: Boolean,
  price: {
    type: Number,
    required: function () { return this.onSale }
  }
});
```

Now we can try to save a game, but we will omit the title and price to see if our validation rules are still working. Our logic to insert the game is here.

```
const Game = mongoose.model('Game', gameSchema);

async function saveGame() {
  const game = new Game({
    publisher: "Nintendo",
    tags: ["adventure", "action"],
    onSale: true,
  });

  try {
    const result = await game.save();
    console.log(result);
  } catch (err) {
    console.log(err.message)
  }
}

saveGame();
```

Running the program shows that these rules are working well. Both price and title are required, so the game is not persisted.

```
mongo-crud $node index.js
Game validation failed: price: Path price is required.,
title: Path title is required.
```

minlength and maxlength

- JavaScript Functions Tutorial
- Getting Started With Testing In Laravel
- How To Create A Custom Menu In WordPress
- Create React App Tutorial
- Multi-Column Layout With CSS
- Underscore JS Each Function
- Php Tutorials For Beginners
- React useState Hook
- Introduction To The D3 JavaScript Library
- Twitter Bootstrap Modal Tutorial
- Simple Styling Techniques For React Elements
- ES6 Arrow Function Tutorial
- Angular Table Filter Component
- How To Use The Linux Terminal
- How To Set Up Form Submission In Laravel
- WordPress Dashboard Tutorial
- Feature Test vs Unit Test And Adding Replies To Threads

In addition to making a string required, you can also specify the minimum length and maximum length it should be. Consider this schema.

```
const gameSchema = new mongoose.Schema({
    title: {
        type: String,
        required: true,
        minlength: 4,
        maxlength: 200
    },
    publisher: String,
    tags: [String],
    date: { type: Date, default: Date.now },
    onSale: Boolean,
    price: {
        type: Number,
        required: function () { return this.onSale }
    }
});
```

Now, lets provide a title, but with only 3 characters and see what happens.

```
const Game = mongoose.model('Game', gameSchema);

async function saveGame() {
    const game = new Game({
        title: "Pac",
        publisher: "Nintendo",
        tags: ["adventure", "action"],
        onSale: true,
    });

    try {
        const result = await game.save();
        console.log(result);
    } catch (err) {
        console.log(err.message)
    }
}

saveGame();
```

When we run the program the validator tells us that our title is too short.

```
mongo-crud $node index.js
Game validation failed: price: Path price is required.,
title: Path title (Pac) is shorter than the minimum allowed
length (4).
```

enum validation

Creating Databases and Tables in MySQL

VueJS Image Upload

Angular Templates and Styles

Angular Styles Vs StyleUrls

Angular Pipes And Data Binding

The Top 15 Most Popular JavaScript String Functions

Upgrading VueJS

When creating a game, we are assigning some tags to it. Using enum validation, we can specify the available tags one could use. Below we are saying that the tags for a game must be any of sports, racing, action, or rpg.

```
const gameSchema = new mongoose.Schema({
    title: {
        type: String,
        required: true,
        minlength: 4,
        maxlength: 200
    },
    publisher: String,
    tags: {
        type: [String],
        required: true,
        enum: ['sports', 'racing', 'action', 'rpg']
    },
    date: { type: Date, default: Date.now },
    onSale: Boolean,
    price: {
        type: Number,
        required: function () { return this.onSale }
    }
});
```

Now we try to save a game using a tag we have not accounted for in the enum validation, adventure.

```
const Game = mongoose.model('Game', gameSchema);

async function saveGame() {
    const game = new Game({
        title: "Pacman",
        publisher: "Nintendo",
        tags: ["adventure", "action"],
        onSale: true,
        price: 29.99
    });

    try {
        const result = await game.save();
        console.log(result);
    } catch (err) {
        console.log(err.message)
    }
}

saveGame();
```

Sure enough, trying to insert that game into the database fails and we get the error that `adventure` is not a valid enum value for path `tags`.

```
mongo-crud $node index.js Game validation failed: tags.0:
adventure is not a valid enum value for path tags.
```

Custom Validators

You may also set up a custom validator in Mongoose. Here we will modify the validation for tags such that a user must provide more than one.

```
const gameSchema = new mongoose.Schema({
    title: {
        type: String,
        required: true,
        minlength: 4,
        maxlength: 200
    },
    publisher: String,
    tags: {
        type: [String],
        validate: {
            validator: function (v) {
                return v.length > 1
            },
            message: 'You must provide more than 1 tag.'
        }
    },
    date: { type: Date, default: Date.now },
    onSale: Boolean,
    price: {
        type: Number,
        required: function () { return this.onSale }
    }
});
```

Now we try to save a game and only provide one tag.

```
const Game = mongoose.model('Game', gameSchema);

async function saveGame() {
    const game = new Game({
        title: "Pacman",
        publisher: "Nintendo",
        tags: ["arcade"],
        onSale: true,
        price: 29.99
    });

    try {
        const result = await game.save();
        console.log(result);
    } catch (err) {
        console.log(err.message)
    }
}

saveGame();
```

Running the program gives us the validation error we expect.

```
mongo-crud $node index.js
Game validation failed: tags: You must provide more than 1
tag.
```

Async Valicators

Async validation comes into play when you need to fetch some remote data, or perform some other type of asynchronous task before persisting to the database. For this we can use an async validator. Let’s have a look at one. We’ll simulate asynchronous work with the setTimeout() function.

```
const gameSchema = new mongoose.Schema({
    title: {
        type: String,
        required: true,
        minlength: 4,
        maxlength: 200
    },
    publisher: String,
    tags: {
        type: [String],
        validate: {
            isAsync: true,
            validator: function (v, callback) {
                // Complete async task
                setTimeout(() => {
                    const result = v.length > 1;
                    callback(result);
                }, 2000);
            },
            message: 'You must provide more than 1 tag.'
        }
    },
    date: { type: Date, default: Date.now },
    onSale: Boolean,
    price: {
        type: Number,
        required: function () { return this.onSale }
    }
});
```

To enable asynchronous validation, all you need to do is add the **isAsync** property to the validate object and set it to `true`. Then you can do your async work whether that be fetching remote data, reading from the filesystem, or working with a database, and the validation will still work properly.

Mongoose Validation Examples Summary

In this tutorial on **Mongoose Validation** we learned that when defining a schema, you can set the type of a property to a SchemaType object. You use this object to define the validation requirements for the given property. We can add validation with code like this.

```
new mongoose.Schema({
    name: { type: String, required: true }
})
```

Validation logic is executed by Mongoose before a document can be saved to the database. It is also possible to trigger it manually by calling the validate() method. Some of the Built-in validators include:

- **Strings:** `minlength`, `maxlength`, `match`, `enum`
- **Numbers:** `min`, `max`
- **Dates:** `min`, `max`
- **All types:** `required`

To set up custom validation, you may set up the validate object and use a function in the validate property.

```
tags: [
    type: Array,
    validate: {
        validator: function (v) { return v && v.length > 0;
        message: 'A game should have at least 1 tag.'
    }
]
```

When talking to a database or a remote service to perform the validation, it is required that you use an async validator. You enable this with the `isAsync` property set to `true`.

```
validate: {
    isAsync: true
    validator: function(v, callback) {
        // Do the validation, when the result is ready, call the
        callback(isValid);
    }
}
```

Some other useful SchemaType properties include:

- **Strings:** `lowercase`, `uppercase`, `trim`
- **All types:** `get`, `set` (to define a custom getter/setter)

Happy Validating!

#javascript #nodejs

← Mongoose Crud Tutorial | Mongoose Relationships Tutorial →