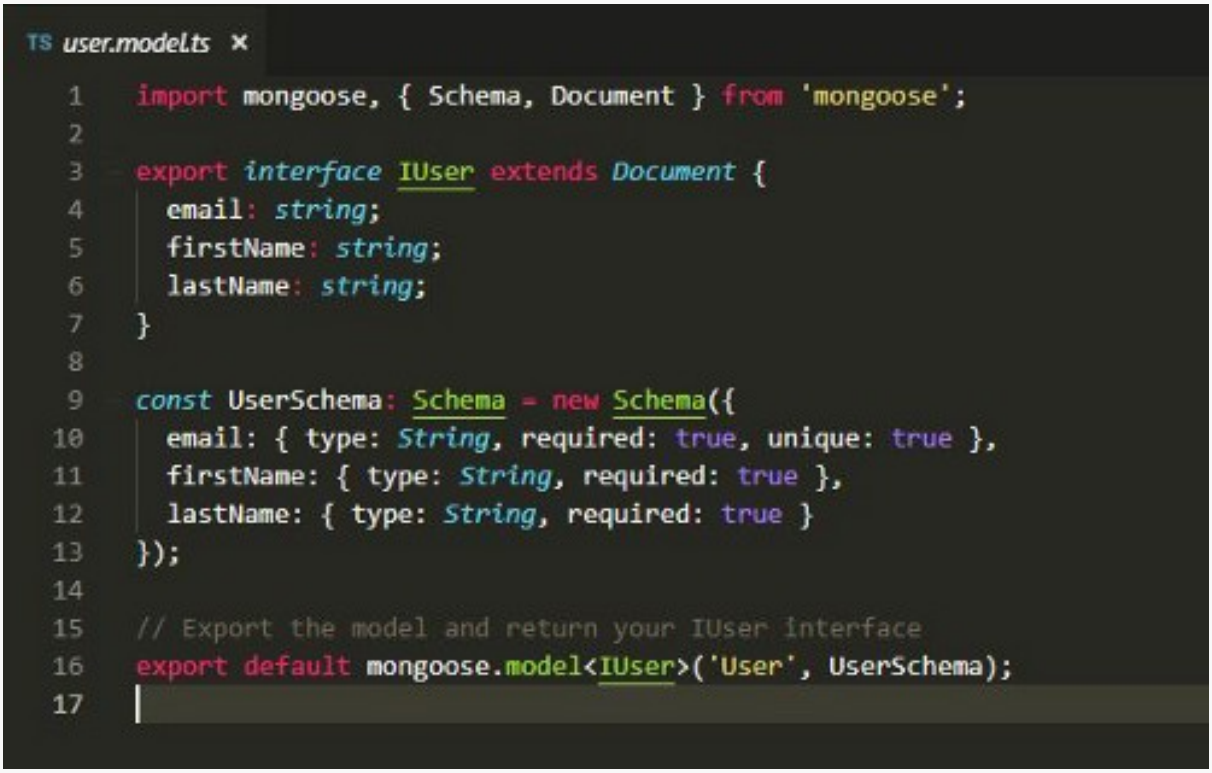
Tom Nagle

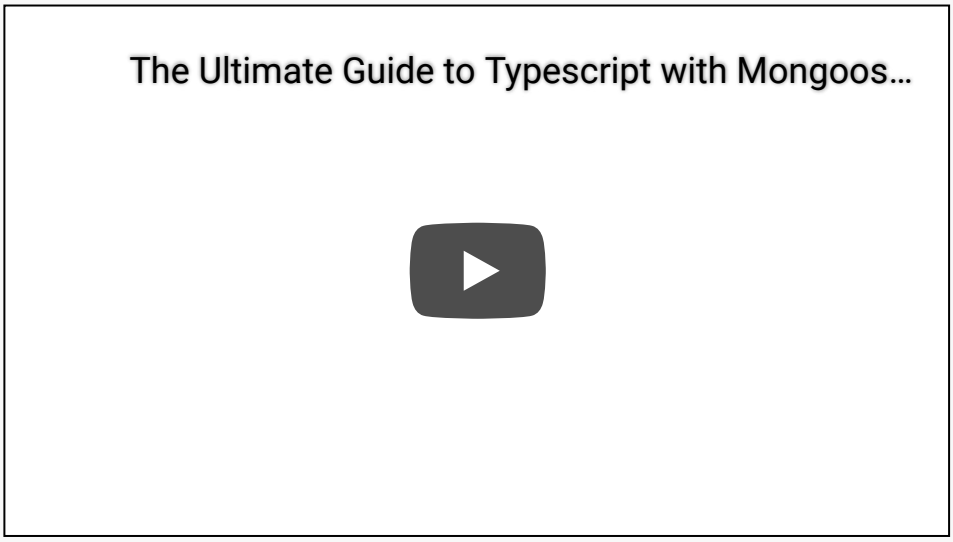
320 Followers About Follow

Strongly typed models with Mongoose and TypeScript

Tom Nagle Feb 11, 2019 · 4 min read ★



Mongoose provides a layer of abstraction over MongoDB that makes validation, casting & business logic boilerplate more enjoyable. Creating your models with a TypeScript



How to create strongly typed Mongoose models with TypeScript

There are two main ways to make your models strongly typed, Typegoose & and custom interfaces. This post will focus on custom interfaces. Typegoose adds too much abstraction to really know what's going on.

Assumptions

- TypeScript installed and working
- @types/mongoose installed & working
- Jest & mongodb-memory-server installed & working

You can see an example repository that has everything you need to get started here: https://github.com/tomanagle/Mongoose-TypeScript-example

Create your model as you usually would with Mongoose.

```
1  import mongoose, { Schema } from 'mongoose';
2
3  const UserSchema: Schema = new Schema({
4    email: { type: String, required: true, unique: true },
5    firstName: { type: String, required: true },
6    lastName: { type: String, required: true },
```

```
9  export default mongoose.model('User', UserSchema);
```

user.model.ts hosted with ♡ by GitHub view raw

We then need to create and export an interface for our user model. You can create this in the model file, or create an interface file that will store all your interfaces.

```
1  export interface IUser extends Document {
2    email: string;
3    firstName: string;
4    lastName: string;
5  }
```

user.interface.ts hosted with ♡ by GitHub view raw

Your newly created interface needs to extend Document, an interface that extends *MongooseDocument*, NodeJS.*EventEmitter* & *ModelProperties*. This will add the required functions and fields to your interface, such as save(), remove(), __v, ect...

The final step is to create & export your Mongoose model. The mongoose.model function takes two parameters, a name, and your schema, it will return your interface.

```
1  import mongoose, { Schema, Document } from 'mongoose';
2
3  export interface IUser extends Document {
4    email: string;
5    firstName: string;
6    lastName: string;
7  }
8
9  const UserSchema: Schema = new Schema({
10   email: { type: String, required: true, unique: true },
11   firstName: { type: String, required: true },
12   lastName: { type: String, required: true }
13 });
14
15 // Export the model and return your IUser interface
16 export default mongoose.model<IUser>('User', UserSchema);
```

user.model.ts hosted with ♡ by GitHub view raw

What about referencing other Mongoose models?

mongoose.Schema.Types.ObjectId, having @types/mongoose installed gives your application access to the ObjectId type. But what we really want to reference is the specific _id of another interface and because our interfaces extend Document, this is possible.

```
import mongoose, { Schema, Document } from 'mongoose';
import { IUser } from './user.model';

export interface IPet extends Document {
  name: string;
  owner: IUser['_id'];
}

const PetSchema: Schema = new Schema({
  name: { type: String, required: true },
  owner: { type: Schema.Types.ObjectId, required: true }
});

export default mongoose.model<IPet>('Pet', PetSchema);
```

pet.model.ts hosted with ♡ by GitHub view raw

Using your interfaces

TypeScript is not only great for catching errors and improving auto suggestions, but it helps you communicate with other developers. When you're creating a function, use your model's interface to tell other developers exactly what the input and output should be.

Instead of declaring everything in your interfaces as primitive types, start using the types straight out of your interface.

Don't:

```
interface ICreateUserInput {
  email: string;
  firstName: string;
  lastName: string;
}
```

ICreateUserInput.ts hosted with ♡ by GitHub view raw

```
interface ICreateUserInput {
  email: IUser['email'];
  firstName: IUser['firstName'];
  lastName: IUser['lastName'];
}
```

ICreateUserInput.ts hosted with ♡ by GitHub view raw

Testing

Testing your Mongoose models with Jest is relatively straight forward. You don't necessarily need to test that Mongoose itself works since that's already been tested.

There are two main reasons to write tests for your Mongoose models:

1. Check that validation errors are thrown as expected
2. Create and test your CRUD operations in an isolated environment, away from code that produces other side-effects

To test Mongoose models against a real instance of Mongoose, you will need to set up a test MongoDB environment and modify it to use Mongoose.

There are plenty of tutorials already that will show you how to test your Mongoose models with Jest, so I won't cover that.

However, there is one problem that you can get stuck on. Consider this: you're writing a test for a function that takes a user and the user comes straight from the database into your function. It's tempting to type the user input with the interface that you created in your Mongoose model.

This is extremely hard to test without using a global setup & teardown of a MongoDB connection. So, what do you do? You don't use the interface straight out of your Mongoose model, you create a new one.

Consider the following function:

```
export function getUserEmailById(
  user: IUser,
  users: IUser[]
): IUser['email'] {
```

getUserEmailById.ts hosted with ♡ by GitHub view raw

To test the function, we need to assemble an array of complete user objects. Because we extended mongoose.Document, this includes a lot of fields that are otherwise not needed in this function. All we need is the _id and email property. Let's refactor the function to be more explicit about the function's parameters.

```
export interface InputUser {
  _id: IUser['_id'];
  email: IUser['email'];
}

export function getUserEmailById(
  user: InputUser,
  users: InputUser[]
): InputUser['email'] {
  return users.filter((item: InputUser) => item._id === user._id)[0].email;
}
```

getUserEmailById-with-interface.ts hosted with ♡ by GitHub view raw

Testing the function becomes a lot easier and the required inputs are much clearer.

```
import mongoose from 'mongoose';
import Faker from 'faker';
import { getUserEmailById, InputUser } from '../user.controller';

describe('Get user email by _id', () => {
  it('should return the right email', () => {
    /*
     * Compile an array of users that only have the required fields
     */
    const users: InputUser[] = Array(10)
      .fill(undefined)
      .map(() => {
        return {
          email: Faker.internet.email(),
          _id: mongoose.Types.ObjectId()
        };
      });

    expect.assertions(users.length);
```

```
23       }
24     });
25   });
```

getUserEmailById.test.ts hosted with ♡ by GitHub view raw

The example repository can be found here:
https://github.com/tomanagle/Mongoose-TypeScript-example

Read the follow-up to this article where I take Mongoose types to the service layer, but not beyond: https://tomanagle.medium.com/even-stronger-typed-models-with-mongoose-and-typescript-916f0a9e5c3c

Strongly Typed Mongoose Models with Typegoose & TypeScript:
https://tomanagle.medium.com/strongly-typed-mongoose-models-with-typegoose-typescript-9290d4ed00ed

. . .

Edit: This article has become a lot more popular than I expected. I do come in and update it every now and again. If you think there is something to be added on the topic, please let me know so we can discuss it. — 10th January 2021

. . .

🌎 Follow me here:

YouTube: https://www.youtube.com/tomdoestech
Facebook: https://www.facebook.com/tomdoestech
Instagram: https://www.instagram.com/tomdoestech
Twitter: https://twitter.com/tomdoesntdotech
TikTok: https://www.tiktok.com/@tomdoestech

☕ Buy me a coffee: https://www.buymeacoffee.com/tomn