Strongly Typed Mongoose Models with Typegoose & TypeScript

Tom Nagle Follow
Feb 28 · 3 min read ★

In my previous article, we looked at how we can integrate Mongoose and TypeScript to create strongly typed models manually. In this article, you will learn how to create Mongoose models & TypeScript types simultaneously with Typegoose.

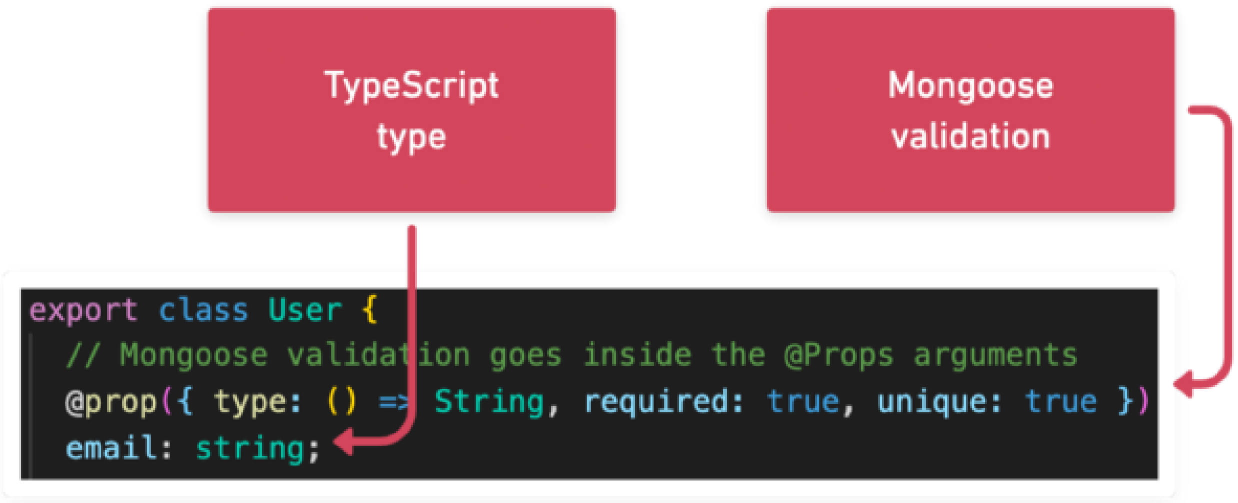
See final repository: https://github.com/tomanagle/Typegoose-example

What's the problem?

Often, the data shapes used throughout our applications start at the model level. We then replicate these data shapes with TypeScript so we can represent the data through the application. Typegoose aims to eliminate the need to create our data shapes in both Mongoose & TypeScript and keep the shapes synchronized throughout the development lifecycle.

How does Typegoose solve the problem?

Typegoose solves the problem by using classes and decorators to output both the Mongoose model and the TypeScript interface. Each property in the class has a TypeScript type and a `@props` decorator. The decorator is what will build the Mongoose model, while the TypeScript property will be used to build an interface. Finally, the class can be used to build a model with the aptly named `getModelForClass` function



```
export class User {
  // Mongoose validation goes inside the @Props arguments
  @prop({ type: () => String, required: true, unique: true })
  email: string;
```

. . .

Before we start with Typegoose, let's have a quick look at where we came from. In the last guide, we created two models.

A user model:

```
import mongoose, { Schema, Document } from 'mongoose';

export enum Gender {
  male = 'male',
  female = 'female',
  undisclosed = 'undisclosed'
}

export interface Address extends Document {
  street: string;
  city: string;
  postCode: string;
}

export interface IUser extends Document {
  email: string;
  firstName: string;
  lastName: string;
  gender?: Gender;
  address?: Address;
}

const UserSchema: Schema = new Schema({
  email: { type: String, required: true, unique: true },
  firstName: { type: String, required: true },
  lastName: { type: String, required: true },
  // Gets the Mongoose enum from the TypeScript enum
  gender: { type: String, enum: Object.values(Gender) },
  address: {
    street: { type: String },
    city: { type: String },
    postCode: { type: String }
  }
});

// Export the model and return your IUser interface
export default mongoose.model<IUser>('User', UserSchema);
```

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

And a pets model:

```
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);
```

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

These two models demonstrate several key Mongoose concepts that we are going to replicate with Typegoose:

- Schema validation such as required and unique
- Referencing another schema
- Embedding a subdocument

Now, let’s looks at how we would rewrite the above schemas with Typegoose.

The user model now looks like this:

```typescript
import { prop, getModelForClass } from "@typegoose/typegoose";

enum Gender {
  male = "male",
  female = "female",
  undisclosed = "undisclosed",
}

class Address {
  @prop({ type: () => String })
  street: string;

  @prop({ type: () => String })
  city: string;

  @prop({ type: () => String })
  postCode: string;
}

export class User {
  // Mongoose validation goes inside the @Props arguments
  @prop({ type: () => String, required: true, unique: true })
  email: string;

  @prop({ type: () => String, required: true })
  firstName: string;

  @prop({ type: () => String, required: true })
  lastName: string;

  // Enum of values male, female or undisclosed
  @prop({ type: () => String, enum: Object.values(Gender) })
  gender: string;

  // Embedded sub doccument
  @prop({ type: () => Address })
  address: Address;
}

const UserModel = getModelForClass(User);

export default UserModel;
```

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

And the pet model now looks like this:

```typescript
import { prop, getModelForClass, Ref } from "@typegoose/typegoose";
import { User } from "./user.model";

export class Pet {
  // Take
  @prop()
  public name: string;

  // A reference to another document
  @prop({ type: () => User })
  public owner: Ref<User>;
}

const PetModel = getModelForClass(Pet);

export default PetModel;
```

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

As you can see, each schema is defined by a class with TypeScript types. Then, an `@props` decorator is layered on top of each type to define the Mongoose schema validation.

Virtuals

Mongoose virtual types are data types that are computed on the output. The result does not get saved in the document. They are perfect for the concatenation of several fields. I have also used them to count the number of subdocuments in an array to save large lists of data being sent to a client just so a count can be shown.

```typescript
import { prop, getModelForClass, Ref } from "@typegoose/typegoose";
import { User } from "./user.model";

export class Pet {
  @prop({ type: () => String })
  public name: string;

  @prop({ type: () => String })
  public type: string;

  // A reference to another document
  @prop({ type: () => User })
  public owner: Ref<User>;
  // A virtual type
  public get greeting() {
    return `Hi! I am ${this.name} the ${this.type}.`;
  }
}

const PetModel = getModelForClass(Pet);

export default PetModel;
```

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

Use the virtual field will appear on the document:

```
export async function getPetGreeting(input: FilterQuery<Pet>): Promise<string> {
  const pet = await PetModel.findOne(input);

  return pet.greeting;
}
```

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

Virtuals Typegoose documentation:
https://typegoose.github.io/typegoose/docs/api/virtuals

Virtuals Mongoose documentation:
https://mongoosejs.com/docs/guide.html#virtuals

Indexing is an important part of any database and MongoDB is no exception. Fortunately, Typegoose supports indexing with an `@index` decorator.

```
import {
  prop,
  getModelForClass,
  Ref,
  modelOptions,
  index,
} from "@typegoose/typegoose";
import { User } from "./user.model";

@modelOptions({ schemaOptions: { collection: "pet", timestamps: true } })
@index({ name: 1 })
export class Pet {
  @prop({ type: () => String })
  public name: string;

  @prop({ type: () => String })
  public type: string;

  // A reference to another document
  @prop({ type: () => User })
  public owner: Ref<User>;

  // A virtual type
  public get greeting() {
    return `Hi! I am ${this.name} the ${this.type}.`;
  }
}

const PetModel = getModelForClass(Pet);

export default PetModel;
```

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

Typegoose index documentation: https://typegoose.github.io/typegoose/docs/api/decorators/indexes

Mongoose indexes documentation: https://mongoosejs.com/docs/guide.html#indexes

Final thoughts

Typegoose is a good alternative to declaring your TypeScript types and Mongoose models independently. The repository looks as though it is well maintained, with the last commit being 13 days ago as of writing this article and there is a low amount of bugs reported in the issues section.

However, when using libraries to define your application's object structure, it's always important to figure out what blockers there may be ahead. You do not want to not be able to create an object in a certain way because a helper library you are using won't support it.

Final repository: https://github.com/tomanagle/Typegoose-example

Typegoose: https://github.com/typegoose/typegoose

. . .

🌎 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

JavaScript Typescript Mongoose Mongodb Nodejs