TypeScript Support

Mongoose introduced officially supported TypeScript bindings in v5.11.0. Mongoose's `index.d.ts` file supports a wide variety of syntaxes and strives to be compatible with `@types/mongoose` where possible. This guide describes Mongoose's recommended approach to working with Mongoose in TypeScript.

Creating Your First Document

To get started with Mongoose in TypeScript, you need to:

1. Create an interface representing a document in MongoDB.
2. Create a Schema corresponding to the document interface.
3. Create a Model.
4. Connect to MongoDB.

```
import { Schema, model, connect } from 'mongoose';

// 1. Create an interface representing a document in MongoDB.
interface User {
  name: string;
  email: string;
  avatar?: string;
}

// 2. Create a Schema corresponding to the document interface.
const schema = new Schema<User>({
  name: { type: String, required: true },
  email: { type: String, required: true },
  avatar: String
});

// 3. Create a Model.
const UserModel = model<User>('User', schema);

run().catch(err => console.log(err));

async function run(): Promise<void> {
  // 4. Connect to MongoDB
  await connect('mongodb://localhost:27017/test');

  const doc = new UserModel({
    name: 'Bill',
    email: 'bill@initech.com',
    avatar: 'https://i.imgur.com/dM7Thhn.png'
  });
  await doc.save();

  console.log(doc.email); // 'bill@initech.com'
}
```

You as the developer are responsible for ensuring that your document interface lines up with your Mongoose schema. For example, Mongoose won't report an error if `email` is `required` in your Mongoose schema but optional in your document interface.

The `UserModel()` constructor returns an instance of `HydratedDocument<User>`. `User` is a *document interface*, it represents the raw object structure that `User` objects look like in MongoDB. `HydratedDocument<User>` represents a hydrated Mongoose document, with methods, virtuals, and other Mongoose-specific features.

```
import { HydratedDocument } from 'mongoose';

const doc: HydratedDocument<User> = new UserModel({
  name: 'Bill',
  email: 'bill@initech.com',
  avatar: 'https://i.imgur.com/dM7Thhn.png'
});
```

Using `extends Document`

Alternatively, your document interface can extend Mongoose's `Document` class. Many Mongoose TypeScript codebases use the below approach.

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

interface User extends Document {
  name: string;
  email: string;
  avatar?: string;
}
```

This approach works, but we recommend your document interface *not* extend `Document`. Using `extends Document` makes it difficult for Mongoose to infer which properties are present on query filters, lean documents, and other cases.

We recommend your document interface contain the properties defined in your schema and line up with what your documents look like in MongoDB. Although you can add instance methods to your document interface, we do not recommend doing so.

Using Custom Bindings

If Mongoose's built-in `index.d.ts` file does not work for you, you can remove it in a postinstall script in your `package.json` as shown below. However, before you do, please open an issue on Mongoose's GitHub page and describe the issue you're experiencing.

```
{
  "postinstall": "rm ./node_modules/mongoose/index.d.ts"
}
```

Next Up

Now that you've seen the basics of how to use Mongoose in TypeScript, let's take a look at statics in TypeScript.