Mongoose, DotEnv

Paket

- Det finns olika paket som hjälper oss på olika sätt med kopplingen till databasen
- För en vanlig koppling används ofta `monk`
- För att få lite mer hjälp med modeller, verifiering av data osv används ofta `mongoose`

Mongoose

- Installera mongoose:

```
npm install mongoose
```

- Hämta in paketet:

```
const mongoose = require('mongoose')
mongoose.connect('mongodb://localhost:27017/test', {
  useNewUrlParser: true,
  useUnifiedTopology: true
});
```

- Dokumentation
  - https://mongoosejs.com/docs/

Mongoose

- Below are some of the options that are important for tuning Mongoose.
  - **useNewUrlParser** - The underlying MongoDB driver has deprecated their current connection string parser. Because this is a major change, they added the useNewUrlParser flag to allow users to fall back to the old parser if they find a bug in the new parser. You should set useNewUrlParser: true unless that prevents you from connecting. Note that if you specify useNewUrlParser: true, you must specify a port in your connection string, like mongodb://localhost:27017/dbname. The new url parser does not support connection strings that do not have a port, like mongodb://localhost/dbname.
  - **useUnifiedTopology** - False by default. Set to true to opt in to using the MongoDB driver's new connection management engine. You should set this option to true, except for the unlikely case that it prevents you from maintaining a stable connection.

Mongoose

What databases can I use?

- There are many popular options, including PostgreSQL, MySQL, Redis, SQLite, and MongoDB.
- Things to consider:
  - time-to-productivity/learning curve
  - performance
  - ease of replication/backup
  - cost
  - community support
  - etc
- While there is no single "best" database, almost any of the popular solutions should be more than acceptable for a small-to-medium-sized site.

Mongoose

What is the best way to interact with a database?

- There are two common approaches for interacting with a database:
  - Using the databases' native query language (e.g. *SQL*)
  - Using an *Object Data Model* (`ODM`) or an *Object Relational Model* (`ORM`).
  - An `ODM/ORM` represents the website's data as JavaScript objects, which are then mapped to the underlying database.
  - Some `ORM`s are tied to a specific database, while others provide a database-agnostic backend.

Mongoose

What ORM/ODM should I use?

- Some popular options:
  - **Mongoose**: Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment.
  - **Waterline**: An ORM extracted from the Express-based Sails web framework. It provides a uniform API for accessing numerous different databases, including Redis, MySQL, LDAP, MongoDB, and Postgres.
  - **Bookshelf**: Features both promise-based and traditional callback interfaces, providing transaction support, eager/nested-eager relation loading, polymorphic associations, and support for one-to-one, one-to-many, and many-to-many relations. Works with PostgreSQL, MySQL, and SQLite3.
  - **Objection**: Makes it as easy as possible to use the full power of SQL and the underlying database engine (supports SQLite3, Postgres, and MySQL).

Mongoose

Connect

```
//Import the mongoose module
var mongoose = require('mongoose');

//Set up default mongoose connection
var mongoDB = 'mongodb://127.0.0.1/my_database';
mongoose.connect(mongoDB, {useNewUrlParser: true, useUnifiedTopology: true});

//Get the default connection
var db = mongoose.connection;

//Bind connection to error event (to get notification of connection errors)
db.on('error', console.error.bind(console, 'MongoDB connection error:'));
```

Mongoose

Defining and creating models

- Mongoose hjälper oss att sätta upp modeller som gör det lättare att hålla våra dokument konsekventa.
- Vi sätter upp modeller med interfacet Schema.
- Schemana ”kompileras” sedan till modeller med `mongoose.model();`
- Varje modell mappar till en collection i MongoDB. Dokumenten kommer att innehålla de fält/typer som vi har definierat i modellen.

Mongoose
Defining schemas

```
//Require Mongoose
var mongoose = require('mongoose');

//Define a schema
var Schema = mongoose.Schema;

var SomeModelSchema = new Schema({
  a_string: String,
  a_date: Date
});

// Compile model from schema
var SomeModel = mongoose.model('SomeModel', SomeModelSchema );
```

Mongoose

Schema types (fields)

```
var schema = new Schema({
  name: String,
  binary: Buffer,
  living: Boolean,
  updated: { type: Date, default: Date.now() },
  age: { type: Number, min: 18, max: 65, required: true },
  mixed: Schema.Types.Mixed,
  _someId: Schema.Types.ObjectId,
  array: [],
  ofString: [String], // You can also have an array of each of the other types too.
  nested: { stuff: { type: String, lowercase: true, trim: true } }
})
```

Mongoose

Schema types (fields)

- `ObjectId`: Represents specific instances of a model in the database. For example, a book might use this to represent its author object. This will actually contain the unique ID (_id) for the specified object. We can use the populate() method to pull in the associated information when needed.

- `Mixed`: An arbitrary schema type.

- `[]`: An array of items. You can perform JavaScript array operations on these models (`push`, `pop`, `unshift`, etc.). The examples above show an array of objects without a specified type and an array of String objects, but you can have an array of any type of object.

Mongoose

Schema types (fields) - declaring a field

- Field name and type as a key-value pair (i.e. as done with fields `name`, `binary` and `living`).
- Field name followed by an object defining the type, and any other options for the field. Options include things like:
  - default values.
  - built-in validators (e.g. max/min values) and custom validation functions.
  - Whether the field is required
  - Whether String fields should automatically be set to lowercase, uppercase, or trimmed (e.g. `{ type: String, lowercase: true, trim: true }`)

Mongoose
Validation

- Mongoose provides built-in and custom validators, and synchronous and asynchronous validators. It allows you to specify both the acceptable range of values and the error message for validation failure in all cases.
- The built-in validators include:
  - All *SchemaTypes* have the built-in required validator. This is used to specify whether the field must be supplied in order to save a document.
  - *Numbers* have `min` and `max` validators.
  - *Strings* have:
    - `enum`: specifies the set of allowed values for the field.
    - `match`: specifies a regular expression that the string must match.
    - `maxlength` and `minlength` for the string.

Mongoose
Validation

```
var breakfastSchema = new Schema({
  eggs: {
    type: Number,
    min: [6, 'Too few eggs'],
    max: 12,
    required: [true, 'Why no eggs?']
  },
  drink: {
    type: String,
    enum: ['Coffee', 'Tea', 'Water',]
  }
});
```

Mongoose

Virtual properties

- Virtual properties are document properties that you can get and set but that do not get persisted to MongoDB. The getters are useful for formatting or combining fields, while setters are useful for de-composing a single value into multiple values for storage.

Mongoose

Methods and query helpers

- A schema can also have instance methods, static methods, and query helpers.
  - The instance and static methods are similar, but with the obvious difference that an instance method is associated with a particular record and has access to the current object.
  - Query helpers allow you to extend mongoose's chainable query builder API (for example, allowing you to add a query "byName" in addition to the `find()`, `findOne()` and `findById()` methods).

Mongoose

Creating and modifying documents

- To create a record you can define an instance of the model and then call `save()`.

```
// Create an instance of model SomeModel
var awesome_instance = new SomeModel({ name: 'awesome' });

// Save the new model instance, passing a callback
awesome_instance.save(function (err) {
  if (err) return handleError(err);
  // saved!
});
```

Mongoose

Creating and modifying documents

- Creation of records (along with updates, deletes, and queries) are asynchronous operations — you supply a callback that is called when the operation completes.
- The API uses the error-first argument convention, so the first argument for the callback will always be an error value (or null).
- If the API returns some result, this will be provided as the second argument.

Mongoose

Creating and modifying documents

- You can also use `create()` to define the model instance at the same time as you save it. The callback will return an error for the first argument and the newly-created model instance for the second argument.

```
SomeModel.create({ name: 'also_awesome' }, function (err, awesome_instance) {
  if (err) return handleError(err);
  // saved!
});
```

Mongoose

Creating and modifying documents

- You can access the fields in this new record using the dot syntax, and change the values. You have to call `save()` or `update()` to store modified values back to the database.

```
// Access model field values using dot notation
console.log(awesome_instance.name); //should log 'also_awesome'

// Change record by modifying the fields, then calling save().
awesome_instance.name = "New cool name";
awesome_instance.save(function (err) {
   if (err) return handleError(err); // saved!
});
```

Mongoose

Searching for records

- You can search for records using query methods, specifying the query conditions as a JSON document. The code fragment below shows how you might find all athletes in a database that play tennis, returning just the fields for athlete name and age. Here we just specify one matching field (sport) but you can add more criteria, specify regular expression criteria, or remove the conditions altogether to return all athletes.

```
var Athlete = mongoose.model('Athlete', yourSchema);

// find all athletes who play tennis, selecting the 'name' and 'age' fields
Athlete.find({ 'sport': 'Tennis' }, 'name age', function (err, athletes) {
  if (err) return handleError(err);
  // 'athletes' contains the list of athletes that match the criteria.
})
```

Mongoose

Searching for records

- Note: All callbacks in Mongoose use the pattern `callback(error, result)`. If an error occurs executing the query, the `error` parameter will contain an error document and result will be `null`. If the query is successful, the error parameter will be `null`, and the result will be populated with the results of the query.
- Note: It is important to remember that *not finding any results is not an error for a search* —but it may be a fail-case in the context of your application. If your application expects a search to find a value you can either check the result in the callback (`results==null`) or daisy chain the `orFail()` method on the query.

Mongoose

Searching for records

- If you don't specify a callback then the API will return a variable of type `Query`. You can use this query object to build up your query and then execute it (with a callback) later using the `exec()` method.

```
// find all athletes that play tennis
var query = Athlete.find({ 'sport': 'Tennis' });

// selecting the 'name' and 'age' fields
query.select('name age');

// limit our results to 5 items
query.limit(5);

// sort by age
query.sort({ age: -1 });

// execute the query at a later time
query.exec(function (err, athletes) {
  if (err) return handleError(err);
  // athletes contains an ordered list of 5 athletes who play Tennis
})
```

Mongoose

Searching for records - alternative

```
Athlete
  .find()
  .where('sport').equals('Tennis')
  .where('age').gt(17).lt(50)  //Additional where query
  .limit(5)
  .sort({ age: -1 })
  .select('name age')
  .exec(callback); // where callback is the name of our callback function.
```

Mongoose

Searching for records

- The `find()` method gets all matching records, but often you just want to get one match. The following methods query for a single record:
  - `findById()`: Finds the document with the specified id (every document has a unique id).
  - `findOne()`: Finds a single document that matches the specified criteria.
  - `findByIdAndRemove()`, `findByIdAndUpdate()`, `findOneAndRemove()`, `findOneAndUpdate()`: Finds a single document by id or criteria and either updates or removes it. These are useful convenience functions for updating and removing records.
  - There is also a `count()` method that you can use to get the number of items that match conditions. This is useful if you want to perform a count without actually fetching the records.

Mongoose

Working with related documents — population

- You can create references from one document/model instance to another using the `ObjectId` schema field, or from one document to many using an array of `ObjectIds`. The field stores the id of the related model. If you need the actual content of the associated document, you can use the `populate()` method in a query to replace the id with the actual data.

Mongoose

Working with related documents — population

```
var mongoose = require('mongoose')
  , Schema = mongoose.Schema

var authorSchema = Schema({
  name    : String,
  stories : [{ type: Schema.Types.ObjectId, ref: 'Story' }]
});

var storySchema = Schema({
  author : { type: Schema.Types.ObjectId, ref: 'Author' },
  title    : String
});

var Story  = mongoose.model('Story', storySchema);
var Author = mongoose.model('Author', authorSchema);
```

- Each author can have multiple stories, which we represent as an array of ObjectId.
- Each story can have a single author. The "ref" (highlighted in bold below) tells the schema which model can be assigned to this field.

Mongoose

Working with related documents — population

```
var bob = new Author({ name: 'Bob Smith' });

bob.save(function (err) {
  if (err) return handleError(err);

  //Bob now exists, so lets create a story
  var story = new Story({
    title: "Bob goes sledding",
    author: bob._id    // assign the _id from the our author Bob. This
ID is created by default!
  });

  story.save(function (err) {
    if (err) return handleError(err);
    // Bob now has his story
  });
});
```

- We can save our references to the related document by assigning the `_id` value.
- Below we create an author, then a story, and assign the author id to our story's author field.

Mongoose

Working with related documents — population

```
Story
  .findOne({ title: 'Bob goes sledding' })
  .populate('author') //This populates the author id
with actual author information!
  .exec(function (err, story) {
    if (err) return handleError(err);
    console.log('The author is %s', story.author.name);
    // prints "The author is Bob Smith"
  });
```

- Our story document now has an author referenced by the author document's ID.
- In order to get the author information in the story results we use `populate()`, as shown below.

Mongoose

Working with related documents — population

- Astute readers will have noted that we added an author to our story, but we didn't do anything to add our story to our author's `stories` array.
- How then can we get all stories by a particular author?
  - One way would be to add our story to the stories array, but this would result in us having two places where the information relating authors and stories needs to be maintained.
  - A better way is to get the `_id` of our *author*, then use `find()` to search for this in the author field across all stories.

```
Story
.find({ author : bob._id })
.exec(function (err, stories) {
  if (err) return handleError(err);
  // returns all stories that have Bob's id as their author.
});
```

Mongoose

One schema/model per file

- While you can create schemas and models using any file structure you like, we highly recommend defining each model schema in its own module (file), then exporting the method to create the model.

```
// File: ./models/somemodel.js

//Require Mongoose
var mongoose = require('mongoose');

//Define a schema
var Schema = mongoose.Schema;

var SomeModelSchema = new Schema({
  a_string: String,
  a_date: Date,
});

//Export function to create "SomeModel" model class
module.exports = mongoose.model('SomeModel', SomeModelSchema );
```

Mongoose

One schema/model per file

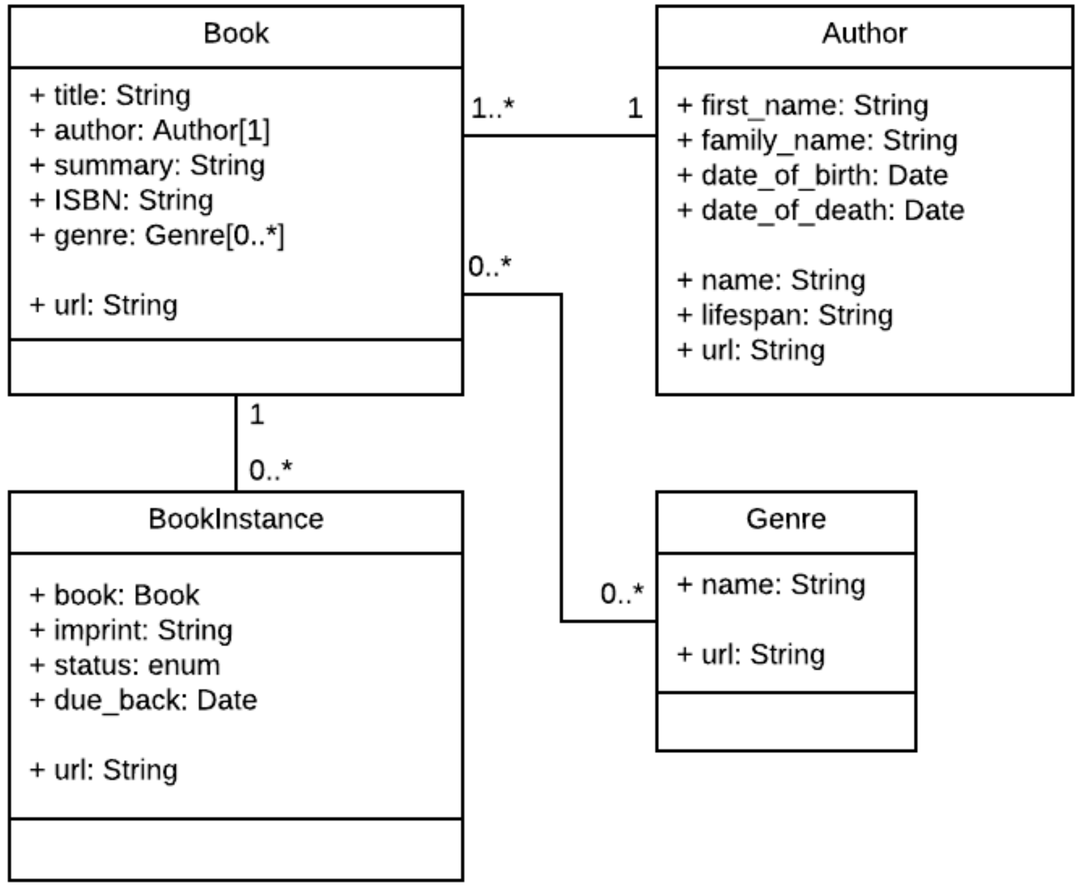
- You can then require and use the model immediately in other files.

```
//Create a SomeModel model just by requiring the module
var SomeModel = require('../models/somemodel')

// Use the SomeModel object (model) to find all SomeModel records
SomeModel.find(callback_function);
```

Mongoose

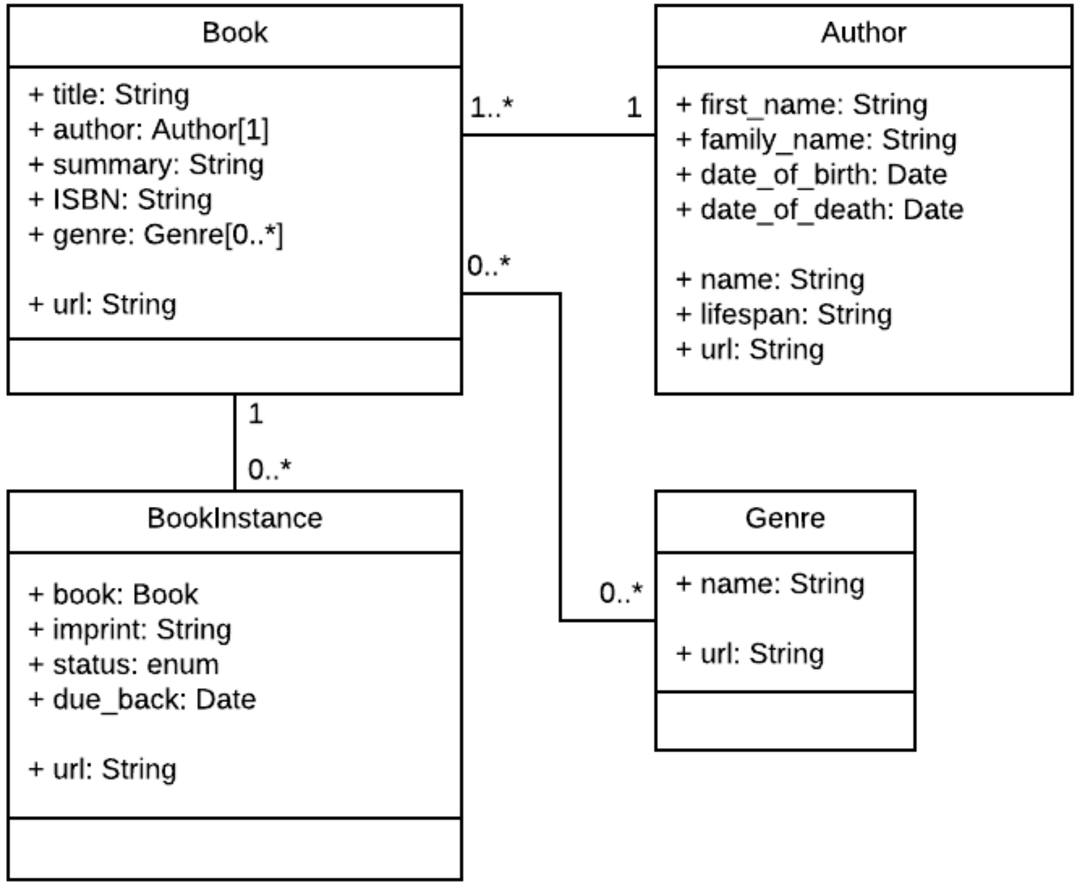
- Vi ska skapa en bok-databas, men först en del teori.



Code along

Bokdatabas / bibliotek

- Vi ska göra en bokdatabas. Mycket kommer ifrån denna tutorial.
  - **https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose**
- Först ska vi dock titta på en liten detalj.



Ansluta till databas

- Vad finns det för problem med att pusha upp denna kod till ett publikt repo?

- ```
  mongoose.connect('mongodb://
  myDBReader:D1fficultP%40ssw0rd@mongodb0.example.com:27017/?authSource=admin', {
    useNewUrlParser: true,
    useUnifiedTopology: true
  });
  ```
  ```

Lokala inställningar

- Man kan också tänka sig att man har olika anslutningsuppgifter i olika miljöer, t ex att man har olika adresser till databasen, olika användarnamn, olika lösenord i sin lokala miljö, i en gemensam testmiljö, eller i en produktionsmiljö.

- ```
  mongoose.connect('mongodb://
  myDBReader:D1fficultP%40ssw0rd@mongodb0.example.com:27017/?authSource=admin', {
    useNewUrlParser: true,
    useUnifiedTopology: true
  });
  ```
  ```

Lokala inställningar
.ENV

- Ett sätt att lösa detta är att ha en lokal fil som innehåller dina anslutningsuppgifter.
- Filen ska heta `.env` och det ska finnas en `.env` för varje miljö.
- Filen är en vanlig text-fil som innehåller key-value-pairs.

```
# API
API_TOKEN=myUniqueApiToken
# Database
DATABASE_NAME=myDatabaseName
```

- Lägg till filen i `.gitignore`.
- Det kan vara en idé att även skapa en `.env_template` eller liknande som innehåller alla värden som används i applikationen för att underlätta för andra utvecklare.

Lokala inställningar

How to use custom environment variables in Node

1. Create an `.env` file. The file should be placed in the root of your project
2. Install the `dotenv` library: `npm install dotenv`
3. Require `dotenv` as early as possible (e.g. in `app.js`):
   ```
   require('dotenv').config({path: __dirname + '/.env'})
   ```
4. Wherever you need to use environment variables (e.g. in GitLab, in Jenkins, in Heroku, …) you need to add your environment variables. The way depends on the platform but it is usually easy to do.
5. *Optional:* create a function which runs at startup of your server. It should check whether all the required environment variables are set and throw an error otherwise.

Code along

Bibliotek

- Tillbaka till projektet…

Eftermiddagsprojekt

Blogg

- (Ni kan även fortsätta med / skriva om receptsamlingen om ni föredrar det.)
- En blogg innehåller inlägg. Man kan tänka sig olika lösningar, men säg att man visar en eller ett par ”huvudinlägg” (senaste inlägget/inläggen) och att man visar länkar till tio ytterligare inlägg. När man klickar på ett inlägg kommer man till detaljsidan för inlägget.
  - Tänkbara fält för ett inlägg: title, content, author(s), categories, tags
  - Man skulle kunna tänka sig att lägga in bilder. Än så länge vet vi inte hur man laddar upp bilder, men vi skulle kunna lägga bilder i en mapp (`/public/images/kitten.jpg`) och spara sökvägen till bilderna i en egenskap i dokumentet. (`image: '/public/images/kitten.jpg'`)
  - Funktioner man skulle kunna vilja ha: möjlighet att CRUD:a inlägg, skriva kommentarer till inlägg, kunna lista inlägg på kategori/tagg, kunna söka efter inlägg, kunna lista inlägg efter författare.