CS5610: Web Development

Introduction to Mongoose & MongoDB

Joydeep Mitra

Pre-class Activity

- Fork the repository https://github.com/CSE-316-Software-Development/learn-express-mongoose
- Follow instructions in README.md to setup MongoDB and the project.
- Create a new branch with today's date.
- Submit all activities in this branch.

Motivation

- Web services are generally *stateless.* Why?
  - HTTP is a stateless protocol.
  - enable easy horizontal scaling.
- Hence, state management is delegated to a database.
  - Provides mechanisms to store, manage, and access data.
- A database can be relational or non-relational.
- We are working with a non-relational database (MongoDB) here!

Interacting with a Database

- There are two ways to interact with the database from a web server:
  - Use the database's **native query language** (e.g., *mongodb query language*).
    - *Pro*: Queries run faster
    - *Con*: Query results need to be serialized in the server.
  - Use an **Object Data/Relational Model** (ODM/ORM) to map the server-side language's data structures to the underlying database.
    - e.g., map JavaScript objects to MongoDB documents.
    - *Pro*: Serialization/Deserialization is automatic; no need for the programmer to manage it.
    - *Con*: queries need to be translated to DB engine's native language.

Mongoose

- An *ODM tool for MongoDB* designed to work in an asynchronous environment.
- It has a rich set of APIs that can be used:
  - Define models with flexible schemas.
  - Define validators to verify the correctness of the data in the models.
  - Insert, query, update, and delete the models asynchronously.
  - the document storage and query system looks like JSON.

Defining Models

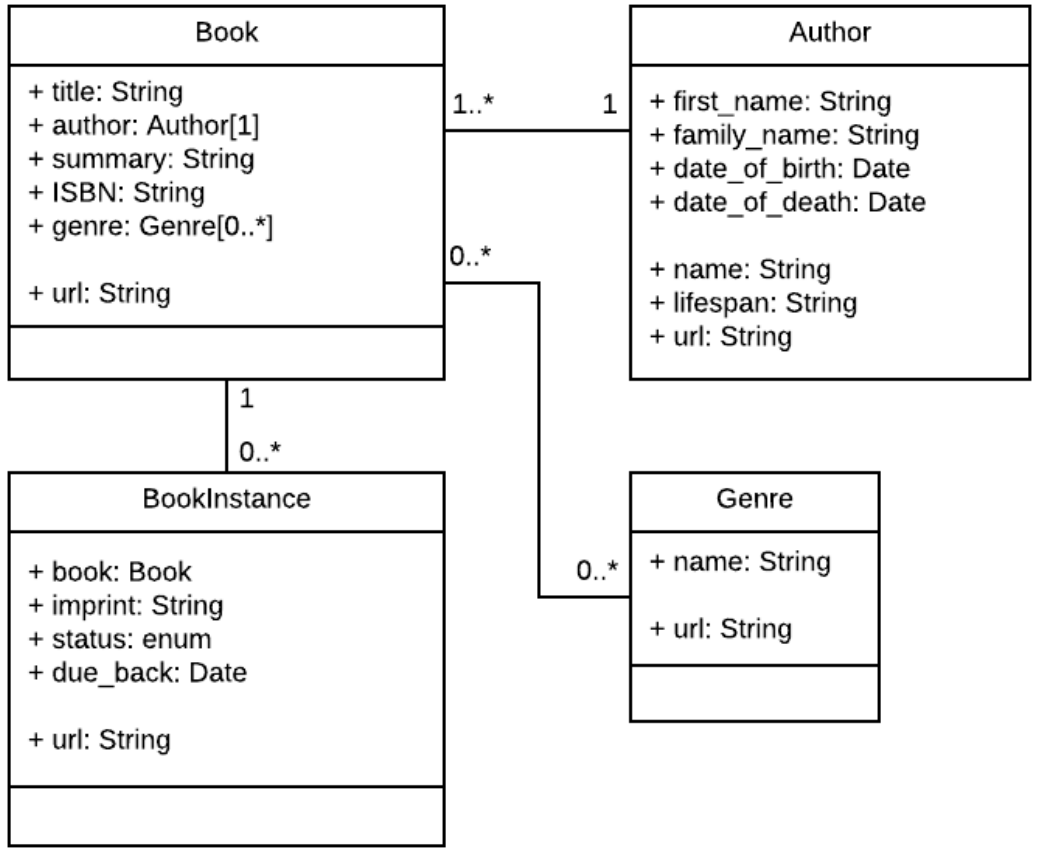
- A *model* is used to define a *document* -- an instance of a data object stored in a MongoDB collection.
  - fields and their validation constraints and default values.
  - helper methods to work with fields.
  - virtual properties to define fields not physically stored in the database.
- The `Schema` interface is used to define a model.
- Schemas are ”compiled” to models using `mongoose.model()`.
  - A ”compiled” model can now be used to interact with the underlying documents in a database collection.

An Example

- Suppose we are developing web services for a library system.
- The data will be stored in a MongoDB database.
- Services will be stateless and will interact with the MongoDB database through the Mongoose ODM.
- Assume the following data model.
  - Objects: books, authors, book instances, and genres.
  - A book has exactly ONE author.
  - A book can belong to 0 or more genres.
  - A book has 0 or more book instances or copies.

The Library Data Model

- Review the objects and their relationships.
- Consider the examples in *models/author.js* and *models/bookInstance.js*



For You to Do

- Define models in *models/book.js* and *models/genres.js* based on the Library data model shown in the UML class diagram.
- Add a virtual property in *models/author.js* called *lifespan* that will return a string of the form <birthyear – deathYear> for an author.
  - E.g., “1992 – 2014” or if not death year then “1992 – “

Referencing vs. Embedding

- What is **embedding**?
  - We have two documents D1 and D2; D2 is embedded as a *subdocument* inside D1.
- What is **referencing**
  - We have two documents D1 and D2; D1 has a *reference* to D1.

Referencing vs. Embedding

- Prefer embedding if
  - Embedded document is accessed in the context of the parent document.
  - There is a need to update related data in a single atomic write operation.
- Prefer referencing if
  - Referenced document is shared across several parent documents.
  - Referenced document needs to be queried independently.

Referencing vs. Embedding: Atomic Updates

- Consider a scenario where a user A transfers X amount from account A1 to account A2.
- With an embedded schema the operations will be atomic!

```
// Schema for account
const accountSchema = new mongoose.Schema({
  accountNumber: {
    type: String,
    required: true
  },
  balance: {
    type: Number,
    required: true
  }
});

// Schema for user
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  accounts: [accountSchema] // Embedded document for accounts
});
```

Referencing vs. Embedding: Atomic Updates

- Consider a scenario where a user A transfers X amount from account A1 to account A2.
- With a referenced schema the operations will NOT be atomic!

```
// Schema for account
const accountSchema = new mongoose.Schema({
  accountNumber: {
    type: String,
    required: true
  },
  balance: {
    type: Number,
    required: true
  }
});

// Schema for user
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true
  },
  accounts: [{
    type: mongoose.Schema.Types.ObjectId, // Referencing account documents
    ref: 'Account' // Reference to the Account model
  }]
});
```

Referencing vs. Embedding: Dependent Documents

- Consider a scenario where users have blog posts.
- With a referenced schema if we delete a user then some posts will be left authorless if we fail to delete the user's posts.

```
// User Schema
const userSchema = new mongoose.Schema({
  username: String,
  email: String,
});

// Post Schema
const postSchema = new mongoose.Schema({
  title: String,
  content: String,
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }
});

const User = mongoose.model('User', userSchema);
const Post = mongoose.model('Post', postSchema);
```

Referencing vs. Embedding: Dependent Documents

- Consider a scenario where users have blog posts.
- With an embedded schema if we delete a user then their posts also get deleted.

```
// User Schema
const userSchema = new mongoose.Schema({
  username: String,
  email: String,
  posts: [{
    title: String,
    content: String
  }]
});

const User = mongoose.model('User', userSchema);
```

Mongoose Operations

- A mongoose query is specified as a JSON document; returns a type `Query`.
- A `Query` executed at any time with `exec()`, which returns a promise.
- When querying a document with a reference to another document, we use `query.populate()` to get everything in one object.
- We use `document.save()` to write a document to the database; also returns a promise.

Example

- Let's explore the queries used in the following services in *server.js*:
  - `GET /home`
  - `GET /books`
  - `GET /book_dtls`
  - `POST /newbook`

For You to Do

- Complete the following services in *server.js* with appropriate Mongoose operations:
  - `GET /available` returns list of objects {title, status} where the status is "available".
  - `GET /authors` returns a list of author objects {name, lifespan}.