Web Programming

Lecture 12 – Building a data model with MongoDB and Mongoose

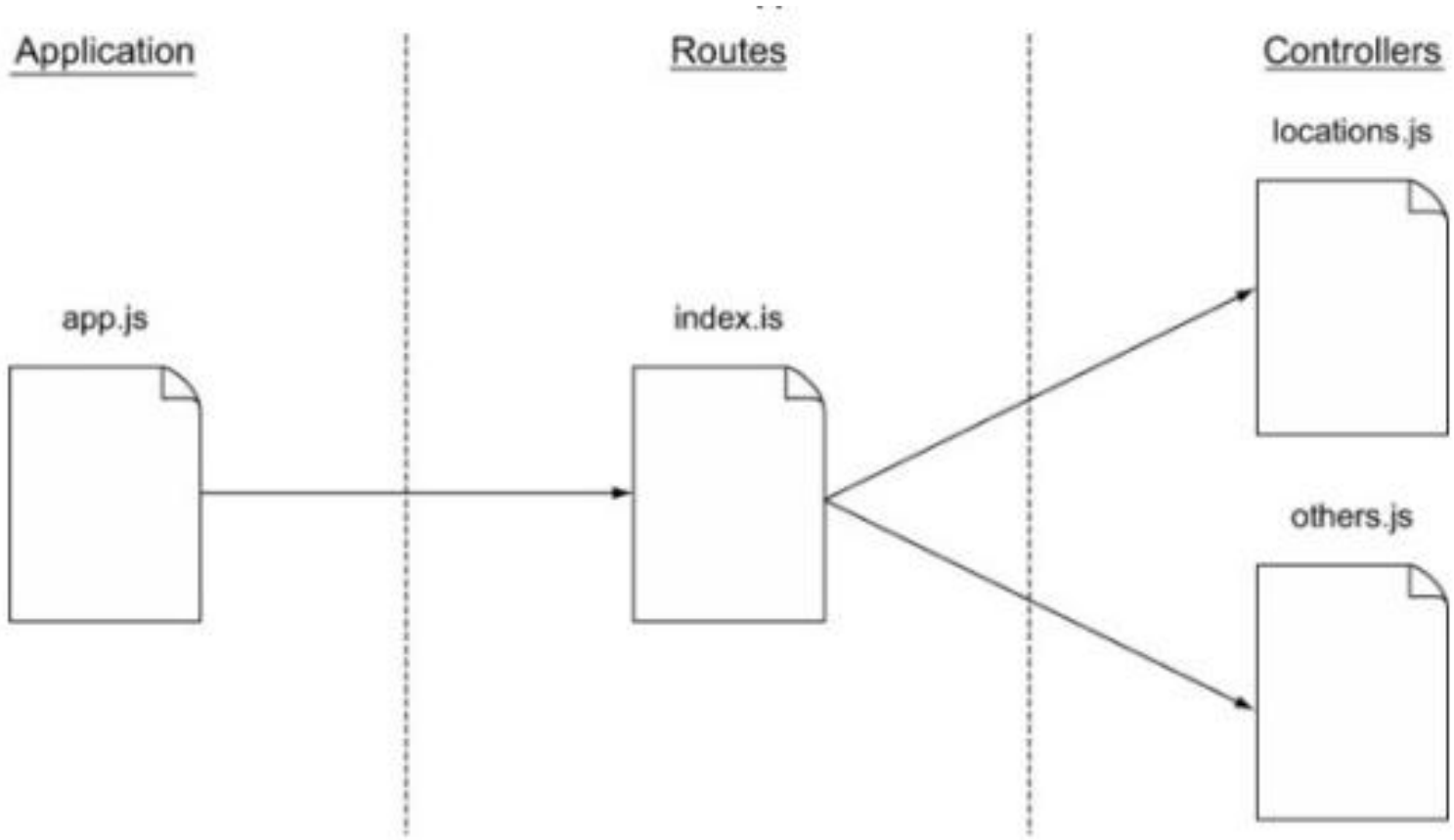
So far …

- Creating and configuring Express projects
  - App.js file
- Setting up an MVC environment
  - Folder structure, requiring modules
- Routing
  - Application level middleware
    - app.use(), app.get(), app.route()
  - Router level middleware
    - Creating a router as a module, defining some routes in it, and mounting the router module on a path in the main app.
    - router.use(), router.get()

Rapid Prototype dev stages

- Stage 1: Build a static site
  - Aims
    - Quickly figure out the layout
    - Ensure that the user flow makes sense
- Stage 2: Design the data model and create the database
- Stage 3: Build our data API
  - Create a REST API that will allow our application to interact with the database
- Stage 4: Hook the database into the application
  - Get our application to talk to our API to get a data-driven app
- Stage 5: Augment
  - Add additional functionality
    - Validation, authentication, etc

Proposed file architecture for routes and controllers in our application



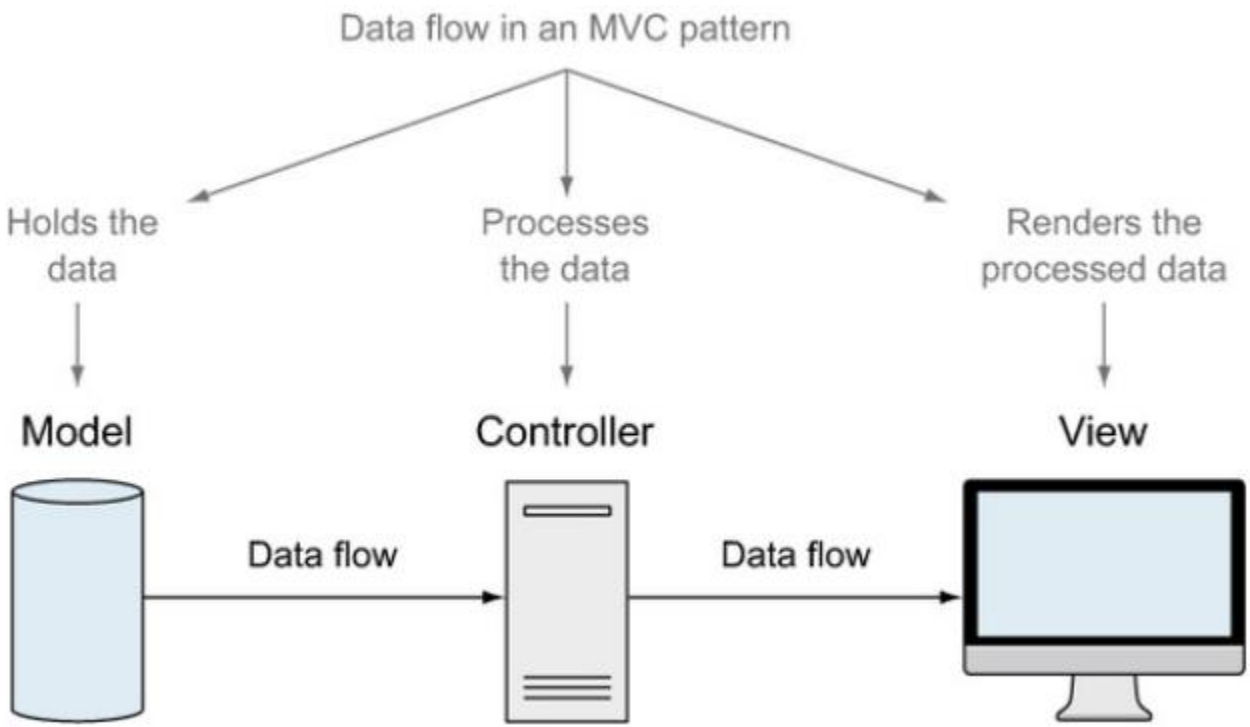
Last Lecture

- Defining the routes and mapping them to controllers
- The layout.pug file is a template for other views to extend!
- Take the data out of the views and make them smarter
- Adding the repeating data array to the controller
- Looping through arrays in a Pug view
- Using includes and mixins to create reusable layout components

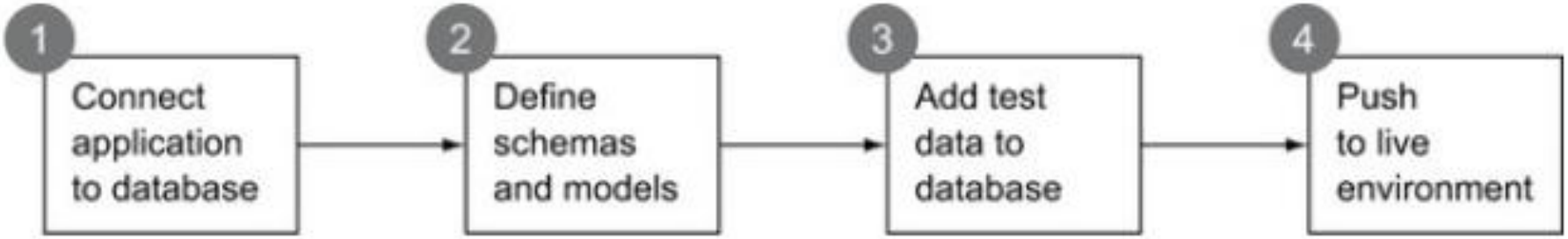
Today’s Agenda

- How Mongoose helps bridge an Express/Node application to a MongoDB database
- Defining schemas for a data model using Mongoose
- Connecting an application to a database

In an MVC pattern, data is held in the model, processed by a controller, and then rendered by a view

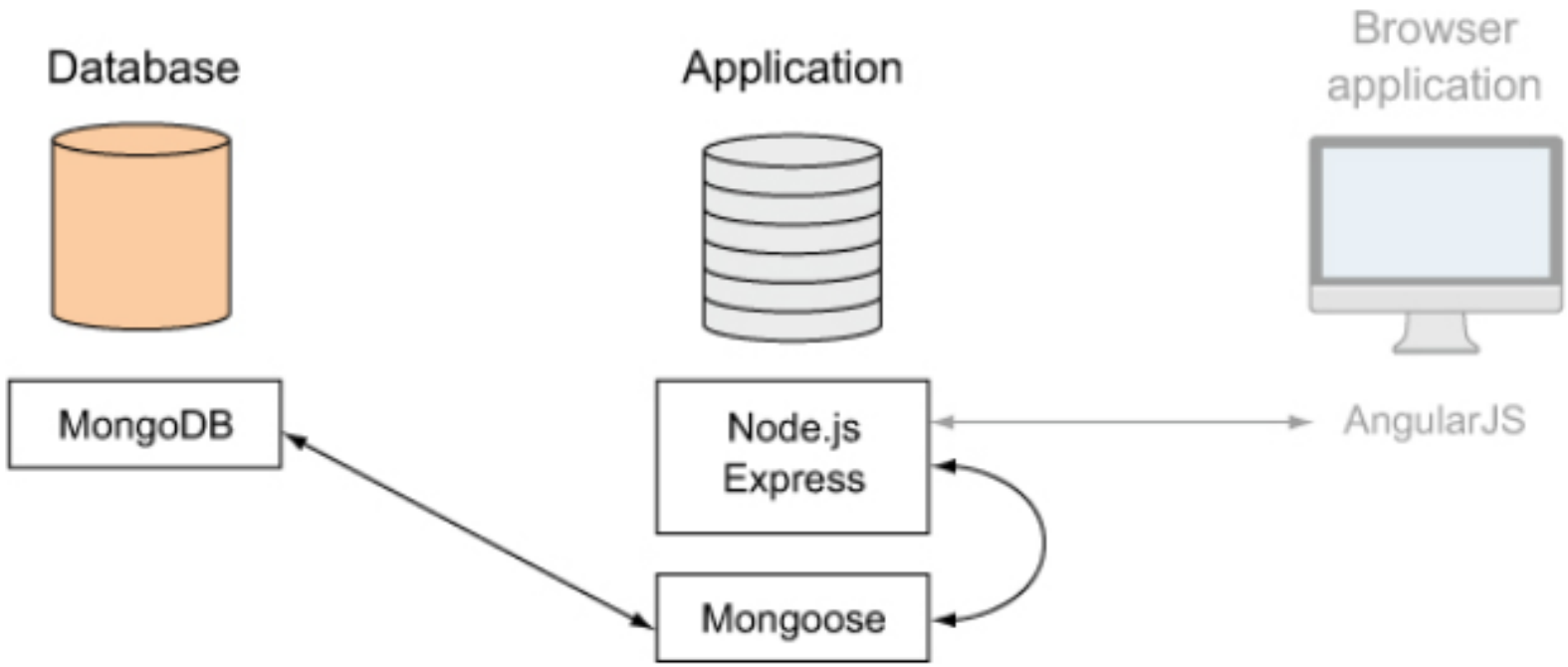


4 steps


1
Connect application to database
2
Define schemas and models
3
Add test data to database
4
Push to live environment

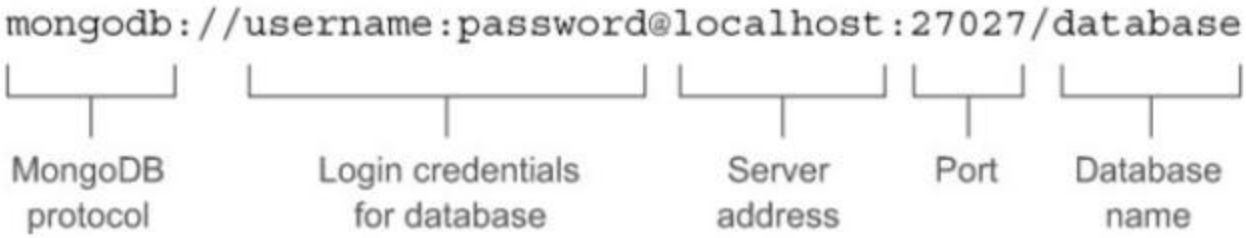
Connecting the Express application to MongoDB using Mongoose

Node/Express application interacts with MongoDB through Mongoose, and Node and Express can then also talk to Angular.



Adding Mongoose to our application

- npm install --save mongoose
- **Adding a Mongoose connection to our application**
  - **Create a file db.js in models folder and add the following line**
    - **var** mongoose = require('mongoose');
  - Bring this file into the application by requiring it in app.js as follows
    - require('./app_server/models/db');
  - **Creating the Mongoose connection**



Creating the Mongoose connection

- **Add the following snippet to db.js**
- **var** dbURI = 'mongodb://localhost/Loc8r';
  mongoose.connect(dbURI);

Monitoring the connection with Mongoose connection events

```
1 mongoose.connection.on('connected', function () {
2   console.log('Mongoose connected to ' + dbURI);
3 });
4 mongoose.connection.on('error',function (err) {
5   console.log('Mongoose connection error: ' + err);
6 });
7 mongoose.connection.on('disconnected', function () {
8   console.log('Mongoose disconnected');
9 });
```

After adding this code in db.js, when you restart your app you should see this

Express server listening on port 3000

Mongoose connected to mongodb://localhost/Loc8r

Closing a Mongoose connection

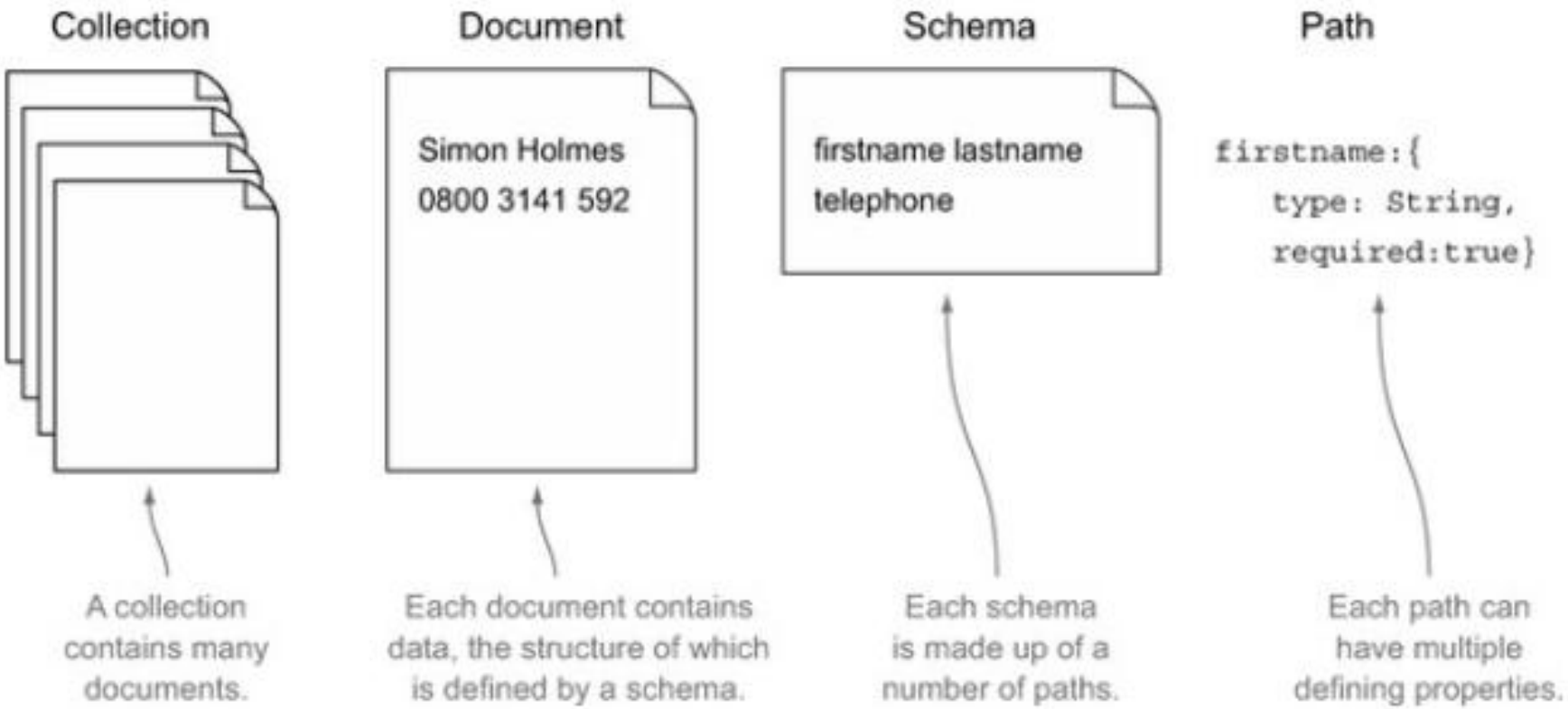
```
1 var gracefulShutdown = function (msg, callback) {
2   mongoose.connection.close(function () {
3     console.log('Mongoose disconnected through ' + msg);
4     callback();
5   });
6 };
```

```
1 process.once('SIGUSR2', function () {
2   gracefulShutdown('nodemon restart', function () {
3     process.kill(process.pid, 'SIGUSR2');
4   });
5 });
6 process.on('SIGINT', function () {
7   gracefulShutdown('app termination', function () {
8     process.exit(0);
9   });
10 });
11 process.on('SIGTERM', function() {
12   gracefulShutdown('Heroku app shutdown', function () {
13     process.exit(0);
14   });
15 });
```

Complete connection file db.js in app_server/models

```
1 var mongoose = require( 'mongoose' );
2 var gracefulShutdown;
3 var dbURI = 'mongodb://localhost/Loc8r';
4 mongoose.connect(dbURI);
5 mongoose.connection.on('connected', function () {
6   console.log('Mongoose connected to ' + dbURI);
7 });
8 mongoose.connection.on('error',function (err) {
9   console.log('Mongoose connection error: ' + err);
10 });
11 mongoose.connection.on('disconnected', function () {
12   console.log('Mongoose disconnected');
13 });
14 gracefulShutdown = function (msg, callback) {
```

Relationships among collections, documents, schemas, and paths in MongoDB and Mongoose



Example MongoDB document and its corresponding Mongoose schema

```
1 {
2   "firstname" : "Simon",
3   "surname" : "Holmes",
4   _id : ObjectId("52279effc62ca8b0c1000007")
5 }
6 {
7   firstname : String,
8   surname : String
9 }
```

Allowed Schema Types

- String
- Number
- Date
- Boolean
- Buffer
- Mixed
- Array
- ObjectId

Defining simple Mongoose schemas

- **Create a file locations.js in models and add the following line**
  - **var** mongoose = require('mongoose');
- In the db.js file add the line at the end
  - require('./locations');
- Mongoose gives a constructor function for defining new schemas
  - **var** locationSchema = **new** mongoose.Schema({ });

Defining a schema from controller data

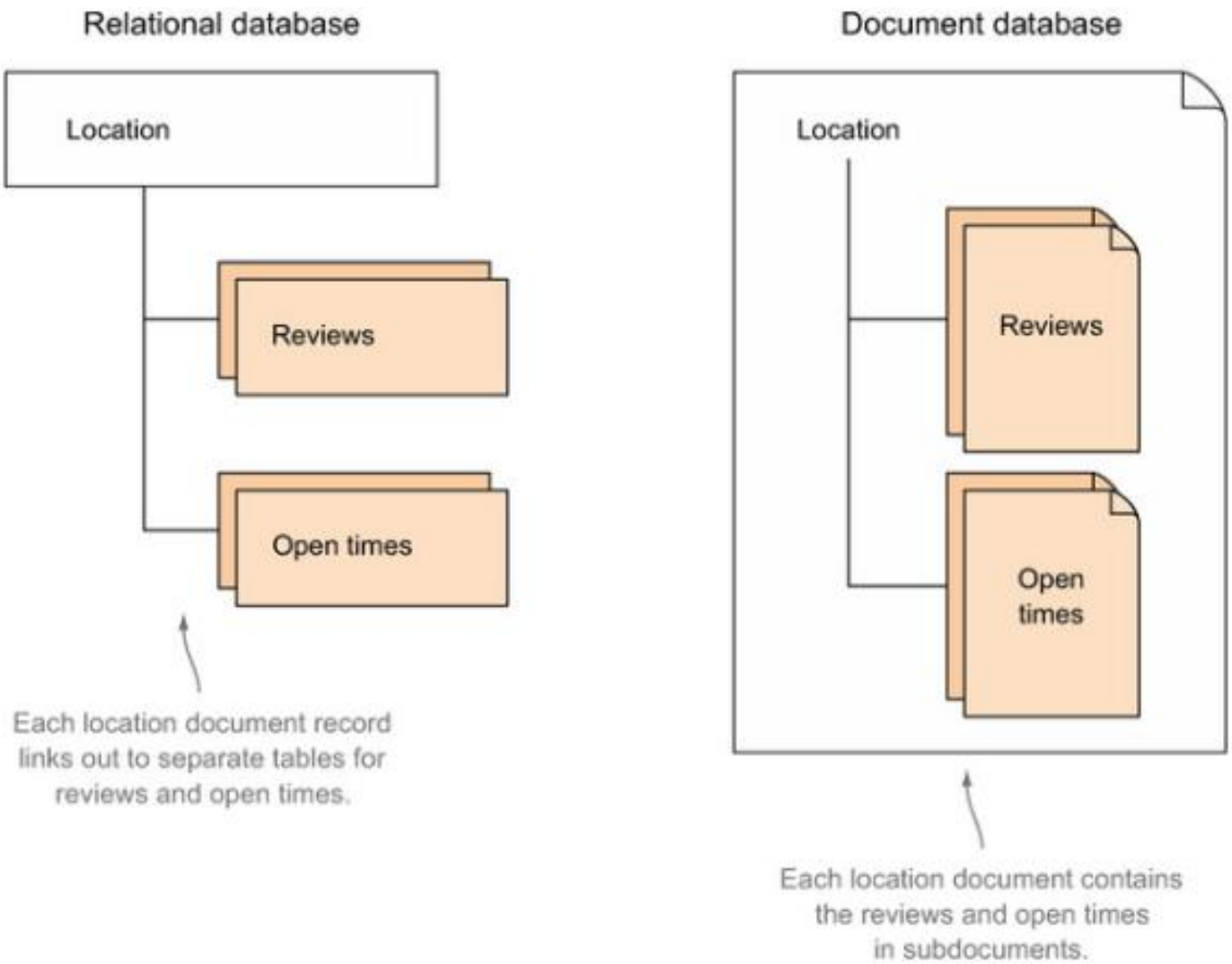
```
1 locations: [{
2   name: 'Starcups',
3   address: '125 High Street, Reading, RG6 1PS',
4   rating: 3,
5   facilities: ['Hot drinks', 'Food', 'Premium wifi'],
6   distance: '100m'
7 }]
```

```
1 var locationSchema = new mongoose.Schema({
2   name: String,
3   address: String,
4   rating: Number,
5   facilities: [String]
6 });
```

Setting up a schema

- **Assigning default values**
  - **rating**: {**type**: Number, "default": 0}
- Adding basic validation
  - name: {type: String, required: true}
  - **rating**: {**type**: Number, "default": 0, **min**: 0, **max**: 5}

Creating more complex schemas with subdocuments



Using nested schemas in Mongoose to define subdocuments

```
1 var openingTimeSchema = new mongoose.Schema({
2   days: {type: String, required: true},
3   opening: String,
4   closing: String,
5   closed: {type: Boolean, required: true}
6 });
```

```
1 var locationSchema = new mongoose.Schema({
2   name: {type: String, required: true},
3   address: String,
4   rating: {type: Number, "default": 0, min: 0, max: 5},
5   facilities: [String],
6   coords: {type: [Number], index: '2dsphere'},
7   openingTimes: [openingTimeSchema]
8 });
```

Using nested schemas

```
1 var reviewSchema = new mongoose.Schema({
2   author: String,
3   rating: {type: Number, required: true, min: 0, max: 5},
4   reviewText: String,
5   createdOn: {type: Date, "default": Date.now}
6 });
```

```
1 var locationSchema = new mongoose.Schema({
2   name: {type: String, required: true},
3   address: String,
4   rating: {type: Number, "default": 0, min: 0, max: 5},
5   facilities: [String],
6   coords: {type: [Number], index: '2dsphere'},
7   openingTimes: [openingTimeSchema],
8   reviews: [reviewSchema]
9 });
```

Final location schema definition, including nested schemas in models/locations.js

```
var mongoose = require( 'mongoose' );
var reviewSchema = new mongoose.Schema({
  author: String
  rating: {type: Number, required: true, min: 0, max: 5},
  reviewText: String,
  createdOn: {type: Date, default: Date.now}
});
var openingTimeSchema = new mongoose.Schema({
  days: {type: String, required: true},
  opening: String,
  closing: String,
  closed: {type: Boolean, required: true}
});
var locationSchema = new mongoose.Schema({
  name: {type: String, required: true},
  address: String,
  rating: {type: Number, "default": 0, min: 0, max: 5},
  facilities: [String],
  coords: {type: [Number], index: '2dsphere'},
  openingTimes: [openingTimeSchema],
  reviews: [reviewSchema]
});
```

Compiling a model from a schema

Add the following line to the locations.js file

```
mongoose.model('Location', locationSchema, 'Locations');
```

Connection name

The name of the model

The schema to use

MongoDB collection name (optional)

Project Assignment 2: Deadline (15th October 2018 11:59 pm)

- Make static pages of your application for every use case/feature
- The code should be pushed to the Project Assignment repo and the heroku link should be added in the readme
- You should implement MVC architecture, data should be passed from the controller to the views
- The models folder should contain schema definitions

References

- Getting MEAN with Mongo, Express, Angular and Node (simon Holmes)