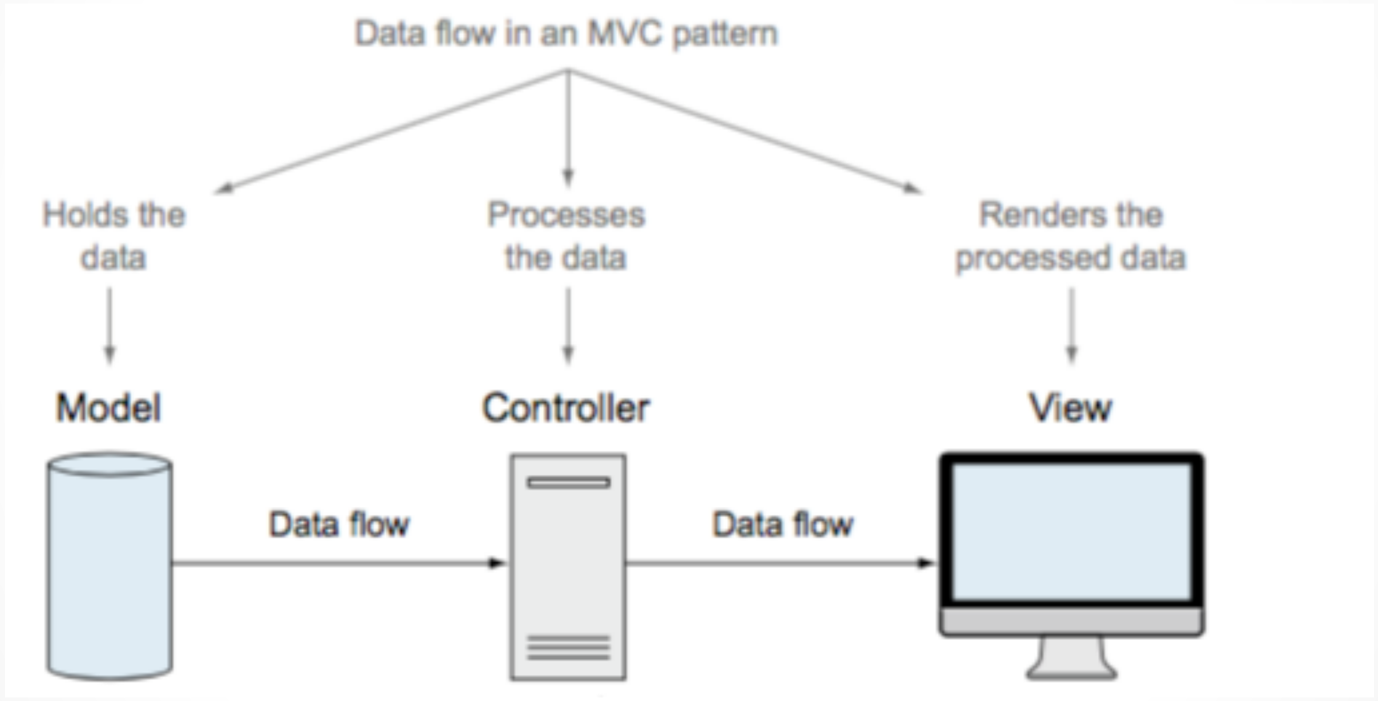
MongoDB

Building data model with MongoDB and Mongoose

MVC Pattern


Data flow in an MVC pattern
Holds the data
Processes the data
Renders the processed data
Model
Controller
View
Data flow
Data flow

Connect Express app to MongoDB with Mongoose

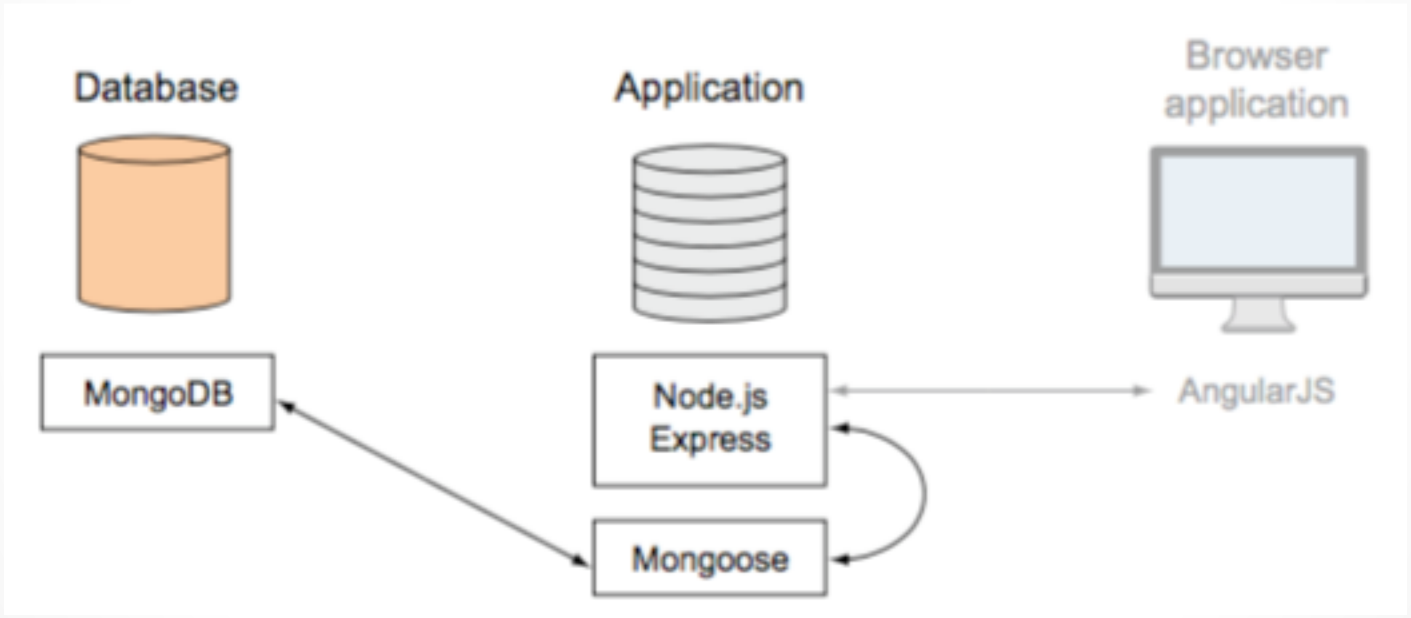
- Could use native MongoDB driver, but not easy to work with
- MongoDB native driver does not offer built-in way of defining and maintaining data structures
- Mongoose exposes most of the functionality of the native driver, but in a more convenient way
- Mongoose enables us to define data structures, and models, maintain them, and use them to interact with the DB

Adding Mongoose to app

- Install mongoose so that MongoDB talks to Monngoose and Mongoose talks to node & express

```
$ npm install mongoose --save
```

Data interactions in the Mean stack



Adding Mongoose connection to app

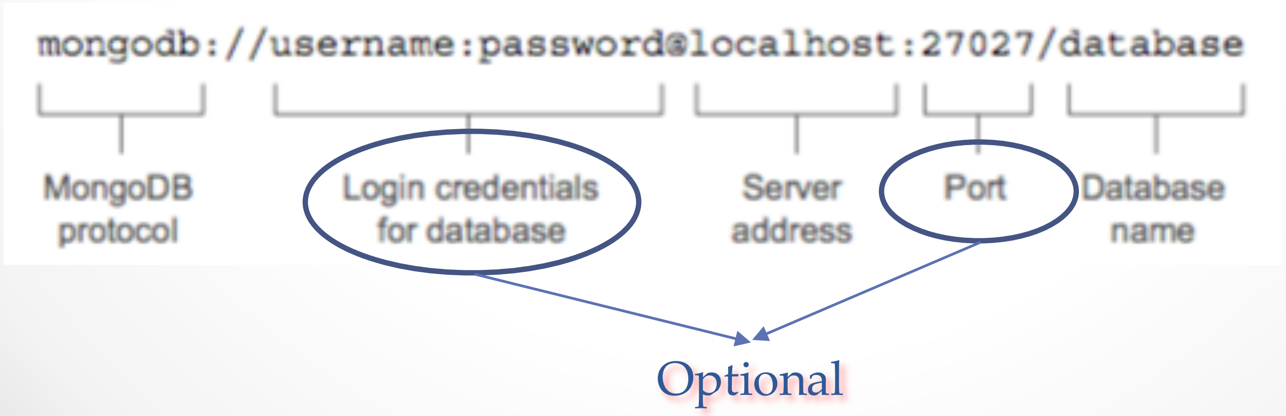
- Mongoose opens a pool of five reusable connections when it connects to a MongoDB database

- This pool of connections is shared between all requests

- **Best practice**: open the connection when your application starts up; leave it open until your application restarts or shuts down.

Setting up connection file

- We will be working in our **ContactsAppBackend** application
- 2 Step process
  - Creating a file called **models/db.js**
  - Use the file in the application by **requiring it in app.js**
- Creating the mongoose connection:



```
mongodb://username:password@localhost:27027/database
```

Monitoring connection with connection events

- Mongoose will **publish events** based on the status of the connection
- Using events to see
  - when the connection is made,
  - when there's an error,
  - and when the connection is disconnected.

- When any one of these events occurs we'll log a message to the console.

```
mongoose.connection.on('connected', function () {
        console.log('Mongoose connected to ' + dbURI);
});
```

Closing mongoose connection

- If you restart the application again, however, you'll notice that you don't get any disconnection messages
- This is because the Mongoose connection doesn't automatically close when the application stops or restarts
- We need to listen for changes in the Node process to deal with this
  - To monitor when the application stops we need to listen to the Node.js process for an event called **SIGINT**.

Listening for SIGINT on Windows

- If you're running on Windows and the disconnection events don't fire, you can emulate them
  - install readline package and
  - add code to db.js to emulate firing of SIGINT signal

```
$ npm install readline --save
```

Capturing process termination events

- If you're using **nodemon** to automatically restart the application, then you'll also have to listen to a second event on the Node process called **SIGUSR2**
- We need three event listeners and one function to close the database connection
- Closing the database is an asynchronous activity, so we're going to need to pass through whatever function is required to restart or end the Node process as a callback

Managing multiple DBs

- Connection in db.js is a default connection

- Need to create named connection to connect to a 2nd DB
  - In place of **mongoose.connect**, use **mongoose.createConnection**
  - Use variable to refer to 2nd connection

```
var dbURIUsr = 'mongodb://localhost/userDbase';
var usrDB = mongoose.createConnection(dbURIUsr);
```

Why model data

- Some times we need structure to our data
- Structure to data gives us consistent naming structure
- Structure of data can accurately reflect the needs of the app
    - Modeling our data describes how we wish to use the data in the app— how the data should be structured
- Mongoose is excellent for helping us model our data

Benefits of mongoose

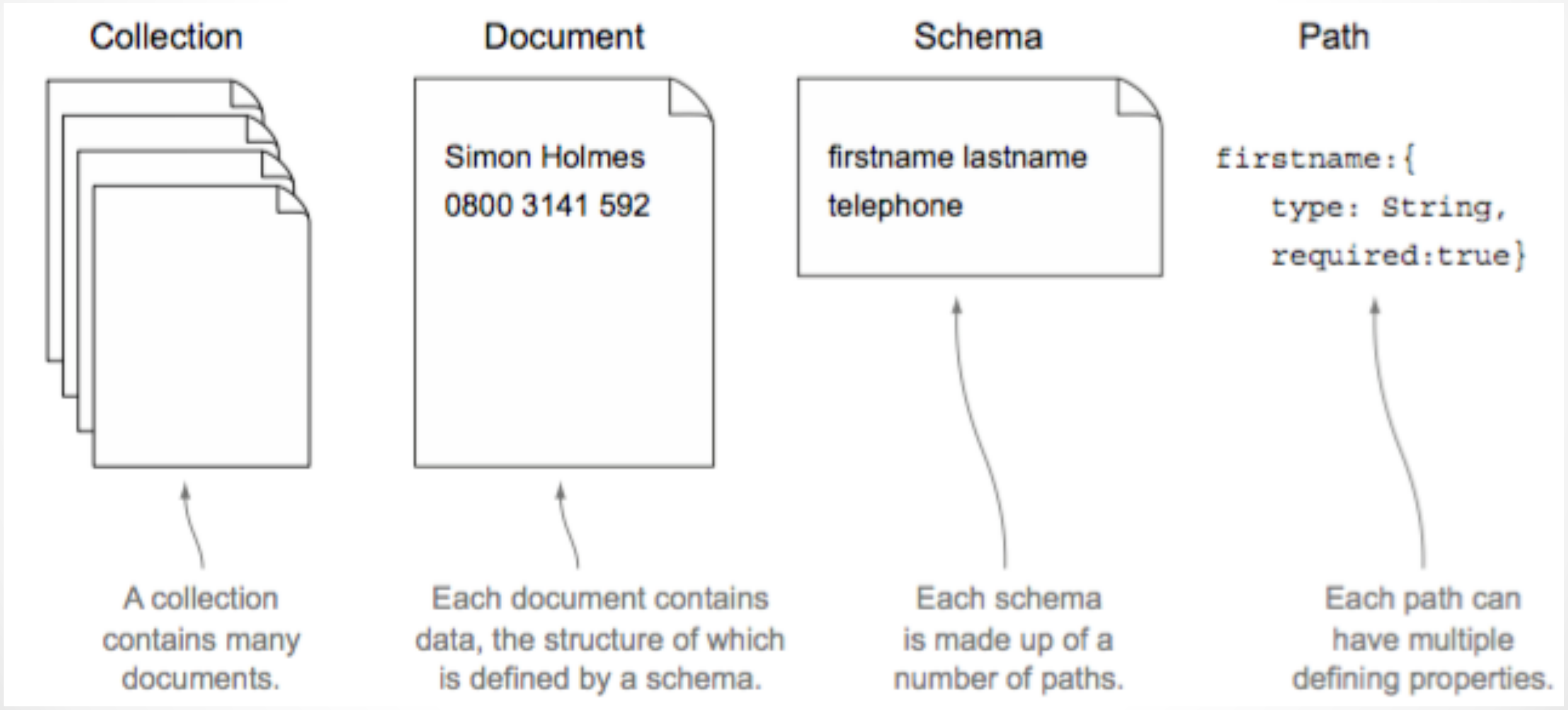
- Mongoose was built specifically as a MongoDB Object-Document Modeler (**ODM**) for Node applications

- One of the key principles is that you can manage your data model from within your application

- You don't have to mess around directly with databases or external frameworks or relational mappers

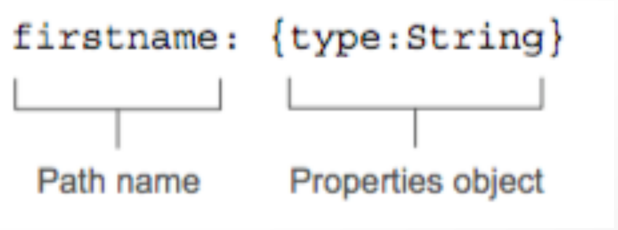
- You can just define your data model in the comfort of your application

Naming conventions


Collection
Document
Schema
Path
Simon Holmes
0800 3141 592
firstname lastname
telephone
firstname:{
type: String,
required:true}
A collection contains many documents.
Each document contains data, the structure of which is defined by a schema.
Each schema is made up of a number of paths.
Each path can have multiple defining properties.

How mongoose models data

- A **model** is the compiled version of a schema
- All data interactions using Mongoose go through the model
- A schema bears a strong resemblance to the data
    - The schema defines the **name** for each data path, and
    - the **data type** it will contain
    - (e.g., model/contact.js)



Allowed schema paths

- **String** - Any string, UTF-8 encoded
- **Number** - default support is enough for most cases
- **Date** - Typically returned from MongoDB as an ISODate object
- **Boolean** - True or false
- **Buffer** - For binary information such as images
- **Mixed** - Any data type
- **Array** - Can either be an array of the same data type, or an array of nested sub-documents
- **ObjectId** - For a unique ID in a path other than _id; typically used to reference _id paths in other documents

Defining simple mongoose schemas

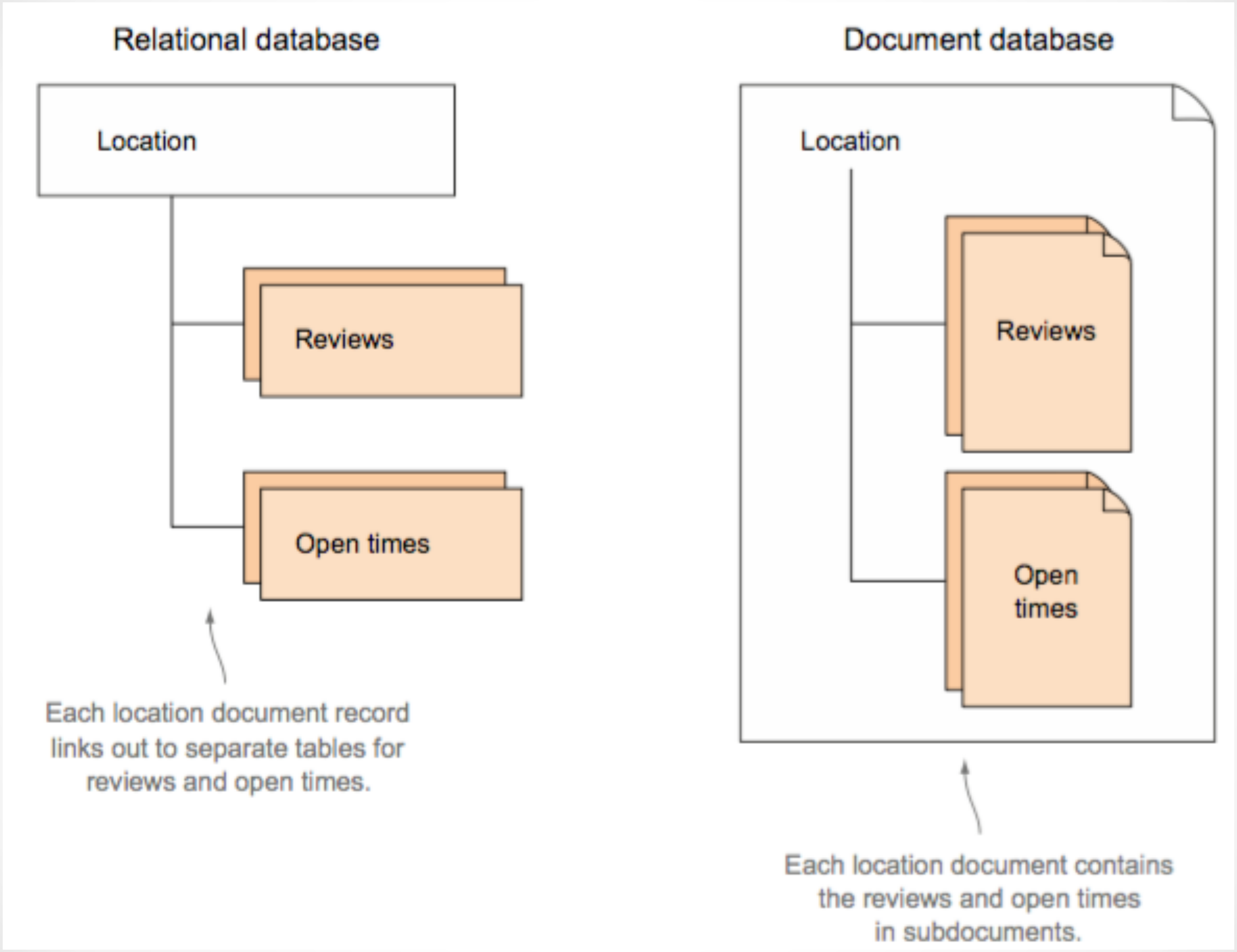
- Schema should be defined in a **model** folder alongside db.js
  - Plural form of the name is preferred
- Should **require it in db.js** or in app.js
- Need Mongoose in model/contacts.js to define a mongoose schema
  - Must require mongoose
- Mongoose provides a constructor function for defining new schema
  - Mongoose.Schema({})

Default values & basic validation

```
var reviewSchema  = new mongoose.Schema({
    stars: {type: Number, ”default”: 0,
            min: 0, max 5},
    body: {type: String, required: true},
    author: String,
    createdOn: {type: Date,
              ”default”: Date.now}
});
```

Complex schemas with subdocuments

- Think of **product object** from **store app** with nested array of reviews
- In relational DB you would create separate table for reviews and join the tables together in a query when you need the information
- Document DBs don't work that way
- Anything that belongs specifically to a parent document should be contained ***within*** the document


Relational database
Location
Reviews
Open times
Each location document record links out to separate tables for reviews and open times.
Document database
Location
Reviews
Open times
Each location document contains the reviews and open times in subdocuments.

Subdocuments

- MongoDB offers the concept of **subdocuments** to store repeating, nested data
- Subdocuments are very much like documents in that **they have their own schema** and each is given a **unique _id** by MongoDB when created.
- But subdocuments are nested inside a document and they can **only** be accessed as a path of that parent document.

Nested schema to define subdocuments

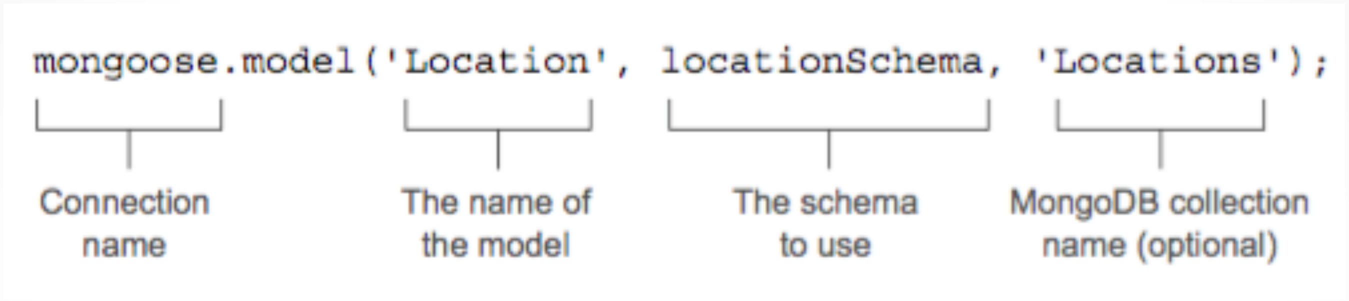
```
var productSchema  = new mongoose.Schema({
    price: {type: Number, min: 0.0},
    name: {type: String, required: true},
    description: String,
    images: [String],
    reviews: [reviewSchema]
});
```

- reviewSchema – Add nested schema by referencing another schema object as an array

Compiling mongoose schemas into models

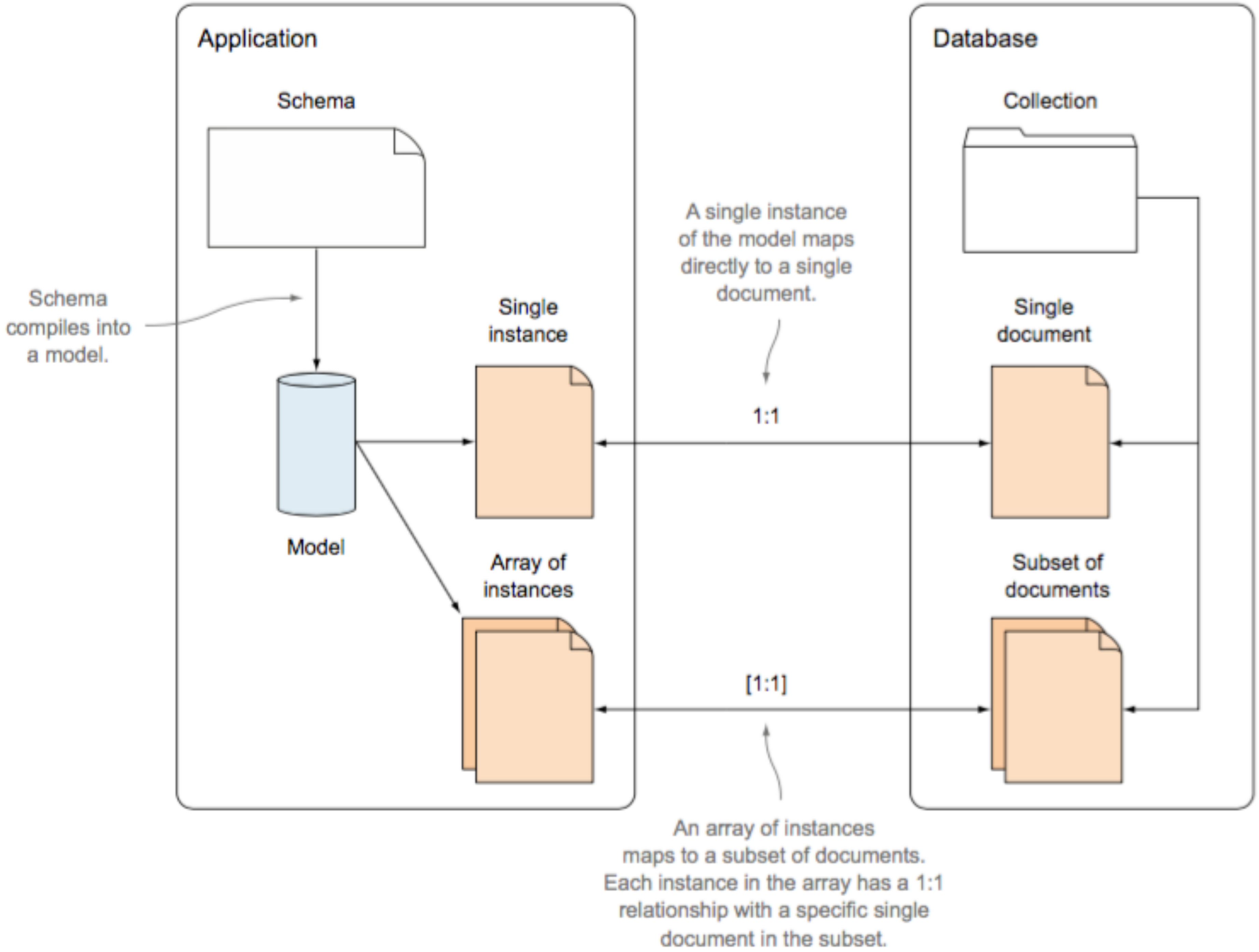
- An application doesn't interact with the schema directly when working with data
- Data interaction is done through models
- In Mongoose, a model is a compiled version of the schema
- Once compiled, a single instance of the model maps directly to a single document in your database

Compiling model from schema



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

- The MongoDB collection name is optional
- If you exclude it Mongoose will use a lowercase pluralized version of the model name
- For example, a model name of Location would look for a collection name of locations unless you specify something different


Application
Schema
Schema compiles into a model.
Model
Single instance
Array of instances
A single instance of the model maps directly to a single document.
1:1
[1:1]
An array of instances maps to a subset of documents. Each instance in the array has a 1:1 relationship with a specific single document in the subset.
Database
Collection
Single document
Subset of documents

Resources

Getting MEAN with Mongo, Express, Angular, and Node

Simon Holmes
November 2015
ISBN 9781617292033
440 pages printed in black & white