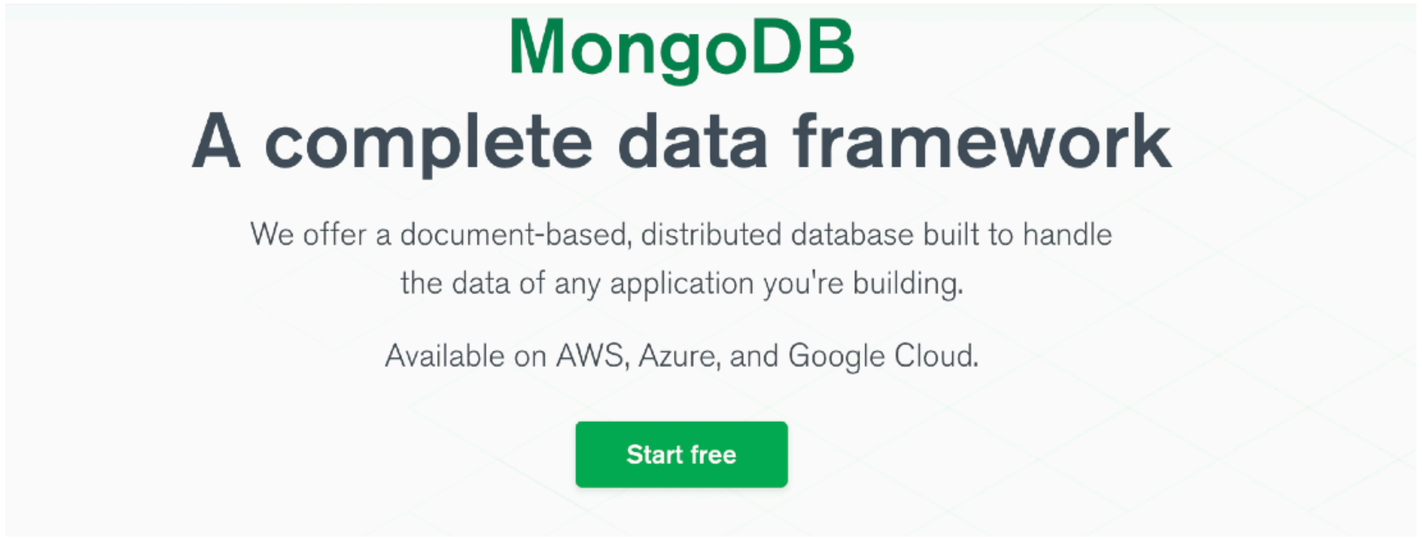
Persisting Data with Mongo DB

Prof. Fabio Ciravegna
Dipartimento di Informatica
Università di Torino
fabio.ciravegna@unito.it

UNIVERSITÀ DI TORINO

MongoDB

- MongoDB is an open-source document database that provides high performance, high availability, and automatic scaling


MongoDB

A complete data framework

We offer a document-based, distributed database built to handle the data of any application you're building.

Available on AWS, Azure, and Google Cloud.

https://docs.mongodb.com/getting-started/shell/introducti

Documents

- A record in MongoDB is a document, which is a data structure composed of field and value pairs.
- MongoDB documents are similar to JSON objects.
- The values of fields may include other documents, arrays, and arrays of documents

SQL Records -> Mongos' Documents

Collections

- MongoDB stores documents in collections. Collections are analogous to tables in relational databases.
  - Unlike a table, however, a collection does not require its documents to have the same schema
- In MongoDB, documents stored in a collection must have a **unique _id** field that acts as a **primary key**

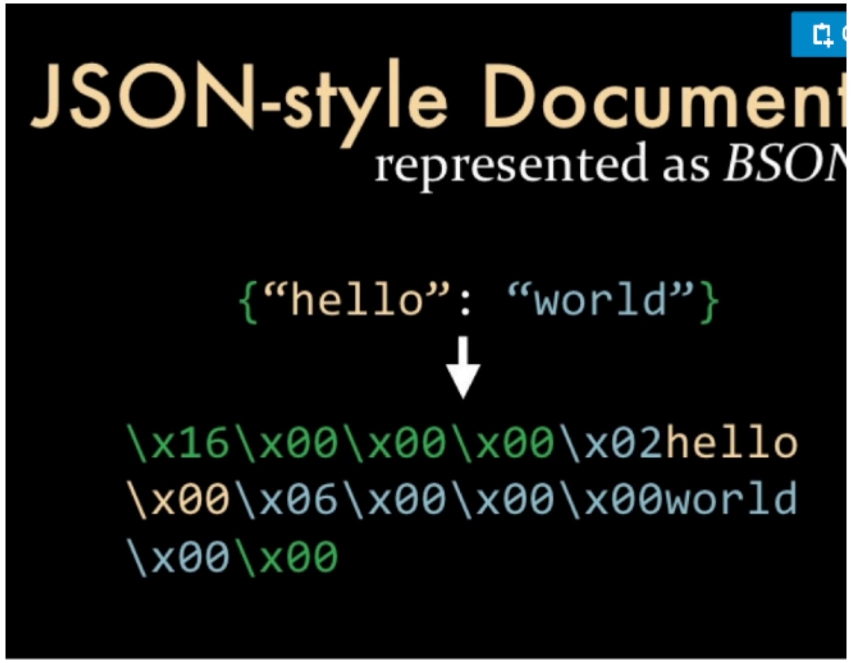
SQL Relations -> Mongos' Collections

A restaurant **document** example

```
{
  "_id" : ObjectId("54c955492b7c8eb21818bd09"),
  "address" : {
    "street" : "2 Avenue",
    "zipcode" : "10075",
    "building" : "1480",
    "coord" : [ -73.9557413, 40.7720266 ]
  },
  "borough" : "Manhattan",
  "cuisine" : "Italian",
  "grades" : [
    {
      "date" : ISODate("2014-10-01T00:00:00Z"),
      "grade" : "A",
      "score" : 11
    },
    {
      "date" : ISODate("2014-01-16T00:00:00Z"),
      "grade" : "B",
      "score" : 17
    }
  ],
  "name" : "Vella",
  "restaurant_id" : "41704620"
}
```

BSON

- Mongo’s documents are internally represented as binary JSON
  - this is why in express you are likely to have to include the npm module called bson

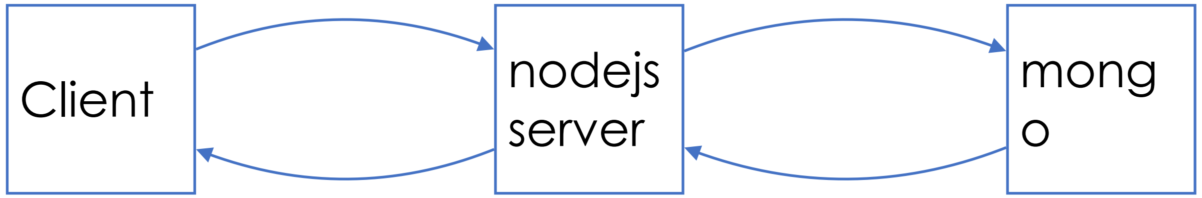

JSON-style Document
represented as BSON

```
{“hello”: “world”}
↓
\x16\x00\x00\x00\x02hello
\x00\x06\x00\x00\x00world
\x00\x00
```

https://www.slideshare.net/mdirolf/introduction-to-mon

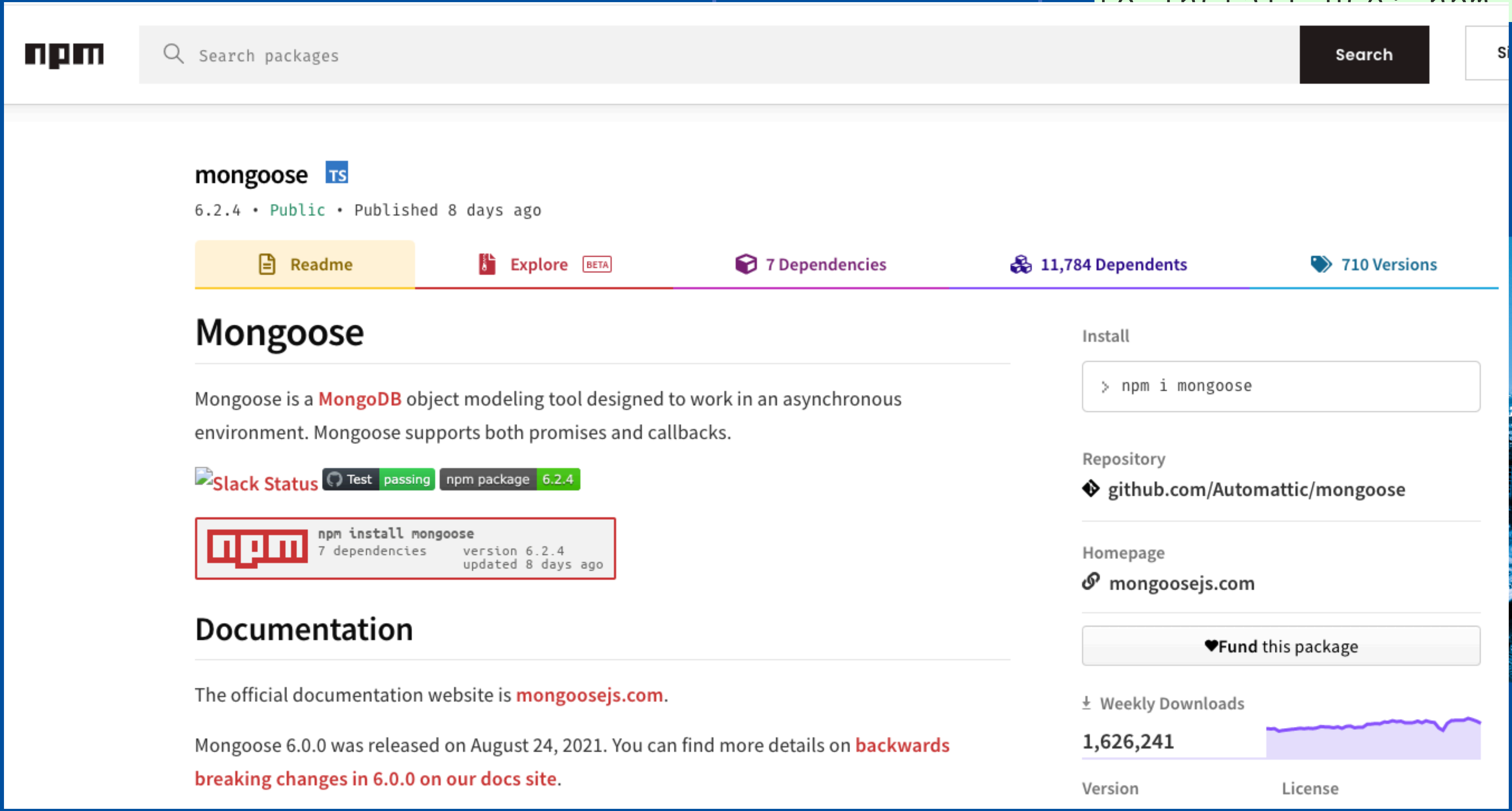
A separate process

- Mongo, as any db system, must run in a separate process from your main nodejs server
  - remember: nodeJS is fast and scalable but no long-running process is to be run on its single thread
    - create a separate process for that and connect using the appropriate library
      - Mongoose in our case



https://www.npmjs.com/p

`to install use: npm i mongoose`



Search packages

Search

mongoose TS

6.2.4 • Public • Published 8 days ago

Readme | Explore BETA | 7 Dependencies | 11,784 Dependents | 710 Versions

Mongoose

Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment. Mongoose supports both promises and callbacks.

Slack Status | Test passing | npm package 6.2.4

```
npm install mongoose
7 dependencies     version 6.2.4
                   updated 8 days ago
```

Documentation

The official documentation website is mongoosejs.com.

Mongoose 6.0.0 was released on August 24, 2021. You can find more details on backwards breaking changes in 6.0.0 on our docs site.

Install

```
> npm i mongoose
```

Repository

github.com/Automattic/mongoose

Homepage

mongoosejs.com

♥Fund this package

Weekly Downloads

1,626,241

Version

License

Creating a Database

- Install MongoDB and start the process
  - see lab class
- In your Express programme import Mongoose

```
const mongoose = require('mongoose');
```

- Create a database in MongoDB:
  - start by creating a MongoClient object,
  - then specify a MongoDB connection URL with
    - the correct ip address and
    - the name of the database you want to create.
- MongoDB will create the database if it does not exist, and make a connection to it.

https://www.w3schools.com/nodejs/nodejs_mongodb

```
const mongoose = require('mongoose');

//The URL which will be queried. Run "mongod.exe" for this //
to connect
const mongoDB = 'mongodb://localhost:27017/characters';

mongoose.Promise = global.Promise;

connection = mongoose.connect(mongoDB, {
   checkServerIdentity: false,
})
    .then(() => {
        console.log('connection to mongodb worked!');
    })
    .catch((error) => {
        console.log('connection to mongodb did not work! '
                        + JSON.stringify(error));
    });
```

Connecting using await

```
configDB.url= mongodb://localhost:27017/database_name
```

```
await mongoose.connect(configDB, {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  useFindAndModify: false,
  useCreateIndex: true
});
```

Mongoose buffers all the commands until it's connected to the database. This means that you don't have to wait until it connects to MongoDB in order to define models, run queries, etc.
but I suggest always to wait otherwise you will get a huge number of consecutive errors if the connection fails

Schemas

- Although mongo is schema-less, it helps to define schemas in order to
  - Validate data integrity of documents
  - Legibility/maintenance
    - pretty much like in Javascript where although the same variable can contain different data structures, you end up specialising the variables to only one type
- Mongoose imposes a schema for document models

Schemas

- A schema is the equivalent of a java interface
  - It is not a class that can be used to create instances
  - it must be implemented in a MODEL before being used to create instances

```
var Character = new Schema(
    {
        first_name: {type: String},
        family_name: {type: String},
        dob: {type: Number},
        whatever: {type: String} //any other field
    }
);
```

Aside from defining the structure of your documents and the types of data you're storing, a Schema handles the definition of:

- Validators (async and sync)
- Defaults
- Getters
- Setters
- Indexes
- Middleware
- Methods definition
- Statics definition
- Plugins
- pseudo-JOINs

Validation

Mongoose provides built-in and custom validators, and synchronous and asynchronous validators. It allows you to specify both the acceptable range or 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.

Validation Example

max string length

```
var Character = new Schema(
    {
        first_name: {type: String, required: true, max: 100},
        family_name: {type: String, required: true, max: 100},
        dob: {type: Number, required: true, max: 2023},
        whatever: {type: String} //any other field
    }
);
```

max value

see details at http://mongoosejs.com/docs/validation.html

UNIVERSITÀ
DI TORINO

Validation example

```
const breakfastSchema = new Schema({
  eggs: {
    type: Number,
    min: [6, 'Too few eggs'],
    max: 12
  },
  bacon: {
    type: Number,
    required: [true, 'Why no bacon?']
  },
  drink: {
    type: String,
    enum: ['Coffee', 'Tea'],
    required: function() {
      return this.bacon > 3;
    }
  }
});
```

UNIVERSITÀ DI TORINO

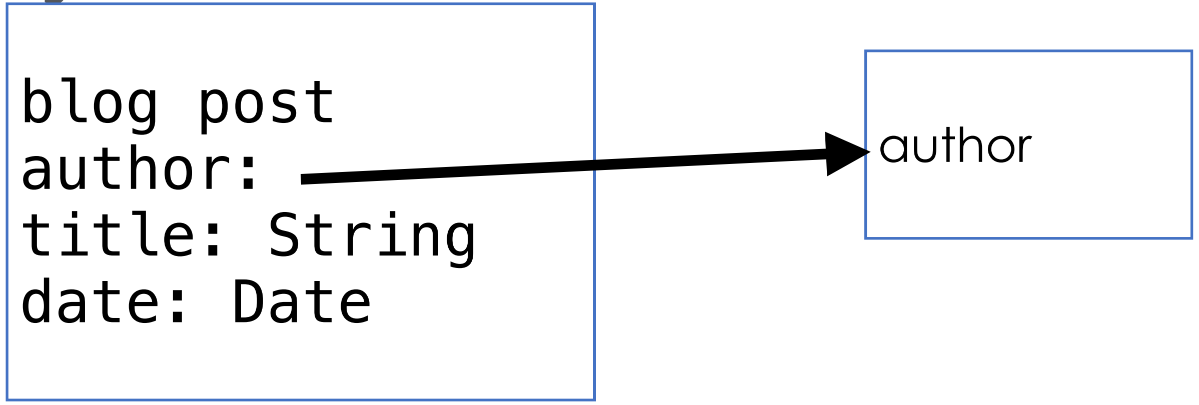
Virtual properties

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs

- 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,
  - It is easier and cleaner and uses less disk space
  - it allows for dynamic properties
- Examples:
  - a full name virtual property starting from concrete fields called first name and last name
  - (Dynamic): the age of a person computed from the current year and date of birth

```
// Virtual for age of a person
Character.virtual('age')
.get(function () {
  const currentDate = new Date().getTime();
  return currentDate-this.date_of_birth;
});
```

Objects in fields



```
const Schema = mongoose.Schema;
const ObjectId = Schema.ObjectId;

const BlogPost = new Schema({
  author: ObjectId,
  title: String,
  body: String,
  date: Date
});
```

- `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.

Middleware : pre/post operations

- Used to perform operations before saving (for example to notify an administrator)

```
schema.pre('set', function (next, path, val, typel) {
  // `this` is the current Document
  this.emit('set', path, val);

  // Pass control to the next pre
  next();
});
```

- you can mutate the incoming method arguments so that subsequent middleware see different values for those arguments. To do so, just pass the new values to next

```
.pre(method, function firstPre (next, methodArg1, methodArg2) {
  // Mutate methodArg1
  next("altered-" + methodArg1.toString(), methodArg2);
});
```

UNIVERSITÀ DI TORINO

Models implement Schemas

- The Schema allows you to define the fields stored in each document along with their validation requirements and default values.
- Schemas are then "compiled" into models using the **`mongoose.model()`** method.
- Once you have a model you can use it to
  - find,
  - create,
  - update,
  - delete

} instances of that model (i.e. records in the db)

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

Models

```
const MyModel = mongoose.model('ModelName',
mySchema);
```

- This creates a model from the schema
  - The model is the equivalent to a Java Class
    - You can create instances off it

```
const instance = new MyModel();
//do some operations here e.g. by initialising the fields
instance.save(function (err) {

});
```

Instances with field initialisation

new <ModelName>({<fields>}) creates an instance of a model

```
let character = new Character({
    first_name: 'Mickey',
    family_name: 'Mouse',
    dob: dob
});

character.save()
    .then ((results) => {
        console.log("object created in init: "+ JSON.stringify(results))
    })
    .catch ((error) => {
        console.log(JSON.stringify(error));
    });
```

Searching

- You can search for records using query methods, specifying the query conditions as a JSON document
  - e.g. find all athletes that play tennis, returning just the fields for athlete name and age
    - 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.

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

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

// find all athletes (MODEL!) who play tennis, selecting the 'name' and 'age' fiel

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

`{ 'sport': 'Tennis' }` is the condition

- you will notice that posing conditions is visually similar to creating an object with those fields

`'name age'` are the fields to be returned

Always remember to check for errors!!

UNIVERSITÀ DI TORINO

Promise-like structure (suggested)

```
find the age of Mikey Mouse
```

```
Character.find({first_name: userData.firstname, family_name: userData.lastname},
    'first_name family_name dob age')
    .then(characters => {
        let character = null;
        if (characters.length > 0) {
            let firstElem = characters[0];
            character = {
                name: firstElem.first_name,
                surname: firstElem.family_name,
                dob: firstElem.dob,
                age: firstElem.age
            };
            res.json(character.age);
        } else {
            res.json("not found");
        }
    })
    .catch((err) => {
        res.status(500).send('Invalid data or not found!' + JSON.stringify(err));
    });
```

Mongoose operations are *Then*able

```
Band.findOne({name: "Guns N' Roses"})
    .then(doc => {
          // use doc
          })
    .catch (e => {
    })
```

Technically they are not promises but you can imagine them as such

Callback Parameters

- If you specify a callback the query will execute immediately while the callback will be invoked when the search completes.
- 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
- **However pls do not use callbacks. Use .then/.catch**

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

UNIVERSITÀ DI TORINO

Querying step by step

```
// 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
})
```

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

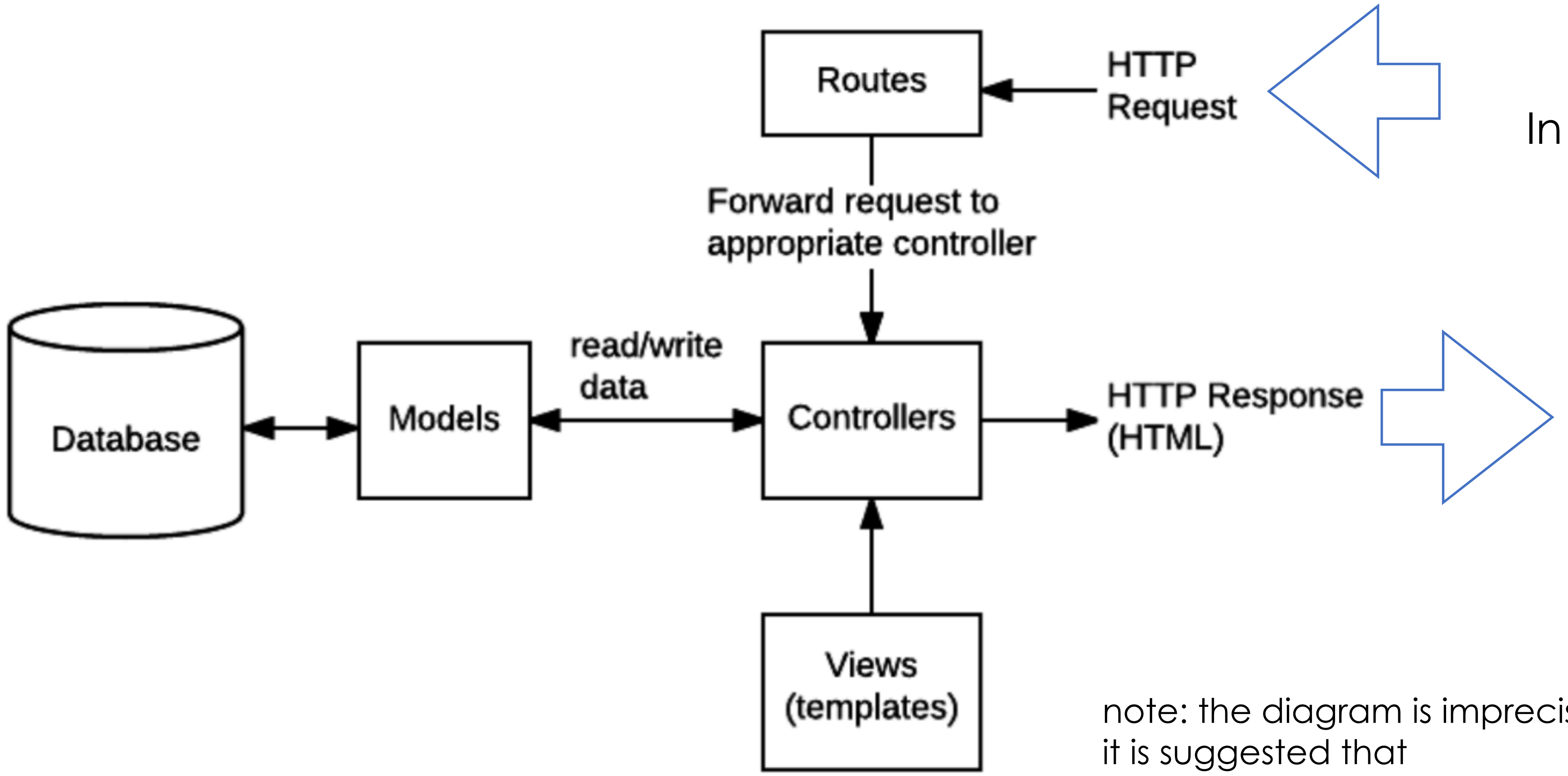
composing with dot

- Better method:
  - using a where() function and chaining all the parts of the query usin the dot operator (.)
  - this is clearer than using separate statements or a big query with implicit parameters as in slide 26

```
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); // callback is the name of our callback function.
```

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

MVC project organisation



In

out

note: the diagram is imprecise:
it is suggested that
the controller returns the values
to the roots so to have just one
entry/exit point

https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_No

UNIVERSITÀ DI TORINO

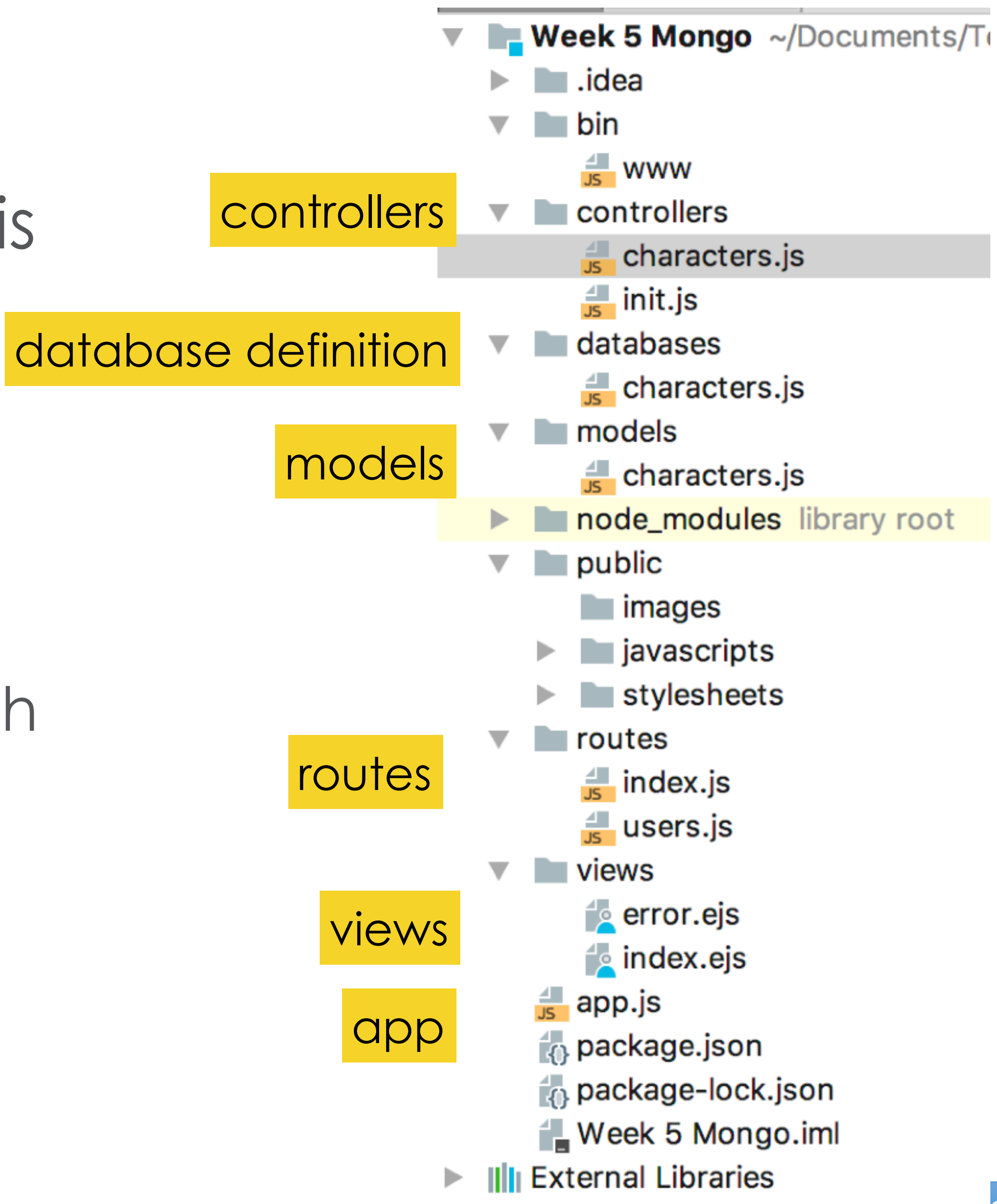
...Organisation

- Routes to forward the supported requests (and any information encoded in request URLs) to the appropriate controller functions.
- Controller functions to get the requested data from the models and return it to the user (e.g. via JSON)
- Models: the declaration of the MongoDb documents types
- Views our enhanced-html pages

In WebStorm

- The program is organised in this way
- There is a database called 'characters'
  - which has a model called 'Character' representing name, surname and year of birth of each character

Note: you <u>must</u> learn this organisation and use it in your assignment!!



```
▼ Week 5 Mongo  ~/Documents/T
  ▶ .idea
  ▼ bin
      www
  ▼ controllers
      characters.js
      init.js
  ▼ databases
      characters.js
  ▼ models
      characters.js
  ▶ node_modules  library root
  ▼ public
      images
    ▶ javascripts
    ▶ stylesheets
  ▼ routes
      index.js
      users.js
  ▼ views
      error.ejs
      index.ejs
    app.js
    package.json
    package-lock.json
    Week 5 Mongo.iml
▶ External Libraries
```

UNIVERSITÀ DI TORINO

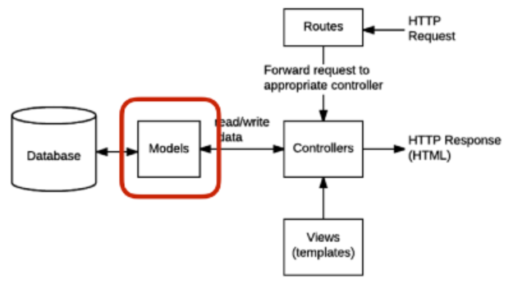
Routes

- Routes are the entry point of the database via the nodeJS server
- they will communicate with the controller which contains the database access functions

```
// require the controller so to be able to access its functions
var author_controller = require('../controllers/authorController');

// route example: GET a request for list of all Authors.
// author_list is a function defined in the controller file author_controller
// you may ask: if that is a function, why is it not called as author_list() ?
// Because in the route we declare the callback. The event handler will call it.
// see next route with the usual callback definition showing it is a function definition
// rather than a call
router.get('/authors', author_controller.author_list);

// route with callback declaration(a bit messier than the previous one but I prefer this because
// it separates the controller functions from the management of the routes, i.e. returning to the
client )
router.get(‘/authors’, function (req, res){ // NOTE: this is a definition rather than a call!
              author_list(req, res, function (err, data)
                  if (!err){ res.writeHead(200, { "Content-Type": "application/json"});
                              res.end(JSON.stringify(data)); …
```

-

Organising Models



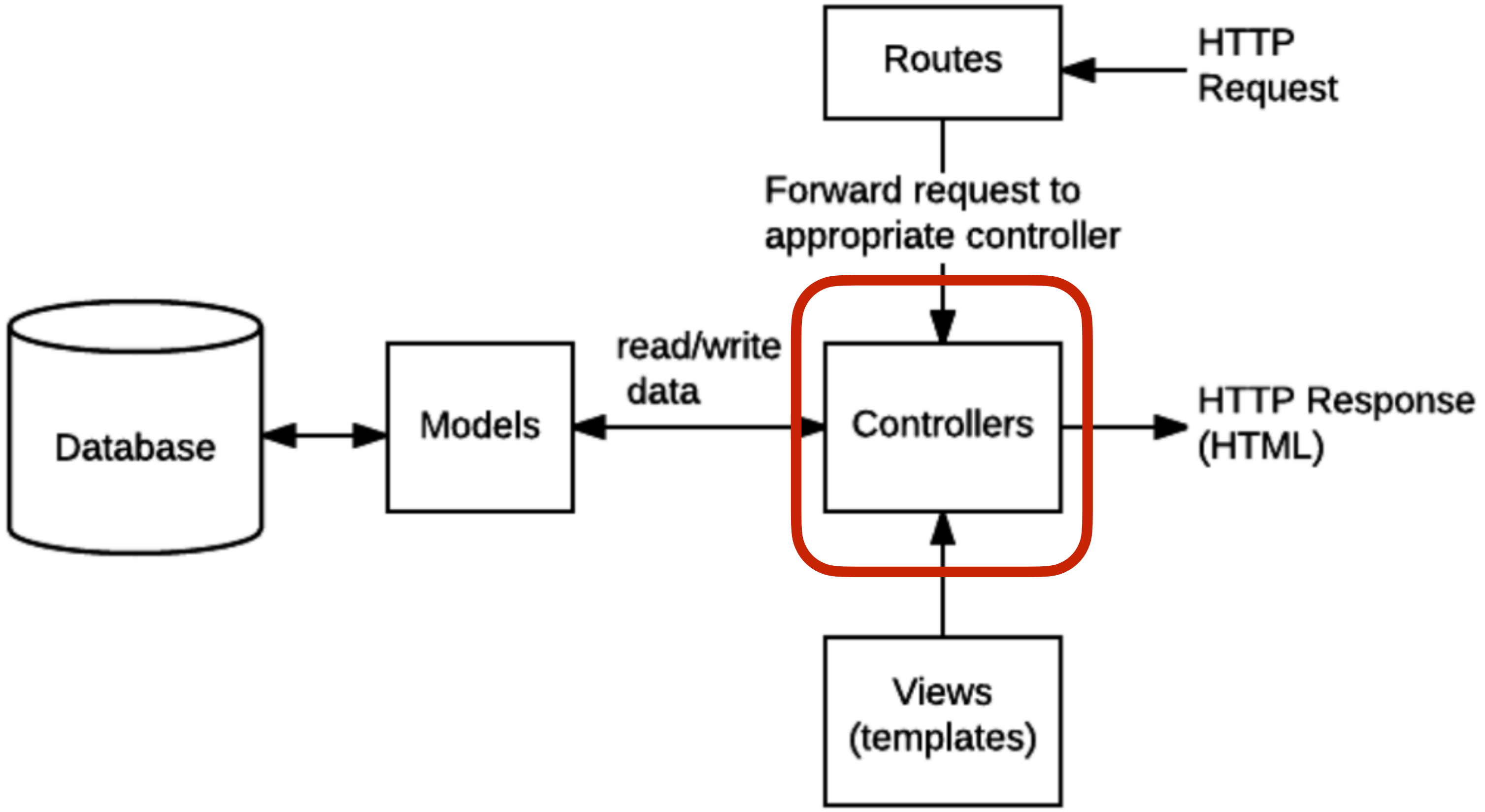
- It is recommended that you define just one schema + model per file
  - and then export the model

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var Character = new Schema({
        first_name: {type: String, required: true, max: 100},
        family_name: {type: String, required: true, max: 100},
        dob: {type: Number},
        whatever: {type: String} });

var characterModel = mongoose.model('Character', Character );

module.exports = characterModel;
```

we will see how to import the model in a controller in a few slides

UNIVERSITÀ DI TORINO

Typical project organisation



https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/routes

Controllers

- Controllers are the route-handler callback functions
  - it is suggest not to insert too much code into the routes file(s)
    - so to keep the code clean and separated

UNIVERSITÀ DI TORINO

Controllers

- They Import the models and use the model as an object, e.g.:
  - then they define the exported functions to be used in the routes
  - being a module, **we must export** the functions otherwise they will not be visible outside the module

```
// the controller must import the model(s) it works on
var Author = require('../models/author');

// Display list of all Authors. Remember to export the function outside the module
exports.author_list = function(req, res) {
        // here we will make operations on the database and return the data
        // for example here we could have a find operation to retrieve the
        authors list
    res.send('NOT IMPLEMENTED: Author list');
};
```

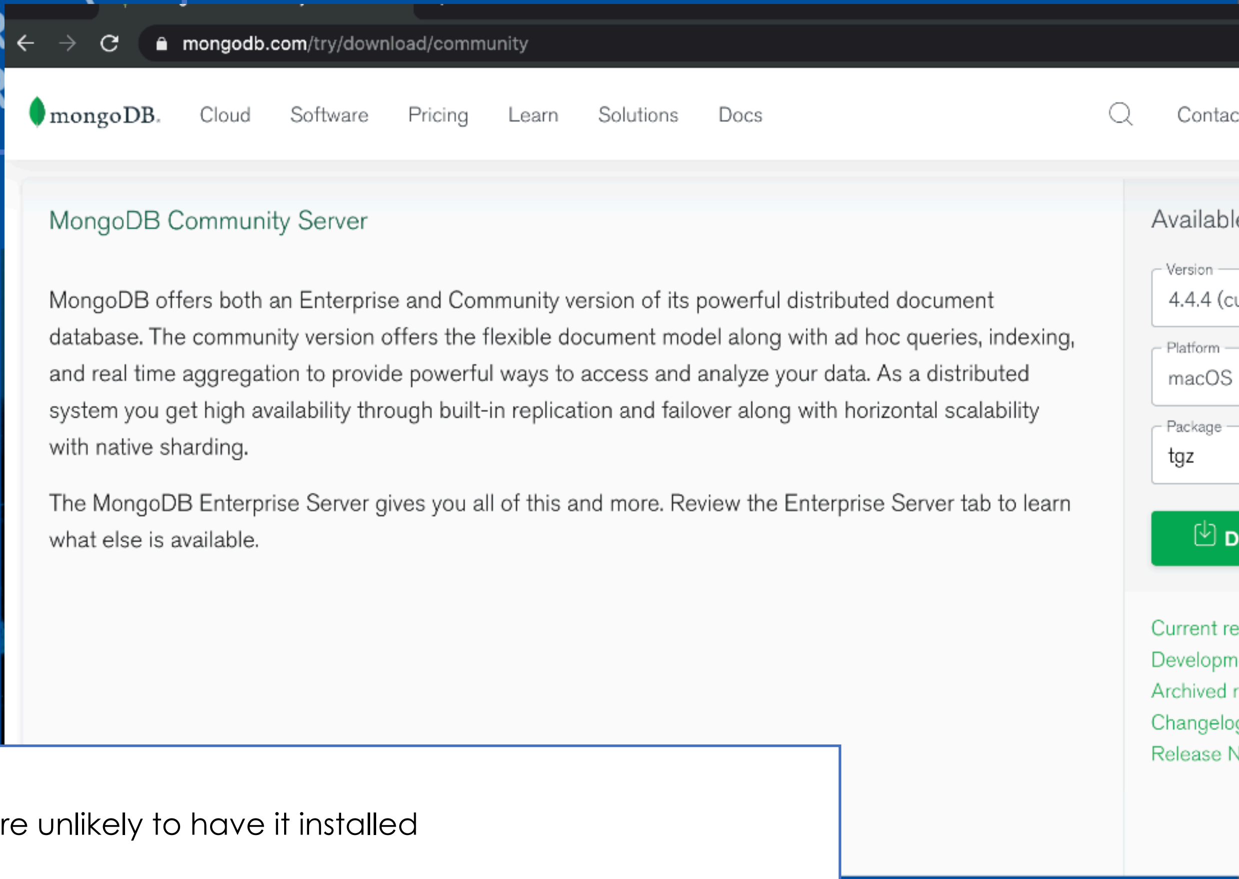
The database declaration

- create a js file called database.js under the folder /databases

```
const mongoose = require('mongoose');

//The URL which will be queried. Run "mongod.exe" for this to connect
//var url = 'mongodb://localhost:27017/test';
const mongoDB = 'mongodb://localhost:27017/characters';
mongoose.Promise = global.Promise;
connection = mongoose.connect(mongoDB, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
    checkServerIdentity: false,
})
    .then(() => {
        console.log('connection to mongodb worked!');
    })
    .catch((error) => {
        console.log('connection to mongodb did not work! ' + JSON.stringify(error));
    });
```

this will connect to the Mongo server and will create the characters database (if not existent)

Download MongoDB


mongodb.com/try/download/community

MongoDB Community Server

MongoDB offers both an Enterprise and Community version of its powerful distributed document database. The community version offers the flexible document model along with ad hoc queries, indexing, and real time aggregation to provide powerful ways to access and analyze your data. As a distributed system you get high availability through built-in replication and failover along with horizontal scalability with native sharding.

The MongoDB Enterprise Server gives you all of this and more. Review the Enterprise Server tab to learn what else is available.

Before the next lab class onto your own computers! The lab computers are unlikely to have it installed

Download the community server
https://www.mongodb.com/try/download/community

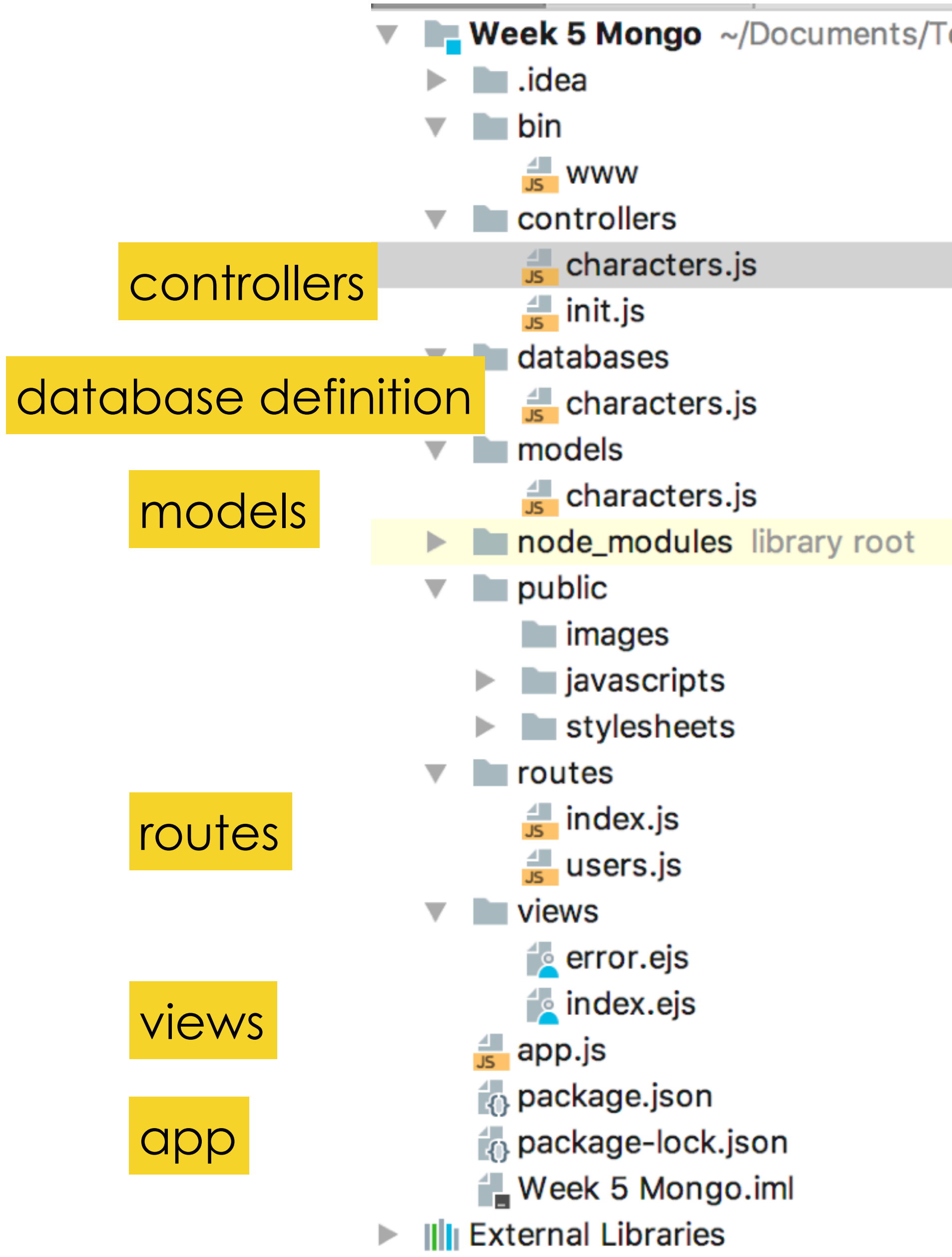
UNIVERSITÀ DI TORINO

Finally…

UNIVERSITÀ DI TORINO

Revision

- Understand how MongoDb works
- Being able to create a small nodeJS program querying a networked Mongo DB
- Have a clear understanding of the required organisation of the code for the Mongo DB in a nodes environment



```
▼ Week 5 Mongo ~/Documents/T
  ▶ .idea
  ▼ bin
      www
  ▼ controllers
      characters.js
      init.js
  ▼ databases
      characters.js
  ▼ models
      characters.js
  ▶ node_modules library root
  ▼ public
      images
    ▶ javascripts
    ▶ stylesheets
  ▼ routes
      index.js
      users.js
  ▼ views
      error.ejs
      index.ejs
    app.js
    package.json
    package-lock.json
    Week 5 Mongo.iml
▶ External Libraries
```

UNIVERSITÀ DI TORINO

Questions?

© Prof. Fabio Cravegna, Università di Torino