

# Express and Mongoose Extras

---

Olivier Liechti & Simon Oulevay  
COMEM Web Services 2016

heig-vd

Haute Ecole d'Ingénierie et de Gestion  
du Canton de Vaud



# Parsing request data

# Retrieving request data

How can I retrieve the HTTP **verb**, the **URL parameters**, the **query parameters**, the **headers** and the **request body**?

```
POST /req/a/b?page=3&pageSize=30&select=foo&select=bar&select=baz HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 55
Content-Type: application/json
Host: localhost:3000
User-Agent: HTTPie/0.9.2
Authorization: Basic Zm9vOmJhcgo=

{
  "age": "24",
  "name": {
    "first": "John",
    "last": "Doe"
  }
}
```

# Retrieving the HTTP verb/method

```
POST /test/a/b?page=3&pageSize=30&select=foo&select=bar&select=baz HTTP/1.1
```

```
Accept: application/json
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 55
Content-Type: application/json
Host: localhost:3000
User-Agent: HTTPie/0.9.2
Authorization: Basic Zm9vOmJhcgo=
```

```
{
  "age": "24",
  "name": {
    "first": "John",
    "last": "Doe"
  }
}
```

```
router.all('/test/:param1/:param2', function(req, res, next) {
  console.log(req.method);
});
```

"POST"

# Retrieving the full path

```
POST /test/a/b?page=3&pageSize=30&select=foo&select=bar&select=baz HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 55
Content-Type: application/json
Host: localhost:3000
User-Agent: HTTPie/0.9.2
Authorization: Basic Zm9vOmJhcgo=

{
  "age": "24",
  "name": {
    "first": "John",
    "last": "Doe"
  }
}
```

```
router.all('/test/:param1/:param2', function(req, res, next) {
  console.log(req.path);
});
```

"/test/a/b"

# Retrieving the URL/path parameters

```
POST /test/a/b?page=3&pageSize=30&select=foo&select=bar&select=baz HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 55
Content-Type: application/json
Host: localhost:3000
User-Agent: HTTPie/0.9.2
Authorization: Basic Zm9vOmJhcgo=
```

```
{
  "age": "24",
  "name": {
    "first": "John",
    "last": "Doe"
  }
}
```

```
router.all('/test/:param1/:param2', function(req, res, next) {
  console.log(req.params);
});
```

```
{
  "param1": "a",
  "param2": "b"
}
```

# Retrieving the query parameters

```
POST /test/a/b?page=3&pageSize=30&select=foo&select=bar&select=baz HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 55
Content-Type: application/json
Host: localhost:3000
User-Agent: HTTPie/0.9.2
Authorization: Basic Zm9vOmJhcgo=

{
  "age": "24",
  "name": {
    "first": "John",
    "last": "Doe"
  }
}
```

```
router.all('/test/:param1/:param2', function(req, res, next) {
  console.log(req.query);
});
```

```
{
  "page": 3,
  "pageSize": 30,
  "select": [ "foo", "bar", "baz" ]
}
```

# Retrieving the headers

```
POST /test/a/b?page=3&pageSize=30&select=foo&select=bar&select=baz HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 55
Content-Type: application/json
Host: localhost:3000
User-Agent: HTTPie/0.9.2
Authorization: Basic Zm9vOmJhcgo=

{
  "age": "24",
  "name": {
    "first": "John",
    "last": "Doe"
  }
}
```

```
router.all('/test/:param1/:param2', function(req, res, next) {
  console.log(req.headers);
});
```

```
{
  "accept": "application/json"
  "content-type": "application/json",
  "authorization": "Basic Zm9vOmJhcgo="
  ...
}
```

**Warning!** header  
names are normalised  
to lowercase.



# Retrieving the headers

```
POST /test/a/b?page=3&pageSize=30&select=foo&select=bar&select=baz HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 55
Content-Type: application/json
Host: localhost:3000
User-Agent: HTTPie/0.9.2
Authorization: Basic Zm9v0mJhcgo=

{
  "age": "24",
  "name": {
    "first": "John",
    "last": "Doe"
  }
}
```

```
router.all('/test/:param1/:param2', function(req, res, next) {
  console.log(req.get("Authorization"));
  console.log(req.get("authorization"));
});
```

"Basic Zm9v0mJhcgo="

Use **req.get** for  
case-insensitive  
retrieval.

# Retrieving the body

```
POST /test/a/b?page=3&pageSize=30&select=foo&select=bar&select=baz HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 55
Content-Type: application/json
Host: localhost:3000
User-Agent: HTTPie/0.9.2
Authorization: Basic Zm9vOmJhcgo=
```

```
{
  "age": "24",
  "name": {
    "first": "John",
    "last": "Doe"
  }
}
```


```
router.all('/test/:param1/:param2', function(req, res, next) {
  console.log(req.body);
});
```

```
{
  "age": 24,
  "name": { "first": "John", "last": "Doe" }
}
```

# Example

This route will parse and return all the request data in the response.

Match any HTTP verb



```
router.all('/test/:param1/:param2', function(req, res, next) {  
  res.send({  
    method: req.method,  
    path: req.path,  
    params: req.params,  
    query: req.query,  
    headers: req.headers,  
    body: req.body  
  });  
});
```

<https://morning-reef-58678.herokuapp.com/test/a/b?page=3&select=foo&select=bar>

express  
web application  
framework for  
node

# Middleware & Routers

# Avoiding repetition with middleware

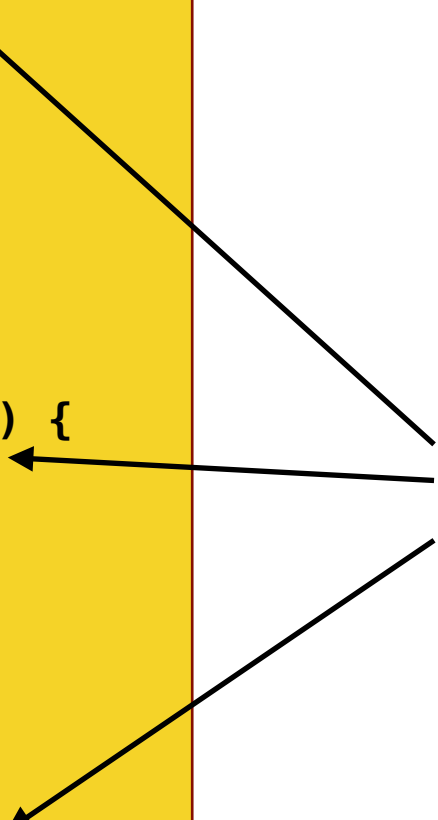
There is **code duplication** in this controller:

```
// GET /api/books/:id
router.get('/:id', function(req, res, next) {
  Book.findById(req.params.id, function(err, book) {
    if (err) { ... }
    else if (!book) { ... }
    // send the book
  });
});

// PUT /api/books/:id
router.put('/:id', function(req, res, next) {
  Book.findById(req.params.id, function(err, book) {
    if (err) { ... }
    else if (!book) { ... }
    // update the book
  });
});

// DELETE /api/books/:id
router.delete('/:id', function(req, res, next) {
  Book.findById(req.params.id, function(err, book) {
    if (err) { ... }
    else if (!book) { ... }
    // delete the book
  });
});
```

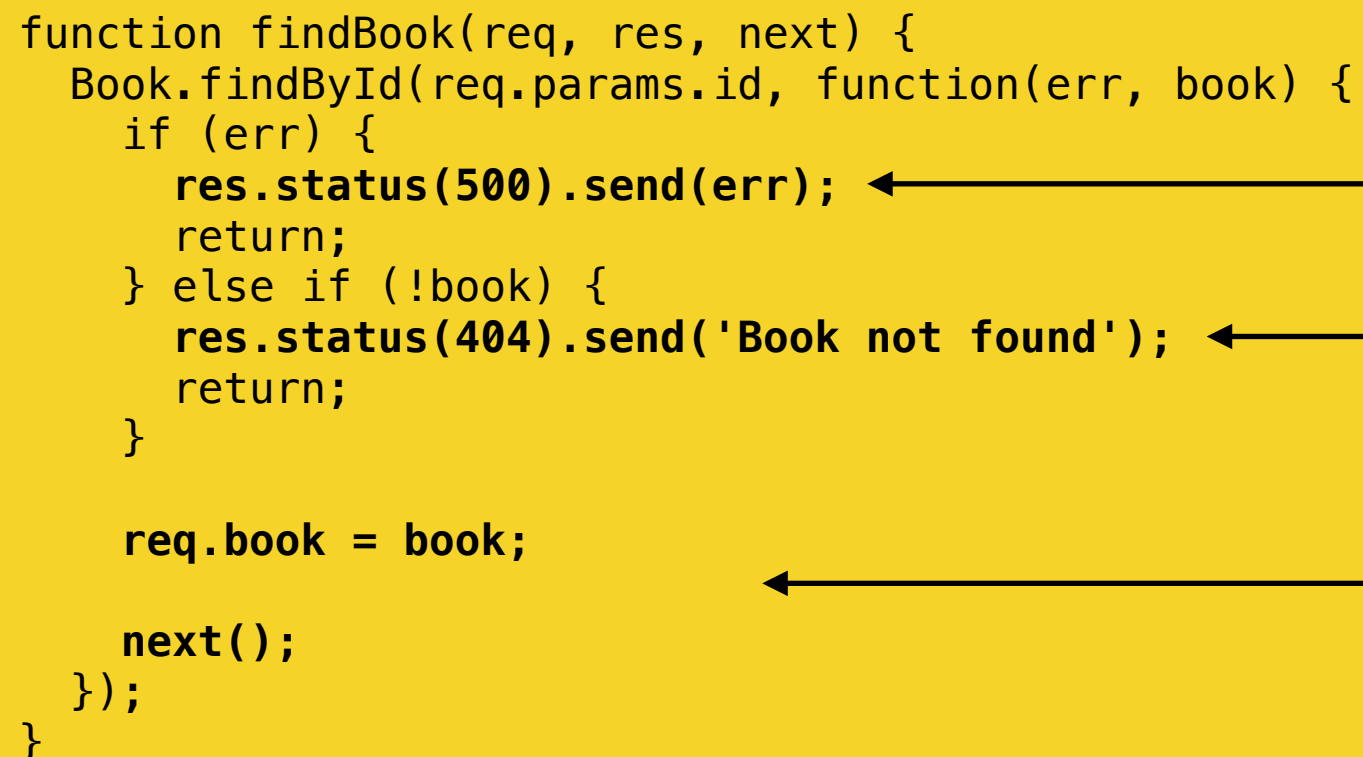
The code to find a book and check for errors is repeated three times.



# Avoiding repetition with middleware

Let's write a **middleware function** that does the job:

This is a middleware function, like all your routes.



```
function findBook(req, res, next) {  
  Book.findById(req.params.id, function(err, book) {  
    if (err) {  
      res.status(500).send(err);  
      return;  
    } else if (!book) {  
      res.status(404).send('Book not found');  
      return;  
    }  
  
    req.book = book;  
  
    next();  
  });  
}
```

Handle unexpected errors.

Handle unknown books.

Otherwise, all is good.

**Store** the book in the **request** object  
and forward the request to the **next**  
**middleware**.

# Avoiding repetition with middleware

A **router** can execute **multiple middleware functions** for one path.

```
// GET /api/books/:id
router.get('/:id', findBook, function(req, res, next) {
  // send the book (available in req.book)
});

// PUT /api/books/:id
router.put('/:id', findBook, function(req, res, next) {
  // update the book (available in req.book)
});

// DELETE /api/books/:id
router.delete('/:id', findBook, function(req, res, next) {
  // delete the book (available in req.book)
});
```

Use the **req.book** variable that you stored in the previous middleware function.

# mongoose

elegant **mongodb** object modeling for **node.js**

## Advanced Mongoose

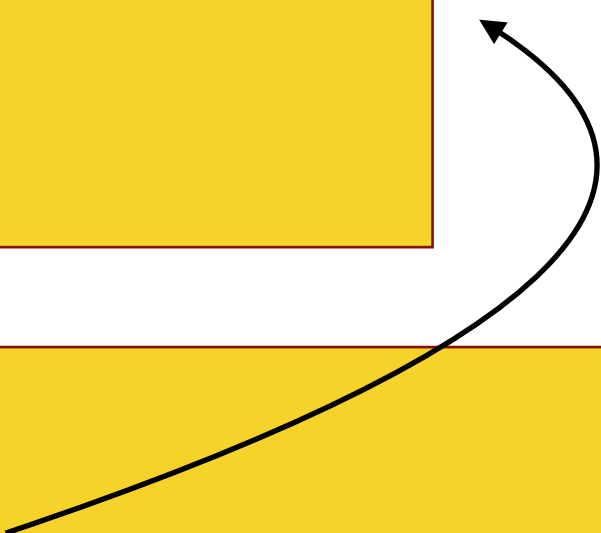


# Sample models

Let's look at more advanced Mongoose queries. Our data model contains **publishers** and **books**. Publishers have a one-to-many relationship to books.

```
var PublisherSchema = new Schema({  
  name: { type: String, required: true },  
  addresses: [  
    {  
      city: { type: String, required: true },  
      street: String  
    }  
  ]  
});
```

```
var BookSchema = new Schema({  
  title: { type: String, required: true },  
  format: String,  
  publisher: { type: Schema.Types.ObjectId, ref: 'Publisher', required: true }  
});
```



# Writing dynamic filters (1)

Finding all books is great, but what about adding some **filters**?

```
// GET /api/books
router.get('/', function(req, res, next) {

  // Find all books.
  Book.find(function(err, books) {
    if (err) {
      res.status(500).send(err);
      return;
    }

    res.send(books);
  });
});
```

What if I want to filter by publisher or book format?

**/api/books?publisher=160983&format=Hardcover**

# Writing dynamic filters (2)

We can use **req.query** and pass conditions to **find**.  
**/api/books?publisher=160983&format=Hardcover**

```
var criteria = {};  
  
// Filter by publisher.  
if (req.query.publisher) {  
  criteria.publisher = req.query.publisher;  
}  
  
// Filter by format.  
if (req.query.format) {  
  criteria.format = req.query.format;  
}  
  
// Find all matching books.  
Book.find(criteria, function(err, books) {  
  if (err) {  
    res.status(500).send(err);  
    return;  
  }  
  
  res.send(books);  
});
```

Give the search criteria to the find.

# Writing dynamic filters (3)

What if I want to find all paperback and hardcover books?

`/api/books?format=Hardcover&format=Paperback`

If the **format** parameter is an array, use the MongoDB **\$in** operator to find all books that have one of those formats.

```
// Filter by format.  
if (typeof(req.query.format) == "object" && req.query.format.length) {  
  criteria.format = { $in: req.query.format };  
} else if (req.query.format) {  
  criteria.format = req.query.format;  
}
```

Otherwise, use an exact match like before.

# Population (1)

---

If you use **ref**, Mongoose knows that this is a reference to another model.

```
var BookSchema = new Schema({  
  title: { type: String, required: true },  
  format: String,  
  publisher: { type: Schema.Types.ObjectId, ref: 'Publisher', required: true }  
});
```

<http://mongoosejs.com/docs/populate.html>

# Population (2)

If you use **ref**, Mongoose knows that this is a reference to another model.

```
var query = Book
  .find(criteria)
  .sort('title')
  .skip(offset)
  .limit(limit);

if (req.query.embed == 'publisher') {
  query.populate('publisher');
}

query.exec(function(err, books) {
  if (err) {
    res.status(500).send(err);
    return;
  }

  res.send(books);
});
```

By default, only the **publisher ID** is serialized. But we also allow the API user to supply the **embed** parameter to indicate that he would like to retrieve the **whole publisher object**.

In this case, we call Mongoose's **populate** method. Since we have defined **publisher** as a **reference**, Mongoose will make the query to retrieve the publisher for us.

# Population (3)

Without the embed query parameter, the response doesn't change.

```
var query = Book
  .find(criteria)
  .sort('title')
  .skip(offset)
  .limit(limit);

if (req.query.embed == 'publisher') {
  query.populate('publisher');
}

query.exec(function(err, books) {
  if (err) {
    res.status(500).send(err);
    return;
  }

  res.send(books);
});
```

**GET /books HTTP/1.1**

**HTTP/1.1 200 OK**  
Content-type: application/json

```
[
  {
    "_id": "3orv8nrg",
    "title": "A Tale of Two Cities",
    "publisher": "0a83u4cn"
  }
]
```

# Population (4)

With the **embed** query parameter, Mongoose will **replace the ID with the publisher object**, and it will be sent in the response.

```
var query = Book
  .find(criteria)
  .sort('title')
  .skip(offset)
  .limit(limit);

if (req.query.embed == 'publisher') {
  query.populate('publisher');
}

query.exec(function(err, books) {
  if (err) {
    res.status(500).send(err);
    return;
  }

  res.send(books);
});
```

GET /books?embed=publisher HTTP/1.1

```
HTTP/1.1 200 OK
Content-type: application/json

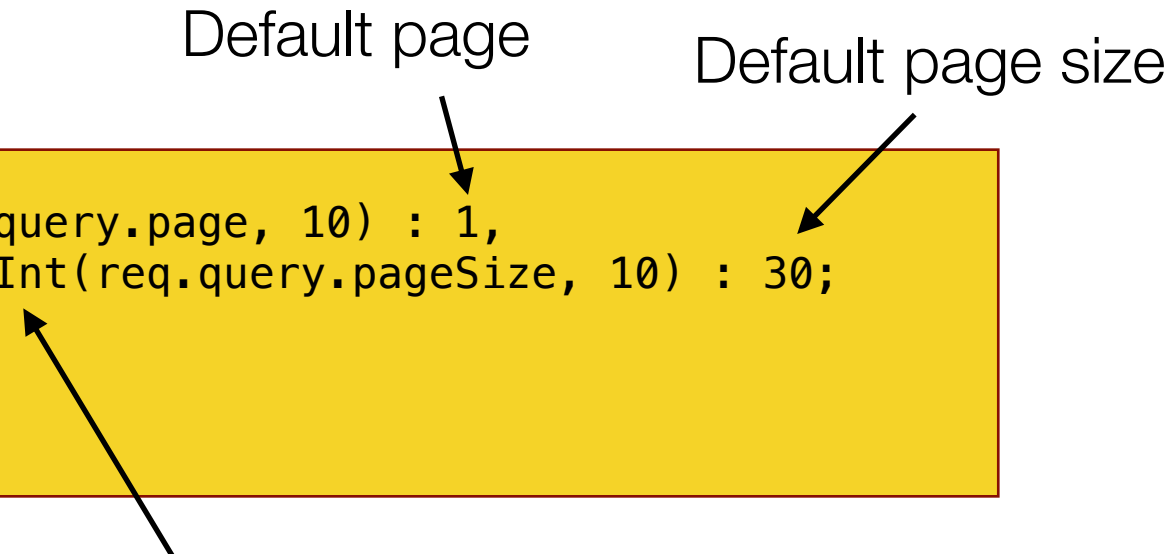
[
  {
    "_id": "3orv8nrg",
    "title": "A Tale of Two Cities",
    "publisher": {
      "_id": "0a83u4cn",
      "name": "Harper Collins",
      "addresses": []
    }
  }
]
```



# Pagination (1)

We'll implement simple pagination. First, let's retrieve the **page** and **page size** from the request data. You also need to convert them to **offset** and **limit** (like we use in SQL).

```
var page = req.query.page ? parseInt(req.query.page, 10) : 1,  
    pageSize = req.query.pageSize ? parseInt(req.query.pageSize, 10) : 30;  
  
var offset = (page - 1) * pageSize,  
    limit = pageSize;
```



Query parameters are always **strings**, so you must parse them to **integers** manually.

# Pagination (2)

Before making your query, you need to **count** the number of books in order to send that information to the API user. There are two things you can count, the **total** number of books (without filters), and the number of books **matching** the filters.

```
Book.count(function(err, totalCount) {  
  if (err) {  
    res.status(500).send(err);  
    return;  
  }  
  
  Book.count(criteria, function(err, filteredCount) {  
    if (err) {  
      res.status(500).send(err);  
      return;  
    }  
  
    res.set('X-Pagination-Page', page);  
    res.set('X-Pagination-Page-Size', pageSize);  
    res.set('X-Pagination-Total', totalCount);  
    res.set('X-Pagination-Filtered-Total', filteredCount);  
  
    // ...  
  })  
})
```

Count the **total** number of books.

Count only the number of books **matching your search criteria**.

Give this information to your API user in **headers**.

# Pagination (3)

Then you can run your **find** query, passing the search criteria and pagination parameters.

Use **skip** and **limit** for pagination.

```
Book.find(criteria)
  .sort('title')
  .skip(offset)
  .limit(limit)
  .exec(function(err, books) {
    if (err) {
      res.status(500).send(err);
      return;
    }

    res.send(books);
  });
```

Do not forget to **sort** the data. It's difficult to use pagination when the data is not sorted.

# Aggregations (1)

---

I want to retrieve publishers sorted by  
**descending number of books.**

With an SQL database, you would do a **JOIN**  
between the **publishers** and the **books** table, a  
**GROUP BY** on the publishers, and a **COUNT** on  
the books.

But in MongoDB, it's **not possible** to make  
queries on **two collections** at the same time.

# Aggregations (2)

We must **first** calculate the number of books by publishers and **filter, sort and/or paginate** these results:

`countBooks(filters, sorting, pagination)`

```
[  
  { "publisherId": "019urv2", "numberOfBooks": 6 },  
  { "publisherId": "09xba23", "numberOfBooks": 2 },  
  ...  
]
```

**Then** we can retrieve the corresponding publishers:

`findPublishers("019urv2", "09xba23", ...)`

```
[  
  { "_id": "019urv2", "name": "Harper Collins" },  
  { "_id": "09xba23", "name": "Shueisha" },  
  ...  
]
```

# Aggregations (3)

<https://docs.mongodb.org/manual/aggregation/>

Aggregations operations **process data records** and return **computed results**. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. MongoDB provides **three ways** to perform aggregation: the **aggregation pipeline**, the **map-reduce** function, and **single purpose aggregation methods**.

↑  
The **aggregation pipeline** is the preferred method in MongoDB and in Mongoose.

# Aggregations (4)

<https://docs.mongodb.org/manual/reference/operator/aggregation/>

When using the **aggregation pipeline**, you build a pipeline of **stages**. Documents will pass through each stage **in sequence**.

## Stage Operators

In the `db.collection.aggregate` method, pipeline stages appear in an array. Documents pass through the stages in sequence.

```
db.collection.aggregate( [ { <stage> }, ... ] )
```

Name	Description
<code>\$project</code>	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
<code>\$match</code>	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. <code>\$match</code> uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).

# Aggregations (5)

Let's run those **books** through the **aggregation pipeline**:

```
Book.aggregate([
  {
    $match: {
      format: { $in: formats }
    },
  },
  {
    $group: {
      _id: '$publisherId',
      total: { $sum: 1 }
    },
  },
  {
    $sort: { total: -1 }
  },
  {
    $skip: 60
  },
  {
    $limit: 30
  }
], function(err, bookCounts) {
  // ...
});
```

**Match** only books with the expected format.

**Group** by publisher ID and **sum (count)** the number of books.

**Sort** by descending total (number of books).

**Skip** the first 60 elements (for pagination).

**Limit** the result to 30 elements (for pagination).

```
[
  { "_id": "10948", "total": 5 },
  { "_id": "sab01", "total": 3 },
  ...
]
```



# Aggregations (6)

Now we can retrieve the associated **publishers**:

```
var criteria = {  
  _id: { $in: publisherIds }  
};  
  
Publisher.find(criteria, function(err, publishers) {  
  if (err) {  
    res.status(500).send(err);  
    return;  
  }  
  
  var responseBody = [];  
  
  for (var i = 0; i < bookCounts.length; i++) {  
    var result = getPublisher(bookCounts[i]._id, publishers).toJSON();  
    result.numberOfBooks = bookCounts[i].total;  
    responseBody.push(result);  
  }  
  
  res.send(responseBody);  
});
```

**Match** only the publishers we are interested in.

**Find** them with Mongoose.

The response array.

**Serialize** each publisher.

Add the **number of results** to the serialized object.

**Finally**, send the response.

# Aggregations (7)

Of course, do not forget that this is all **asynchronous**.

```
function countBooks(format, callback) {  
  Book.aggregate([  
    ...  
  ], function(err, bookCounts) {  
    if (err) {  
      callback(err);  
    } else {  
      callback(undefined, bookCounts);  
    }  
  });  
}
```

```
router.get('/', function(req, res, next) {  
  var format = req.query.format;
```

```
  countBooks(format, function(err, bookCounts) {  
    // handle error (if any)  
  
    var criteria = ...;  
    Publisher.find(criteria, function(err, publishers) {  
      // handle error (if any)  
      // serialize and send response  
    });  
  });  
});
```

Let's write a function for the first task of **counting** the number of books by publisher. Since it's asynchronous, it should take a **callback function** which we will call later.

If there's an **error**, call the callback function with it.

Or, if all went well, call it with no error and the **result**.

Let's use our shiny new function in our **GET /publishers** route.

**Then**, we find the publishers.

And **finally**, we can send the response.

# More examples...

---

"Cascade delete"

Sub-document resource