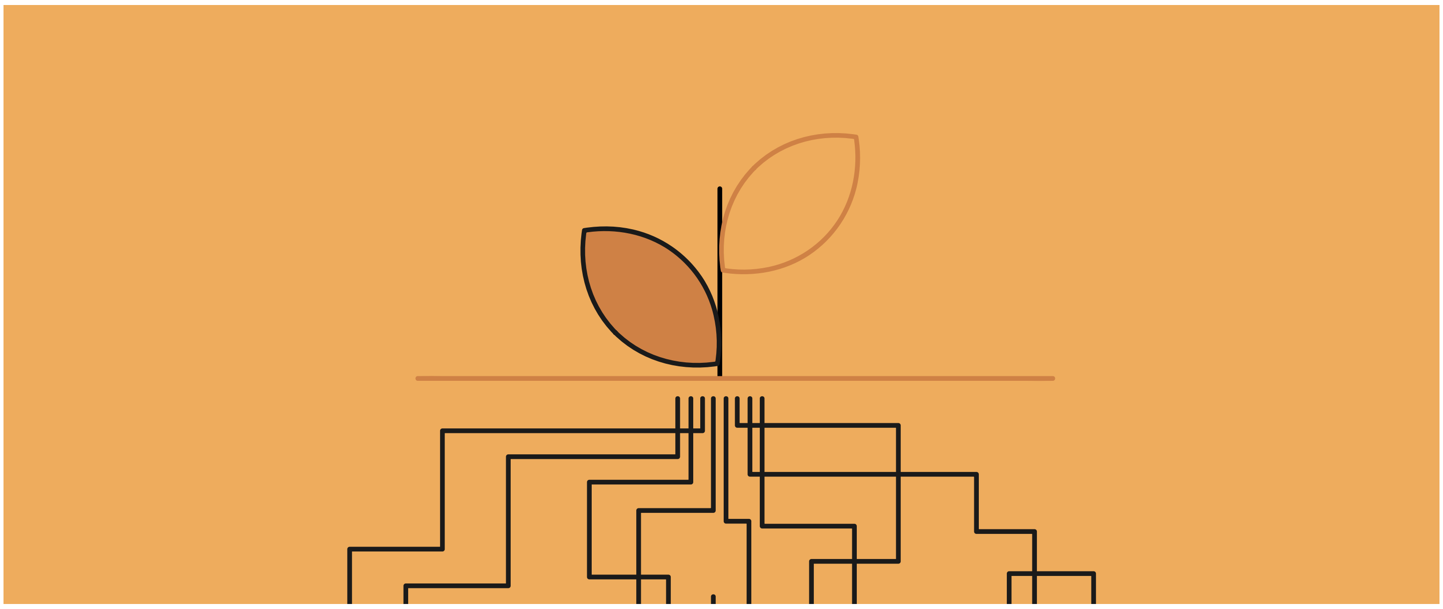
{() => fs}



c

Saving data to MongoDB

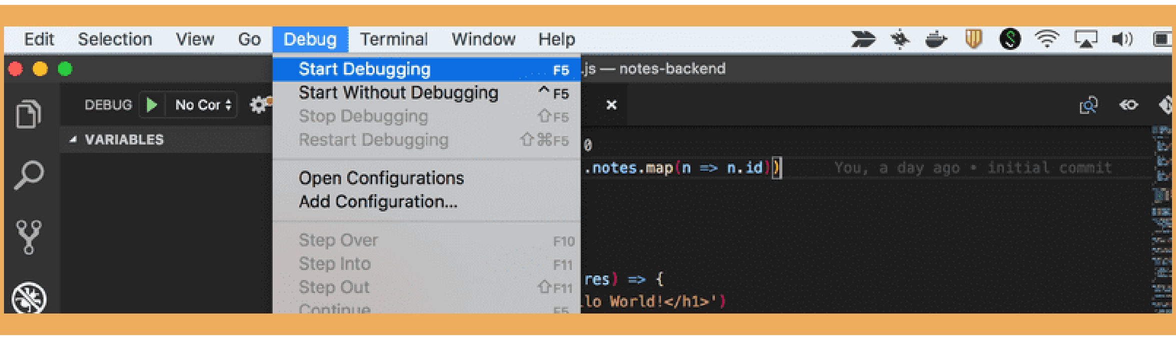
Before we move into the main topic of persisting data in a database, we will take a look at a few different ways of debugging Node applications.

Debugging Node applications

Debugging Node applications is slightly more difficult than debugging JavaScript running in your browser. Printing to the console is a tried and true method, and it's always worth doing. There are people who think that more sophisticated methods should be used instead, but I disagree. Even the world's elite open source developers use this method.

Visual Studio Code

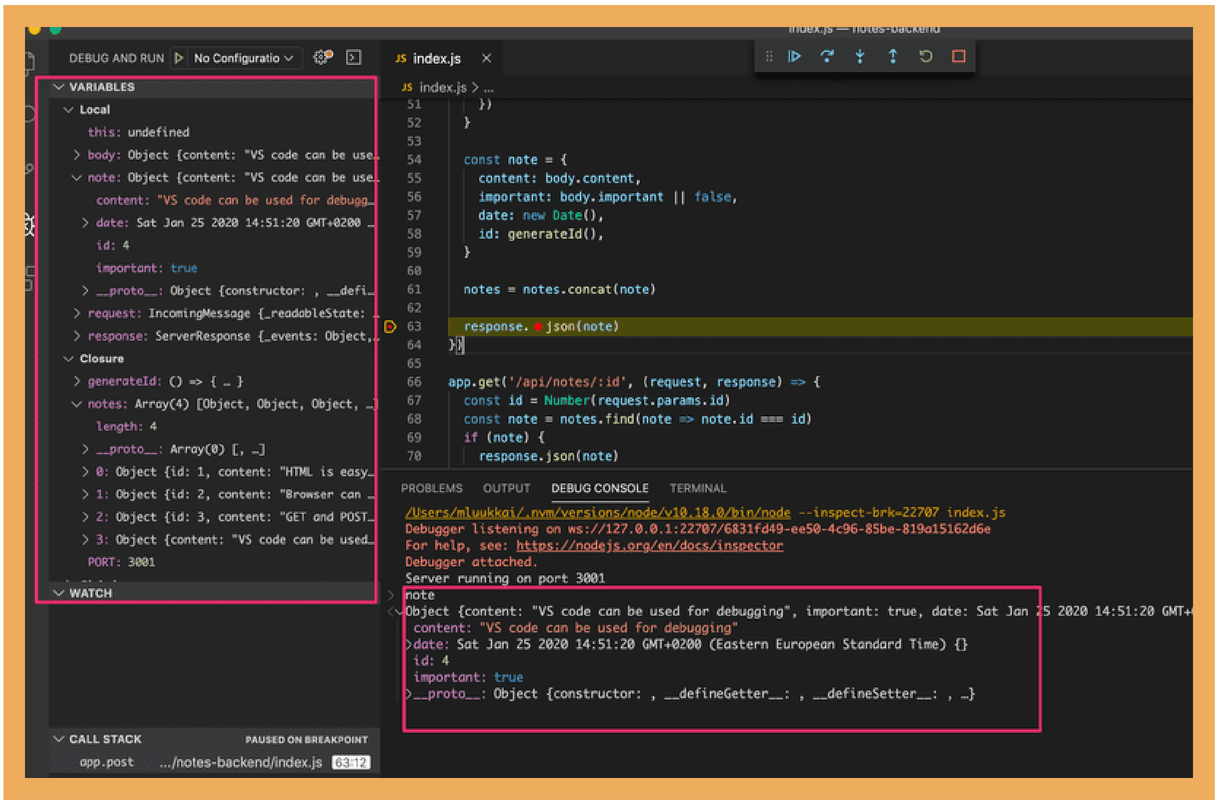
The Visual Studio Code debugger can be useful in some situations. You can launch the application in debugging mode like this:



```
.notes.map(n => n.id))      You, a day ago • initial commit
res) => {
lo World!</h1>')
```

Note that the application shouldn't be running in another console, otherwise the port will already be in use.

Below you can see a screenshot where the code execution has been paused in the middle of saving a new note:



```
VARIABLES
  Local
    this: undefined
  > body: Object {content: "VS code can be use…
  v note: Object {content: "VS code can be use…
      content: "VS code can be used for debugg…
    > date: Sat Jan 25 2020 14:51:20 GMT+0200 …
      id: 4
      important: true
    > __proto__: Object {constructor: , __defi…
  > request: IncomingMessage {_readableState: …
  > response: ServerResponse {_events: Object,…
  Closure
  > generateId: () => { … }
  v notes: Array(4) [Object, Object, Object, …]
      length: 4
    > __proto__: Array(0) [, …]
    > 0: Object {id: 1, content: "HTML is easy…
    > 1: Object {id: 2, content: "Browser can …
    > 2: Object {id: 3, content: "GET and POST…
    > 3: Object {content: "VS code can be used…
    PORT: 3001
WATCH
```

```
51      })
52    }
53
54    const note = {
55      content: body.content,
56      important: body.important || false,
57      date: new Date(),
58      id: generateId(),
59    }
60
61    notes = notes.concat(note)
62
63    response.json(note)
64  })
65
66  app.get('/api/notes/:id', (request, response) => {
67    const id = Number(request.params.id)
68    const note = notes.find(note => note.id === id)
69    if (note) {
70      response.json(note)
```

```
/Users/mluukkai/.nvm/versions/node/v10.18.0/bin/node --inspect-brk=22707 index.js
Debugger listening on ws://127.0.0.1:22707/6831fd49-ee50-4c96-85be-819a15162d6e
For help, see: https://nodejs.org/en/docs/inspector
Debugger attached.
Server running on port 3001
note
Object {content: "VS code can be used for debugging", important: true, date: Sat Jan 25 2020 14:51:20 GMT+
 content: "VS code can be used for debugging"
>date: Sat Jan 25 2020 14:51:20 GMT+0200 (Eastern European Standard Time) {}
 id: 4
 important: true
>__proto__: Object {constructor: , __defineGetter__: , __defineSetter__: , …}
```

The execution has stopped at the *breakpoint* in line 63. In the console you can see the value of the *note* variable. In the top left window you can see other things related to the state of the application.

The arrows at the top can be used for controlling the flow of the debugger.

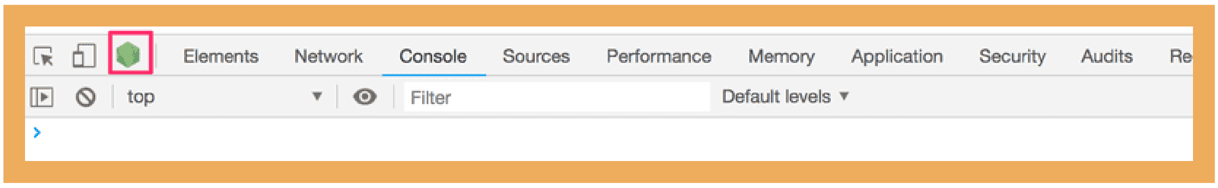
For some reason, I don't use the Visual Studio Code debugger a whole lot.

Chrome dev tools

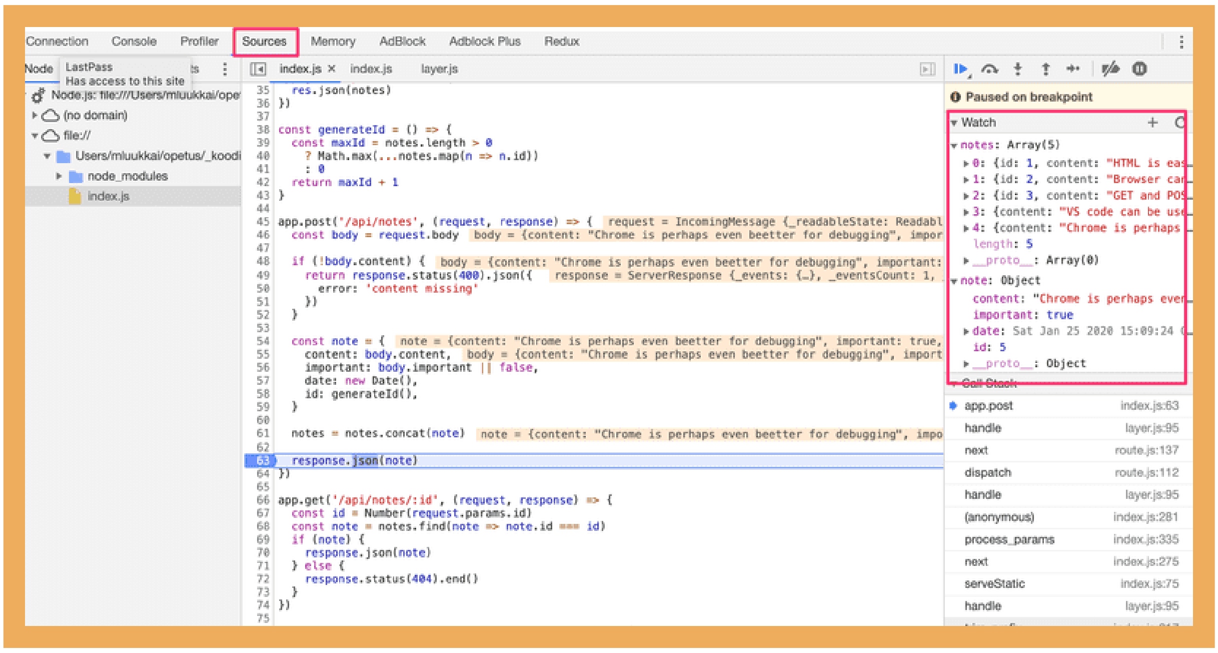
Debugging is also possible with the Chrome developer console by starting your application with the command:

```
node --inspect index.js
```

You can access the debugger by clicking the green icon - the node logo - that appears in the Chrome developer console:



The debugging view works the same way as it did with React applications. The *Sources* tab can be used for setting breakpoints where the execution of the code will be paused.



```
  res.json(notes)
})

const generateId = () => {
  const maxId = notes.length > 0
    ? Math.max(...notes.map(n => n.id))
    : 0
  return maxId + 1
}

app.post('/api/notes', (request, response) => {  request = IncomingMessage {_readableState: Readabl
  const body = request.body  body = {content: "Chrome is perhaps even beetter for debugging", impor

  if (!body.content) {  body = {content: "Chrome is perhaps even beetter for debugging", important:
    return response.status(400).json({   response = ServerResponse {_events: {…}, _eventsCount: 1,
      error: 'content missing'
    })
  }

  const note = {  note = {content: "Chrome is perhaps even beetter for debugging", important: true,
    content: body.content,  body = {content: "Chrome is perhaps even beetter for debugging", import
    important: body.important || false,
    date: new Date(),
    id: generateId(),
  }

  notes = notes.concat(note)  note = {content: "Chrome is perhaps even beetter for debugging", impo

  response.json(note)
})

app.get('/api/notes/:id', (request, response) => {
  const id = Number(request.params.id)
  const note = notes.find(note => note.id === id)
  if (note) {
    response.json(note)
  } else {
    response.status(404).end()
  }
})
```

Paused on breakpoint

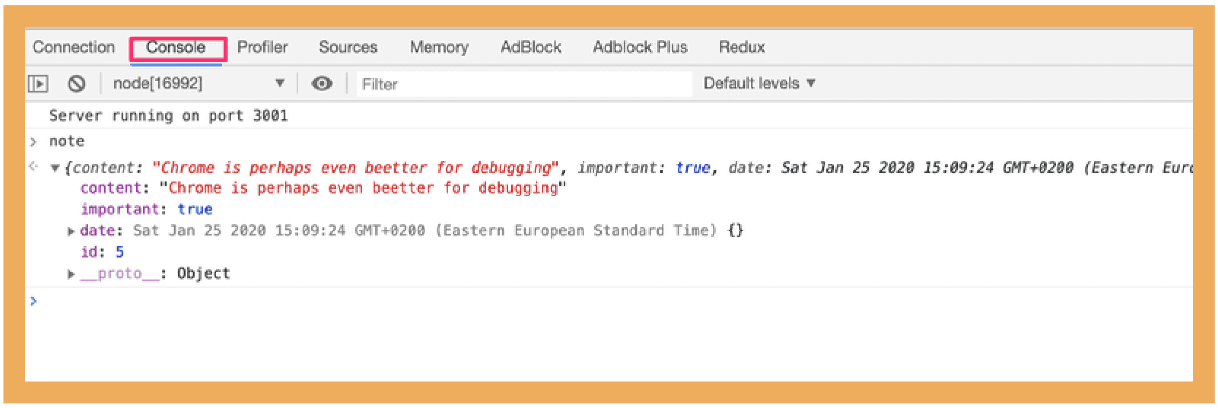
Watch

```
▾notes: Array(5)
  ▸0: {id: 1, content: "HTML is eas…
  ▸1: {id: 2, content: "Browser can…
  ▸2: {id: 3, content: "GET and POS…
  ▸3: {content: "VS code can be use…
  ▸4: {content: "Chrome is perhaps …
   length: 5
  ▸__proto__: Array(0)
▾note: Object
   content: "Chrome is perhaps even…
   important: true
  ▸date: Sat Jan 25 2020 15:09:24 (…
   id: 5
  ▸__proto__: Object
```

Call Stack

```
app.post        index.js:63
handle          layer.js:95
next            route.js:137
dispatch        route.js:112
handle          layer.js:95
(anonymous)     index.js:281
process_params  index.js:335
next            index.js:275
serveStatic     index.js:75
handle          layer.js:95
```

All of the application's *console.log* messages will appear in the *Console* tab of the debugger. You can also inspect values of variables and execute your own JavaScript code.



```
Server running on port 3001
> note
< ▾{content: "Chrome is perhaps even beetter for debugging", important: true, date: Sat Jan 25 2020 15:09:24 GMT+0200 (Eastern Eur
    content: "Chrome is perhaps even beetter for debugging"
    important: true
   ▸date: Sat Jan 25 2020 15:09:24 GMT+0200 (Eastern European Standard Time) {}
    id: 5
   ▸__proto__: Object
>
```

Question everything

Debugging Full Stack applications may seem tricky at first. Soon our application will also have a database in addition to the frontend and backend, and there will be many potential areas for bugs in the application.

When the application "does not work", we have to first figure out where the problem actually occurs. It's very common for the problem to exist in a place where you didn't expect it to, and it can take minutes, hours, or even days before you find the source of the problem.

The key is to be systematic. Since the problem can exist anywhere, *you must question everything*, and eliminate all possibilities one by one. Logging to the console, Postman, debuggers, and experience will help.

When bugs occur, *the worst of all possible strategies* is to continue writing code. It will guarantee that your code will soon have even more bugs, and debugging them will be even more difficult. The stop and fix principle from Toyota Production Systems is very effective in this situation as well.

MongoDB

In order to store our saved notes indefinitely, we need a database. Most of the courses taught at the University of Helsinki use relational databases. In this course we will use MongoDB which is a so-called document database.

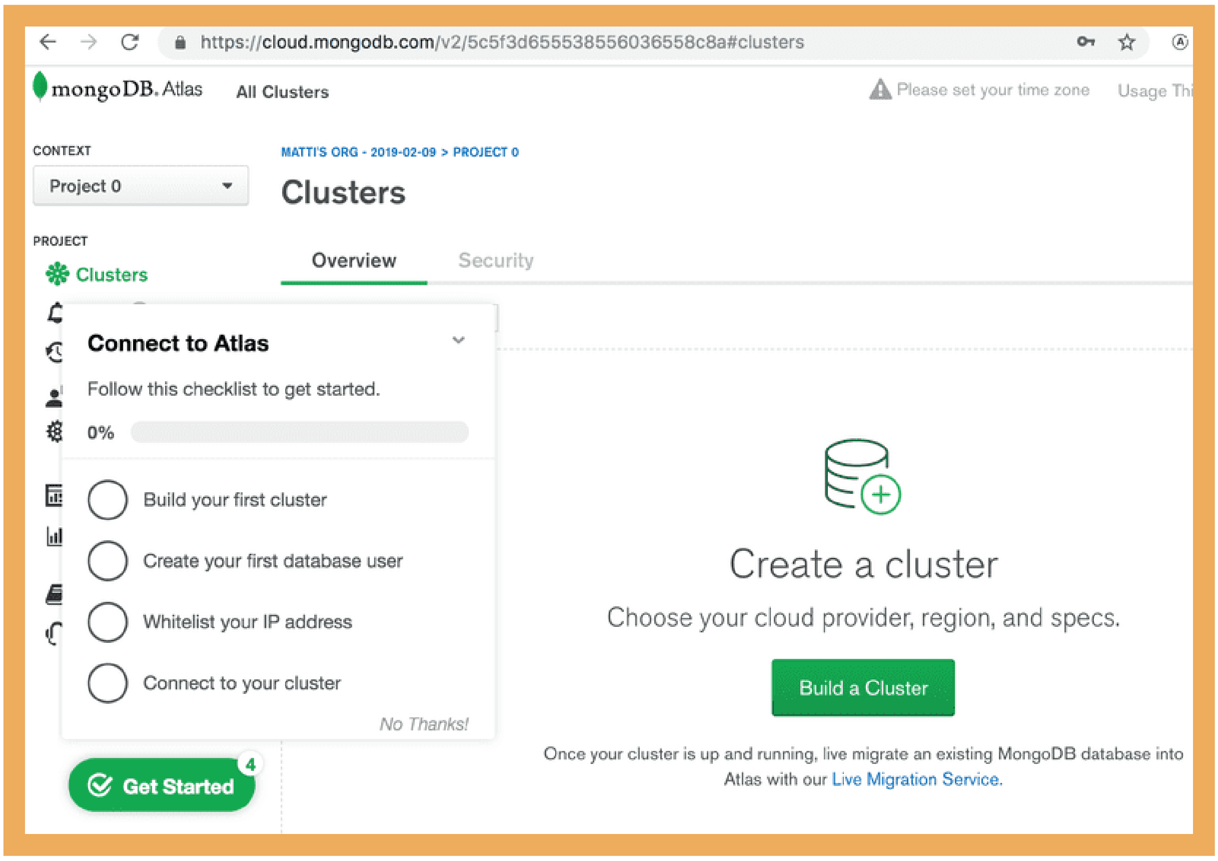
Document databases differ from relational databases in how they organize data as well as the query languages they support. Document databases are usually categorized under the NoSQL umbrella term.

You can read more about document databases and NoSQL from the course material for week 7 of the Introduction to Databases course. Unfortunately the material is currently only available in Finnish.

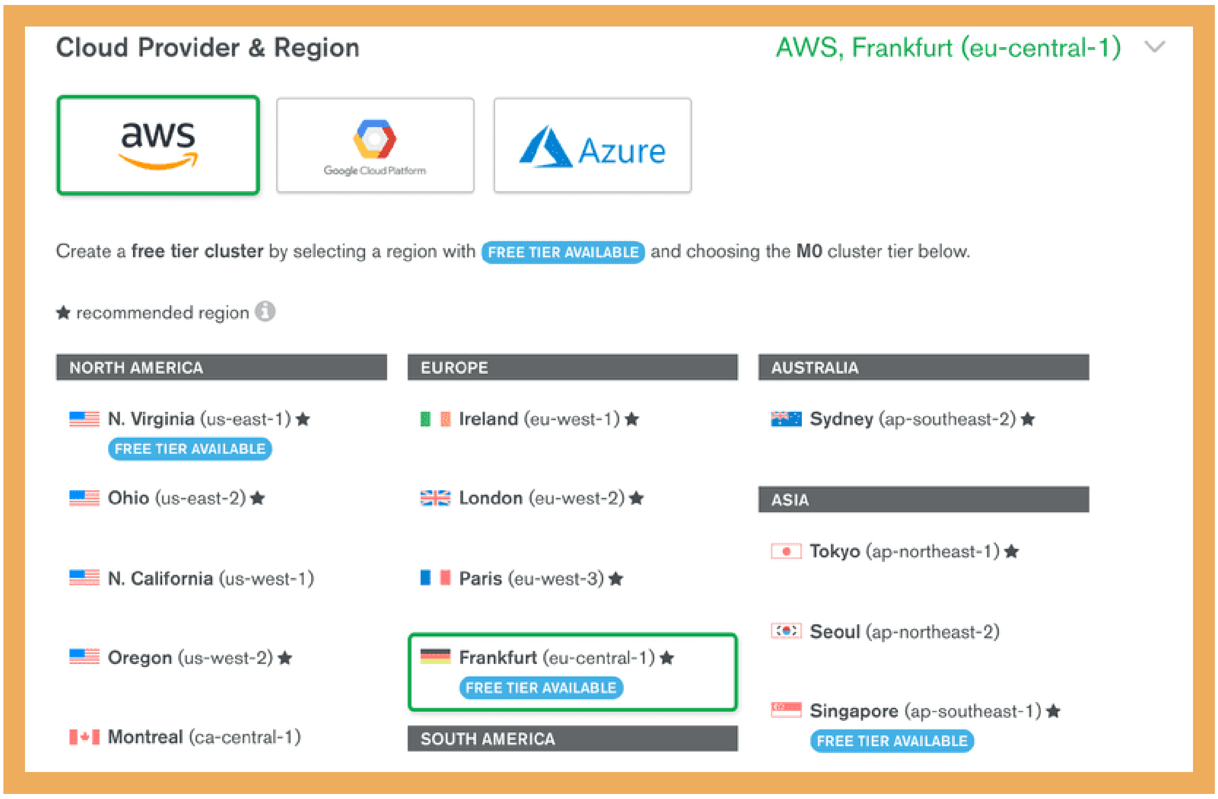
Read now the chapters on collections and documents from the MongoDB manual to get a basic idea on how a document database stores data.

Naturally, you can install and run MongoDB on your own computer. However, the internet is also full of Mongo database services that you can use. Our preferred MongoDB provider in this course will be MongoDB Atlas.

Once you've created and logged into your account, Atlas will recommend creating a cluster:


https://cloud.mongodb.com/v2/5c5f3d655538556036558c8a#clusters
mongoDB. Atlas
All Clusters
Please set your time zone
CONTEXT
Project 0
PROJECT
Clusters
MATTI'S ORG - 2019-02-09 > PROJECT 0
Clusters
Overview
Security
Connect to Atlas
Follow this checklist to get started.
0%
Build your first cluster
Create your first database user
Whitelist your IP address
Connect to your cluster
No Thanks!
Get Started
Create a cluster
Choose your cloud provider, region, and specs.
Build a Cluster
Once your cluster is up and running, live migrate an existing MongoDB database into Atlas with our Live Migration Service.

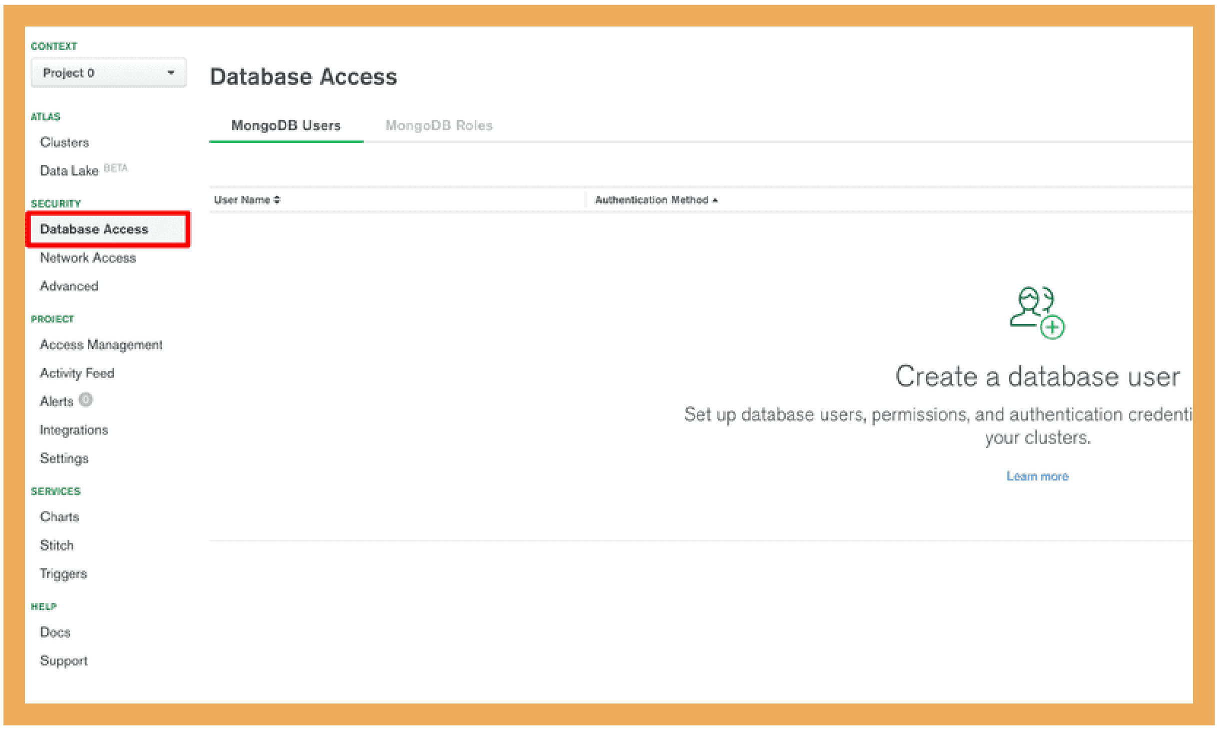
Let's choose *AWS* as the provider and *Frankfurt* as the region, and create a cluster.


Cloud Provider & Region
AWS, Frankfurt (eu-central-1)
aws
Google Cloud Platform
Azure
Create a free tier cluster by selecting a region with FREE TIER AVAILABLE and choosing the M0 cluster tier below.
★ recommended region
NORTH AMERICA
N. Virginia (us-east-1) ★ FREE TIER AVAILABLE
Ohio (us-east-2) ★
N. California (us-west-1)
Oregon (us-west-2) ★
Montreal (ca-central-1)
EUROPE
Ireland (eu-west-1) ★
London (eu-west-2) ★
Paris (eu-west-3) ★
Frankfurt (eu-central-1) ★ FREE TIER AVAILABLE
SOUTH AMERICA
AUSTRALIA
Sydney (ap-southeast-2) ★
ASIA
Tokyo (ap-northeast-1) ★
Seoul (ap-northeast-2)
Singapore (ap-southeast-1) ★ FREE TIER AVAILABLE

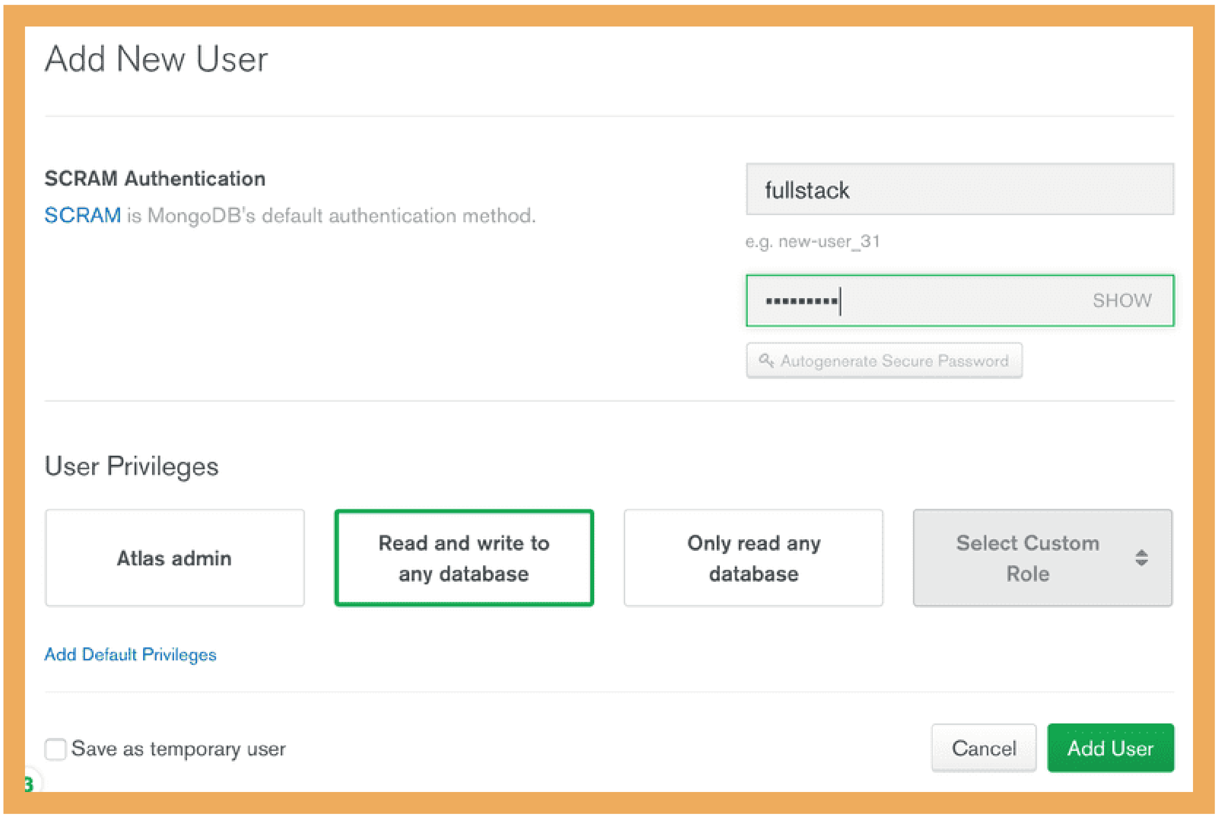
Let's wait for the cluster to be ready for use. This can take approximately 10 minutes.

NB do not continue before the cluster is ready.

Let's use the *database access* tab for creating user credentials for the database. Please note that these are not the same credentials you use for logging into MongoDB Atlas. These will be used for your application to connect to the database.

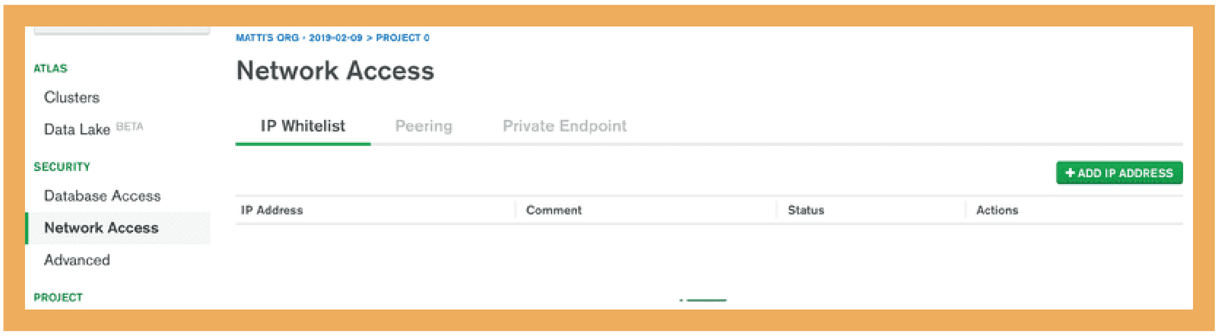


Let's grant the user with permissions to read and write to the databases.

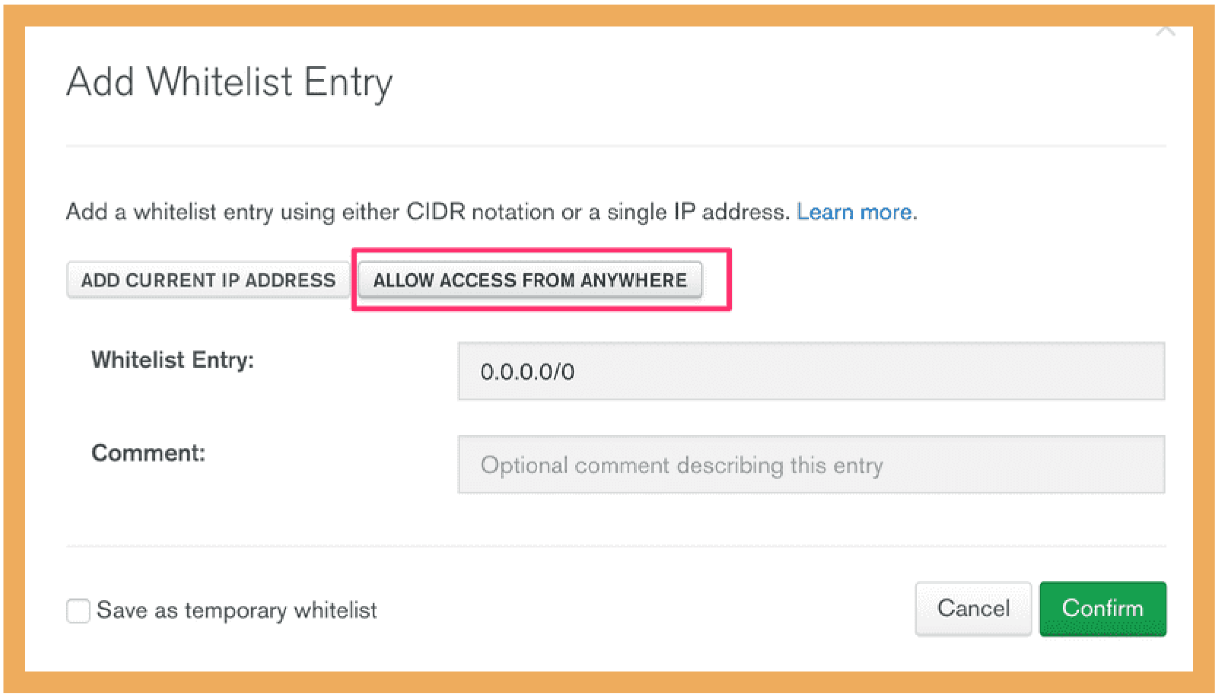


NB: Some people reported the new user credentials not working immediately after creation. In some cases it has taken minutes before the credentials started working.

Next we have to define the IP addresses that are allowed access to the database.



For the sake of simplicity we will allow access from all IP addresses:

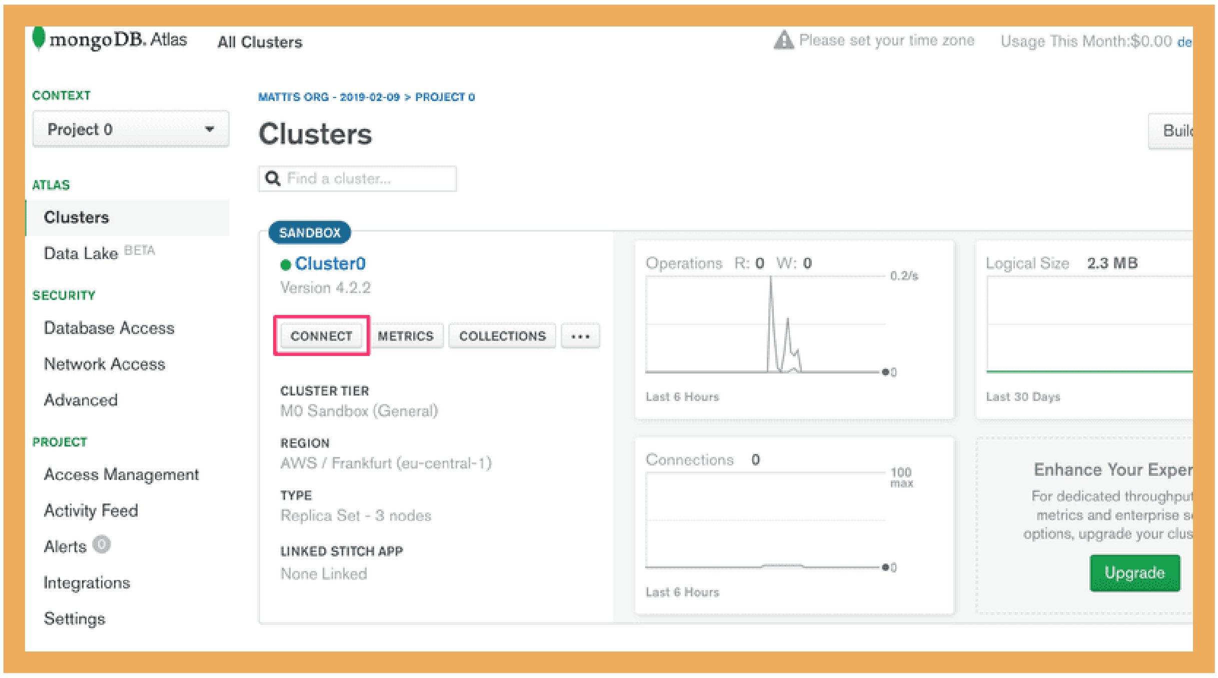


Add a whitelist entry using either CIDR notation or a single IP address. Learn more.

Whitelist Entry: 0.0.0.0/0

Comment: Optional comment describing this entry

Finally we are ready to connect to our database. Start by clicking *connect*:



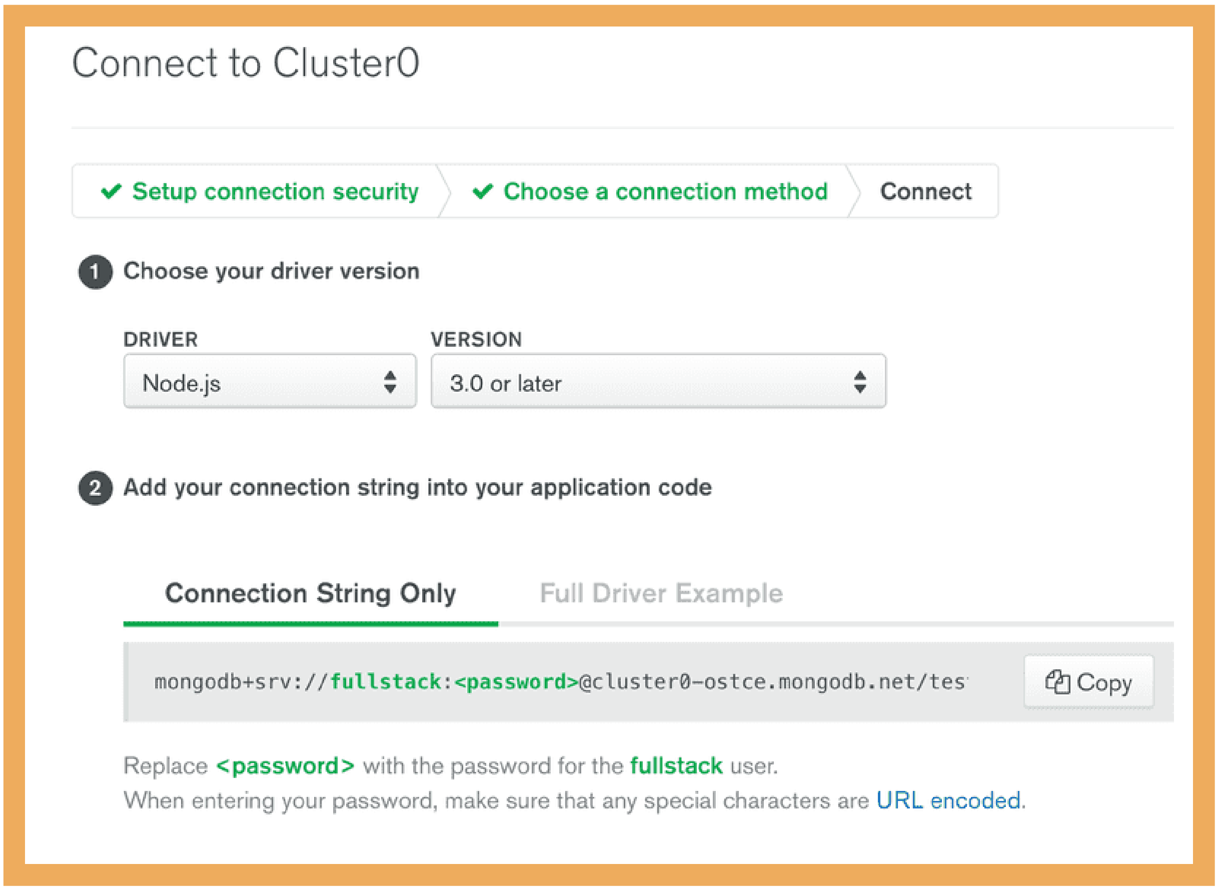
CLUSTER TIER
M0 Sandbox (General)

REGION
AWS / Frankfurt (eu-central-1)

TYPE
Replica Set - 3 nodes

LINKED STITCH APP
None Linked

and choose *Connect your application*:



Connect to Cluster0

✓ Setup connection security ✓ Choose a connection method Connect

1 Choose your driver version

DRIVER: Node.js
VERSION: 3.0 or later

2 Add your connection string into your application code

```
mongodb+srv://fullstack:<password>@cluster0-ostce.mongodb.net/tes
```

Replace <password> with the password for the fullstack user.
When entering your password, make sure that any special characters are URL encoded.

The view displays the *MongoDB URI*, which is the address of the database that we will supply to the MongoDB client library we will add to our application.

The address looks like this:

```
mongodb+srv://fullstack:<PASSWORD>@cluster0-ostce.mongodb.net/test?retryWrites=true
```

We are now ready to use the database.

We could use the database directly from our JavaScript code with the official MongoDb Node.js driver library, but it is quite cumbersome to use. We will instead use the Mongoose library that offers a higher level API.

Mongoose could be described as an *object document mapper* (ODM), and saving JavaScript objects as Mongo documents is straightforward with this library.

Let's install Mongoose:

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

Let's not add any code dealing with Mongo to our backend just yet. Instead, let's make a practice application by creating a new file, *mongo.js*:

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

if (process.argv.length < 3) {
  console.log('Please provide the password as an argument: node mongo.js <password>')
  process.exit(1)
}

const password = process.argv[2]

const url =
  `mongodb+srv://fullstack:${password}@cluster0-ostce.mongodb.net/test?retryWrites=true`

mongoose.connect(url, { useNewUrlParser: true, useUnifiedTopology: true })

const noteSchema = new mongoose.Schema({
  content: String,
  date: Date,
  important: Boolean,
})

const Note = mongoose.model('Note', noteSchema)

const note = new Note({
  content: 'HTML is Easy',
  date: new Date(),
  important: true,
})

note.save().then(result => {
  console.log('note saved!')
  mongoose.connection.close()
})
```

NB: Depending on which region you selected when building your cluster, the *MongoDB URI* may be different from the example provided above. You should verify and use the correct URI that was generated from MongoDB Atlas.

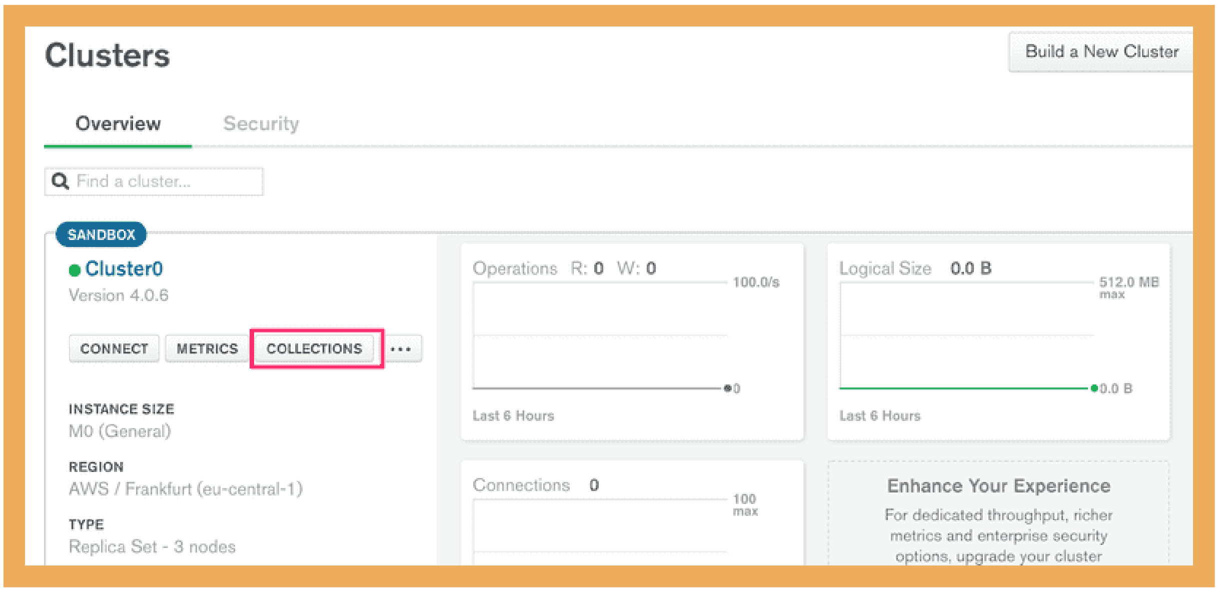
The code also assumes that it will be passed the password from the credentials we created in MongoDB Atlas, as a command line parameter. We can access the command line parameter like this:

```
const password = process.argv[2]
```

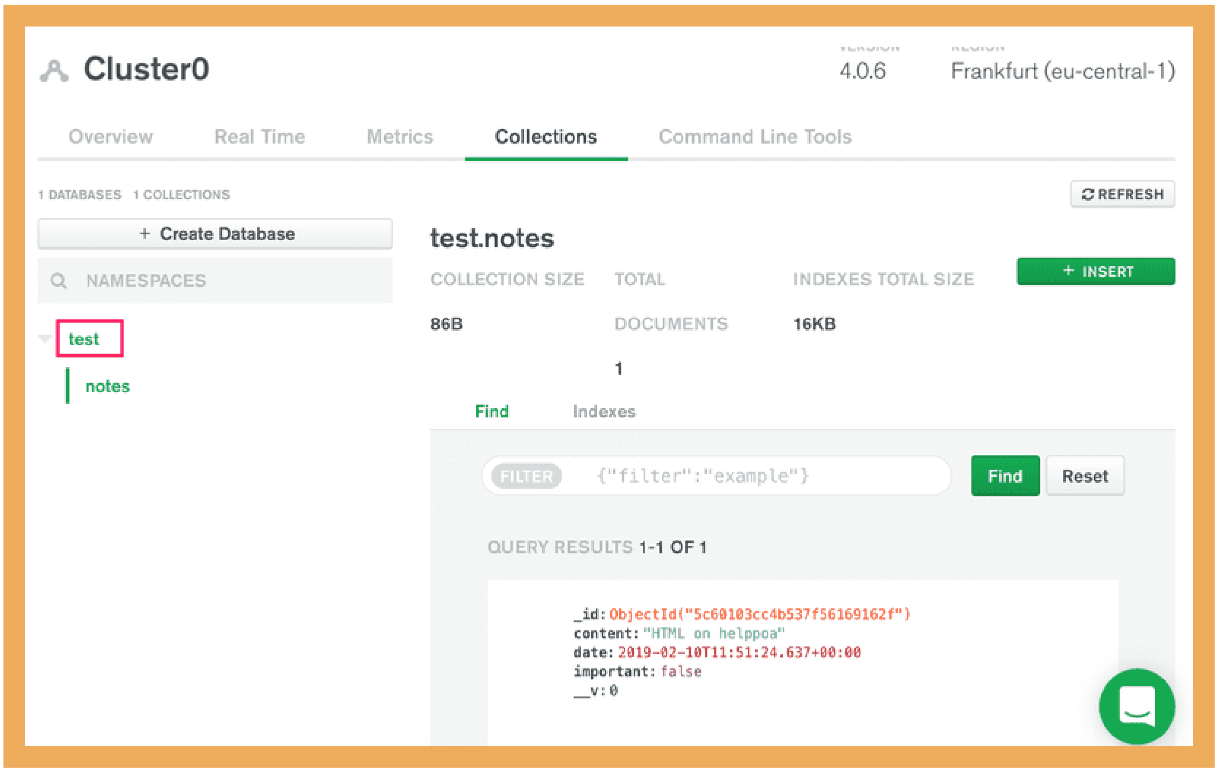
When the code is run with the command *node mongo.js password*, Mongo will add a new document to the database.

NB: Please note the password is the password created for the database user, not your MongoDB Atlas password. Also, if you created password with special characters, then you'll need to URL encode that password.

We can view the current state of the database from the MongoDB Atlas from *Collections*, in the Overview tab.

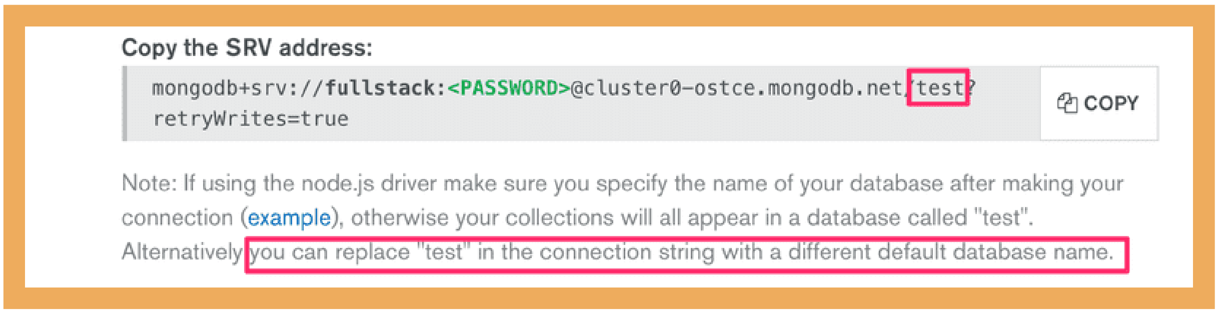


As the view states, the *document* matching the note has been added to the *notes* collection in the *test* database.



```
_id: ObjectId("5c60103cc4b537f56169162f")
content: "HTML on helppoa"
date: 2019-02-10T11:51:24.637+00:00
important: false
__v: 0
```

We should give a better name to the database. Like the documentation says, we can change the name of the database from the URI:



Copy the SRV address:

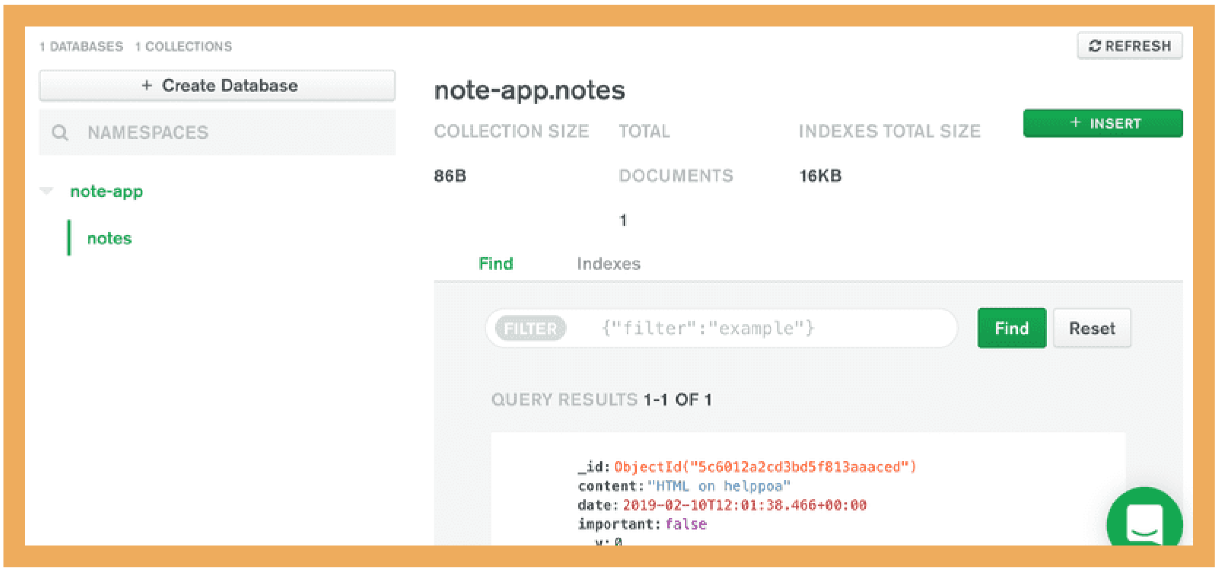
```
mongodb+srv://fullstack:<PASSWORD>@cluster0-ostce.mongodb.net/test?
retryWrites=true
```

Note: If using the node.js driver make sure you specify the name of your database after making your connection (example), otherwise your collections will all appear in a database called "test". Alternatively you can replace "test" in the connection string with a different default database name.

Let's destroy the *test* database. Let's now change the name of database referenced in our connection string to *note-app* instead, by modifying the URI:

```
mongodb+srv://fullstack:<PASSWORD>@cluster0-ostce.mongodb.net/note-app?retryWrites=true
```

Let's run our code again.



```
_id: ObjectId("5c6012a2cd3bd5f813aaaced")
content: "HTML on helppoa"
date: 2019-02-10T12:01:38.466+00:00
important: false
__v: 0
```

The data is now stored in the right database. The view also offers the *create database* functionality, that can be used to create new databases from the website. Creating the database like this is not necessary, since MongoDB Atlas automatically creates a new database when an application tries to connect to a database that does not exist yet.

Schema

After establishing the connection to the database, we define the schema for a note and the matching model:

```
const noteSchema = new mongoose.Schema({
  content: String,
  date: Date,
  important: Boolean,
})
```

```
const Note = mongoose.model('Note', noteSchema)
```

First we define the schema of a note that is stored in the `noteSchema` variable. The schema tells Mongoose how the note objects are to be stored in the database.

In the `Note` model definition, the first *"Note"* parameter is the singular name of the model. The name of the collection will be the lowercased plural *notes*, because the Mongoose convention is to automatically name collections as the plural (e.g. *notes*) when the schema refers to them in the singular (e.g. *Note*).

Document databases like Mongo are *schemaless*, meaning that the database itself does not care about the structure of the data that is stored in the database. It is possible to store documents with completely different fields in the same collection.

The idea behind Mongoose is that the data stored in the database is given a *schema at the level of the application* that defines the shape of the documents stored in any given collection.

Creating and saving objects

Next, the application creates a new note object with the help of the *Note* model:

```
const note = new Note({
  content: 'HTML is Easy',
  date: new Date(),
  important: false,
})
```

Models are so-called *constructor functions* that create new JavaScript objects based on the provided parameters. Since the objects are created with the model's constructor function, they have all the properties of the model, which include methods for saving the object to the database.

Saving the object to the database happens with the appropriately named `save` method, that can be provided with an event handler with the `then` method:

```
note.save().then(result => {
  console.log('note saved!')
  mongoose.connection.close()
})
```

When the object is saved to the database, the event handler provided to `then` gets called. The event handler closes the database connection with the command `mongoose.connection.close()`. If the connection is not closed, the program will never finish its execution.

The result of the save operation is in the `result` parameter of the event handler. The result is not that interesting when we're storing one object to the database. You can print the object to the console if you want to take a closer look at it while implementing your application or during debugging.

Let's also save a few more notes by modifying the data in the code and by executing the program again.

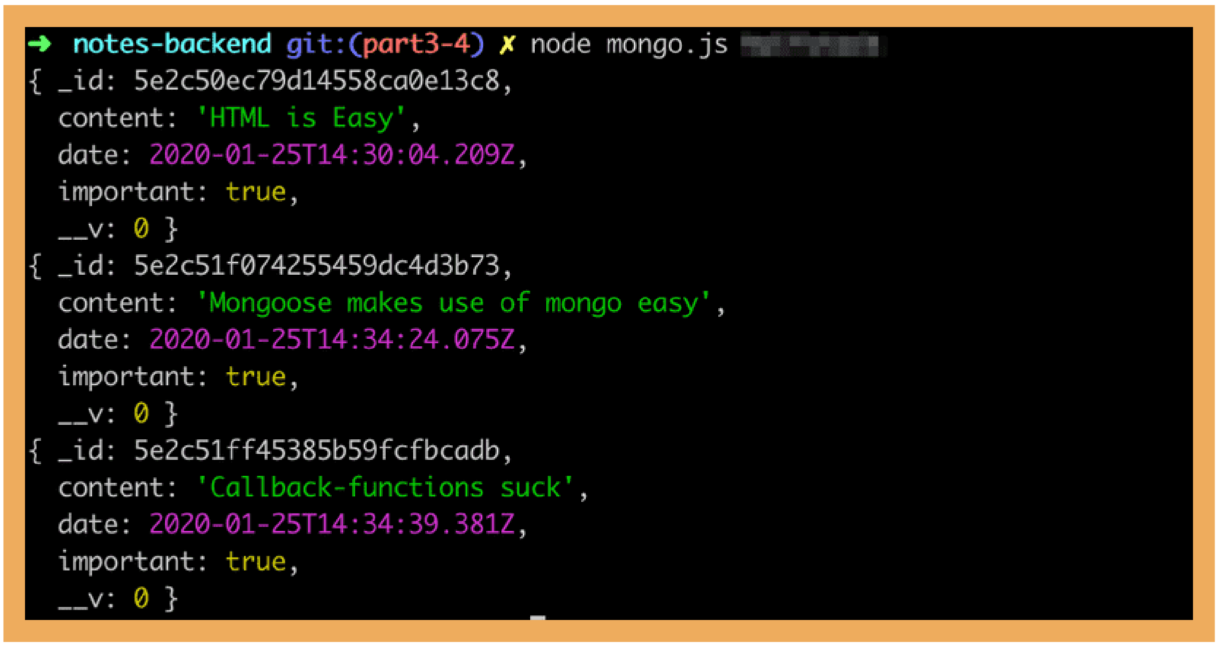
NB: Unfortunately the Mongoose documentation is not very consistent, with parts of it using callbacks in its examples and other parts, other styles, so it is not recommended to copy paste code directly from there. Mixing promises with old-school callbacks in the same code is not recommended.

Fetching objects from the database

Let's comment out the code for generating new notes and replace it with the following:

```
Note.find({}).then(result => {
  result.forEach(note => {
    console.log(note)
  })
  mongoose.connection.close()
})
```

When the code is executed, the program prints all the notes stored in the database:



```
➜ notes-backend git:(part3-4) ✗ node mongo.js [redacted]
{ _id: 5e2c50ec79d14558ca0e13c8,
  content: 'HTML is Easy',
  date: 2020-01-25T14:30:04.209Z,
  important: true,
  __v: 0 }
{ _id: 5e2c51f074255459dc4d3b73,
  content: 'Mongoose makes use of mongo easy',
  date: 2020-01-25T14:34:24.075Z,
  important: true,
  __v: 0 }
{ _id: 5e2c51ff45385b59fcfbcadb,
  content: 'Callback-functions suck',
  date: 2020-01-25T14:34:39.381Z,
  important: true,
  __v: 0 }
```

The objects are retrieved from the database with the find method of the `Note` model. The parameter of the method is an object expressing search conditions. Since the parameter is an empty object `{}`, we get all of the notes stored in the `notes` collection.

The search conditions adhere to the Mongo search query syntax.

We could restrict our search to only include important notes like this:

```
Note.find({ important: true }).then(result => {
  // ...
})
```

Exercise 3.12.

3.12: Command-line database

Create a cloud-based MongoDB database for the phonebook application with MongoDB Atlas.

Create a *mongo.js* file in the project directory, that can be used for adding entries to the phonebook, and for listing all of the existing entries in the phonebook.

NB: Do not include the password in the file that you commit and push to GitHub!

The application should work as follows. You use the program by passing three command-line arguments (the first is the password), e.g.:

```
node mongo.js yourpassword Anna 040-1234556
```

As a result, the application will print:

```
added Anna number 040-1234556 to phonebook
```

The new entry to the phonebook will be saved to the database. Notice that if the name contains whitespace characters, it must be enclosed in quotes:

```
node mongo.js yourpassword "Arto Vihavainen" 040-1234556
```

If the password is the only parameter given to the program, meaning that it is invoked like this:

```
node mongo.js yourpassword
```

Then the program should display all of the entries in the phonebook:

```
phonebook:
Anna 040-1234556
Arto Vihavainen 045-1232456
Ada Lovelace 040-1231236
```

You can get the command-line parameters from the process.argv variable.

NB: do not close the connection in the wrong place. E.g. the following code will not work:

```
Person
  .find({})
  .then(persons=> {
    // ...
  })

mongoose.connection.close()
```

In the code above the *mongoose.connection.close()* command will get executed immediately after the *Person.find* operation is started. This means that the database connection will be closed immediately, and the execution will never get to the point where *Person.find* operation finishes and the *callback* function gets called.

The correct place for closing the database connection is at the end of the callback function:

```
Person
  .find({})
  .then(persons=> {
    // ...
    mongoose.connection.close()
  })
```

NB: If you define a model with the name *Person*, mongoose will automatically name the associated collection as *people*.

Backend connected to a database

Now we have enough knowledge to start using Mongo in our application.

Let's get a quick start by copy pasting the Mongoose definitions to the *index.js* file:

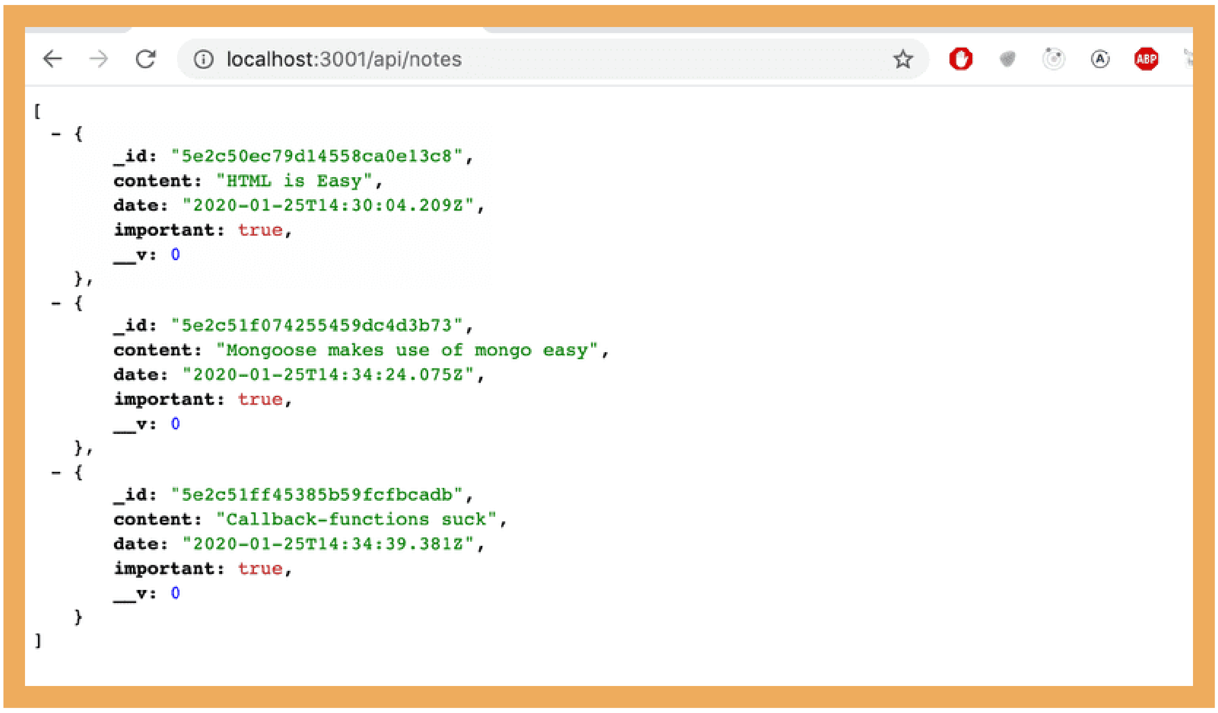
```
const mongoose = require('mongoose')

// DO NOT SAVE YOUR PASSWORD TO GITHUB!!
```

```
const url =
  'mongodb+srv://fullstack:sekred@cluster0-ostce.mongodb.net/note-app?retryWrites=true'

mongoose.connect(url, { useNewUrlParser: true, useUnifiedTopology: true })

const noteSchema = new mongoose.Schema({
  content: String,
  date: Date,
  important: Boolean,
})

const Note = mongoose.model('Note', noteSchema)
```

Let's change the handler for fetching all notes to the following form:

```
app.get('/api/notes', (request, response) => {
  Note.find({}).then(notes => {
    response.json(notes)
  })
})
```

We can verify in the browser that the backend works for displaying all of the documents:



```
[
  - {
        _id: "5e2c50ec79d14558ca0e13c8",
        content: "HTML is Easy",
        date: "2020-01-25T14:30:04.209Z",
        important: true,
        __v: 0
    },
  - {
        _id: "5e2c51f074255459dc4d3b73",
        content: "Mongoose makes use of mongo easy",
        date: "2020-01-25T14:34:24.075Z",
        important: true,
        __v: 0
    },
  - {
        _id: "5e2c51ff45385b59fcfbcadb",
        content: "Callback-functions suck",
        date: "2020-01-25T14:34:39.381Z",
        important: true,
        __v: 0
    }
]
```

The application works almost perfectly. The frontend assumes that every object has a unique id in the *id* field. We also don't want to return the mongo versioning field *__v* to the frontend.

One way to format the objects returned by Mongoose is to modify the `toJSON` method of the schema, which is used on all instances of the models produced with that schema. Modifying the method works like this:

```
noteSchema.set('toJSON', {
  transform: (document, returnedObject) => {
    returnedObject.id = returnedObject._id.toString()
    delete returnedObject._id
    delete returnedObject.__v
  }
})
```

Even though the *_id* property of Mongoose objects looks like a string, it is in fact an object. The `toJSON` method we defined transforms it into a string just to be safe. If we didn't make this change, it would cause more harm for us in the future once we start writing tests.

Let's respond to the HTTP request with a list of objects formatted with the `toJSON` method:

```
app.get('/api/notes', (request, response) => {
  Note.find({}).then(notes => {
    response.json(notes)
  })
})
```

Now the `notes` variable is assigned to an array of objects returned by Mongo. When the response is sent in the JSON format, the `toJSON` method of each object in the array is called automatically by the JSON.stringify method.

Database configuration into its own module

Before we refactor the rest of the backend to use the database, let's extract the Mongoose specific code into its own module.

Let's create a new directory for the module called *models*, and add a file called *note.js*:

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

const url = process.env.MONGODB_URI

console.log('connecting to', url)

mongoose.connect(url, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(result => {
    console.log('connected to MongoDB')
  })
  .catch((error) => {
    console.log('error connecting to MongoDB:', error.message)
  })

const noteSchema = new mongoose.Schema({
  content: String,
  date: Date,
  important: Boolean,
})

noteSchema.set('toJSON', {
  transform: (document, returnedObject) => {
    returnedObject.id = returnedObject._id.toString()
    delete returnedObject._id
    delete returnedObject.__v
  }
})

module.exports = mongoose.model('Note', noteSchema)
```

Defining Node modules differs slightly from the way of defining ES6 modules in part 2.

The public interface of the module is defined by setting a value to the `module.exports` variable. We will set the value to be the *Note* model. The other things defined inside of the module, like the variables `mongoose` and `url` will not be accessible or visible to users of the module.

Importing the module happens by adding the following line to *index.js*:

```
const Note = require('./models/note')
```

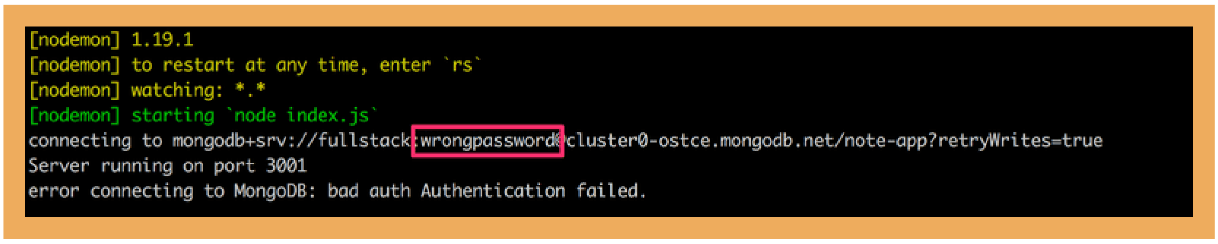
This way the `Note` variable will be assigned to the same object that the module defines.

The way that the connection is made has changed slightly:

```
const url = process.env.MONGODB_URI

console.log('connecting to', url)

mongoose.connect(url, { useNewUrlParser: true, useUnifiedTopology: true })
  .then(result => {
    console.log('connected to MongoDB')
  })
  .catch((error) => {
    console.log('error connecting to MongoDB:', error.message)
  })
```

It's not a good idea to hardcode the address of the database into the code, so instead the address of the database is passed to the application via the `MONGODB_URI` environment variable.

The method for establishing the connection is now given functions for dealing with a successful and unsuccessful connection attempt. Both functions just log a message to the console about the success status:



```
[nodemon] 1.19.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node index.js`
connecting to mongodb+srv://fullstack:wrongpassword@cluster0-ostce.mongodb.net/note-app?retryWrites=true
Server running on port 3001
error connecting to MongoDB: bad auth Authentication failed.
```

There are many ways to define the value of an environment variable. One way would be to define it when the application is started:

```
MONGODB_URI=address_here npm run watch
```

A more sophisticated way is to use the dotenv library. You can install the library with the command:

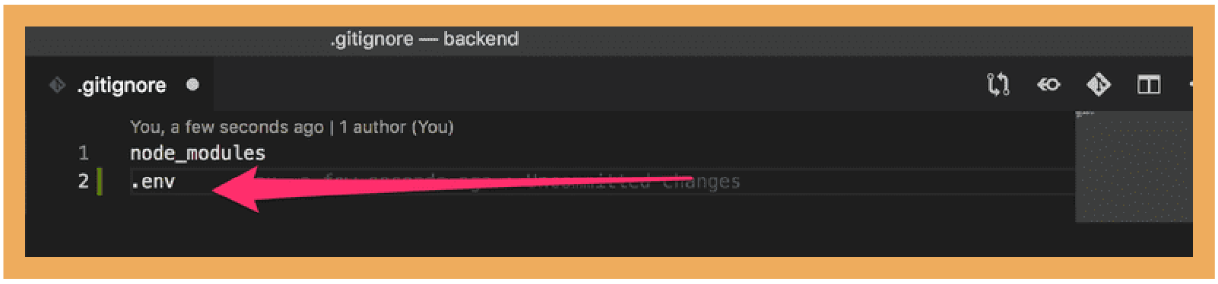
```
npm install dotenv --save
```

To use the library, we create a *.env* file at the root of the project. The environment variables are defined inside of the file, and it can look like this:

```
MONGODB_URI='mongodb+srv://fullstack:sekred@cluster0-ostce.mongodb.net/note-app?retryWrit
PORT=3001
```

We also added the hardcoded port of the server into the `PORT` environment variable.

The *.env* file should be gitignored right away, since we do not want to publish any confidential information publicly online!



```
node_modules
.env
```

The environment variables defined in the *.env* file can be taken into use with the expression `require('dotenv').config()` and you can reference them in your code just like you would reference normal environment variables, with the familiar `process.env.MONGODB_URI` syntax.

Let's change the *index.js* file in the following way:

```
require('dotenv').config()
const express = require('express')
const app = express()
const Note = require('./models/note')

// ..

const PORT = process.env.PORT
app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`)
})
```

It's important that *dotenv* gets imported before the *note* model is imported. This ensures that the environment variables from the *.env* file are available globally before the code from the other modules is imported.

Using database in route handlers

Next, let's change the rest of the backend functionality to use the database.

Creating a new note is accomplished like this:

```
app.post('/api/notes', (request, response) => {
  const body = request.body

  if (body.content === undefined) {
    return response.status(400).json({ error: 'content missing' })
  }

  const note = new Note({
    content: body.content,
    important: body.important || false,
    date: new Date(),
  })

  note.save().then(savedNote => {
    response.json(savedNote)
  })
})
```

The note objects are created with the `Note` constructor function. The response is sent inside of the callback function for the `save` operation. This ensures that the response is sent only if the operation succeeded. We will discuss error handling a little bit later.

The `savedNote` parameter in the callback function is the saved and newly created note. The data sent back in the response is the formatted version created with the `toJSON` method:

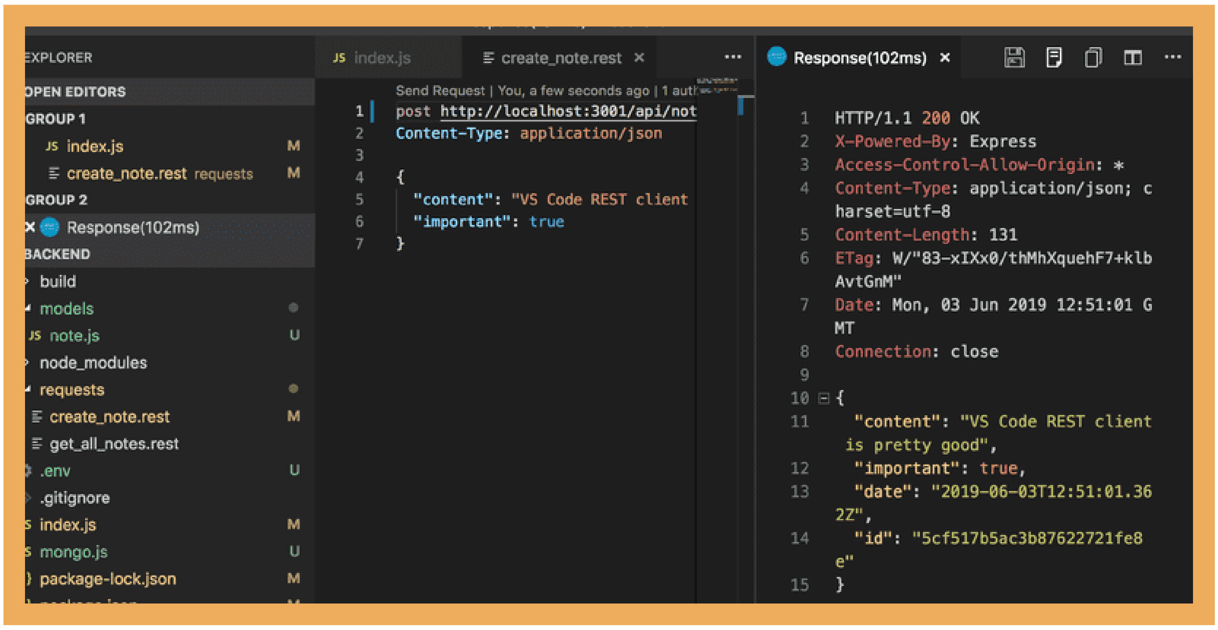
```
response.json(savedNote)
```

Fetching an individual note gets changed into the following:

```
app.get('/api/notes/:id', (request, response) => {
  Note.findById(request.params.id).then(note => {
    response.json(note)
  })
})
```

Verifying frontend and backend integration

When the backend gets expanded, it's a good idea to test the backend first with **the browser, Postman or the VS Code REST client**. Next, let's try creating a new note after taking the database into use:



```
post http://localhost:3001/api/not
Content-Type: application/json

{
  "content": "VS Code REST client
  "important": true
}
```

```
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
Content-Type: application/json; c
harset=utf-8
Content-Length: 131
ETag: W/"83-xIXx0/thMhXquehF7+klb
AvtGnM"
Date: Mon, 03 Jun 2019 12:51:01 G
MT
Connection: close

{
  "content": "VS Code REST client
 is pretty good",
  "important": true,
  "date": "2019-06-03T12:51:01.36
2Z",
  "id": "5cf517b5ac3b87622721fe8
e"
}
```

Only once everything has been verified to work in the backend, is it a good idea to test that the frontend works with the backend. It is highly inefficient to test things exclusively through the frontend.

It's probably a good idea to integrate the frontend and backend one functionality at a time. First, we could implement fetching all of the notes from the database and test it through the backend endpoint in the browser. After this, we could verify that the frontend works with the new backend. Once everything seems to work, we would move onto the next feature.

Once we introduce a database into the mix, it is useful to inspect the state persisted in the database, e.g. from the control panel in MongoDB Atlas. Quite often little Node helper programs like the *mongo.js* program we wrote earlier can be very helpful during development.

You can find the code for our current application in its entirety in the *part3-4* branch of this Github repository.

Exercises 3.13.-3.14.

The following exercises are pretty straightforward, but if your frontend stops working with the backend, then finding and fixing the bugs can be quite interesting.

3.13: Phonebook database, step1

Change the fetching of all phonebook entries so that the data is *fetched from the database*.

Verify that the frontend works after the changes have been made.

In the following exercises, write all Mongoose-specific code into its own module, just like we did in the chapter Database configuration into its own module.

3.14: Phonebook database, step2

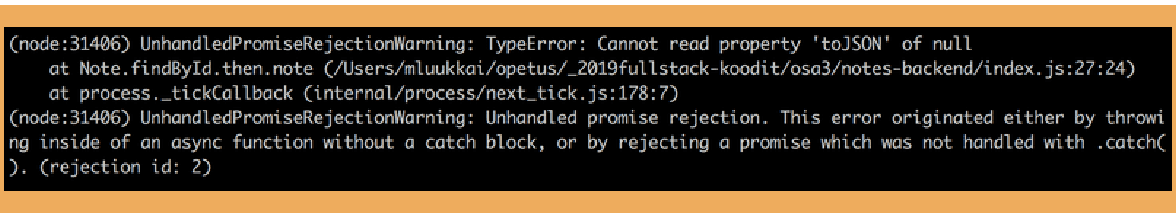
Change the backend so that new numbers are *saved to the database*. Verify that your frontend still works after the changes.

At this point, you can choose to simply allow users to create all phonebook entries. At this stage, the phonebook can have multiple entries for a person with the same name.

Error handling

If we try to visit the URL of a note with an id that does not actually exist e.g. http://localhost:3001/api/notes/5c41c90e84d891c15dfa3431 where *5c41c90e84d891c15dfa3431* is not an id stored in the database, then the browser will simply get "stuck" since the server never responds to the request.

We can see the following error message appear in the logs for the backend:


```
(node:31406) UnhandledPromiseRejectionWarning: TypeError: Cannot read property 'toJSON' of null
    at Note.findById.then.note (/Users/mluukkai/opetus/_2019fullstack-koodit/osa3/notes-backend/index.js:27:24)
    at process._tickCallback (internal/process/next_tick.js:178:7)
(node:31406) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either by throwi
ng inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(
). (rejection id: 2)
```

The request has failed and the associated Promise has been *rejected*. Since we don't handle the rejection of the promise, the request never gets a response. In part 2, we already acquainted ourselves with handling errors in promises.

Let's add a simple error handler:

```
app.get('/api/notes/:id', (request, response) => {
  Note.findById(request.params.id)
    .then(note => {
      response.json(note)
    })
    .catch(error => {
      console.log(error)
      response.status(404).end()
    })
})
```

Every request that leads to an error will be responded to with the HTTP status code 404 not found. The console displays more detailed information about the error.

There's actually two different types of error situations. In one of those situations, we are trying to fetch a note with a wrong kind of `id`, meaning an `id` that doesn't match the mongo identifier format.

If we make the following request, we will get the error message shown below:

```
Method: GET
Path:   /api/notes/someInvalidId
Body:   {}
---
{ CastError: Cast to ObjectId failed for value "someInvalidId" at path "_id"
    at CastError (/Users/mluukkai/opetus/_fullstack/osa3-muisiinpanot/node_modules
    at ObjectId.cast (/Users/mluukkai/opetus/_fullstack/osa3-muisiinpanot/node_mod
    ...
```

The other error situation happens when the id is in the correct format, but no note is found in the database for that id. In this case the value of `note` is `null` and the response body will be empty. We should distinguish between these two different types of error situations. The latter is in fact an error caused by our own code.

Let's change the code in the following way:

```
app.get('/api/notes/:id', (request, response) => {
  Note.findById(request.params.id)
    .then(note => {
      if (note) {
        response.json(note)
      } else {
        response.status(404).end()
      }
    })
    .catch(error => {
      console.log(error)
      response.status(400).send({ error: 'malformatted id' })
    })
})
```

If no matching object is found in the database, the value of `note` will be undefined and the `else` block is executed. This results in a response with the status code *404 not found*.

If the format of the id is incorrect, then we will end up in the error handler defined in the `catch` block. The appropriate status code for the situation is 400 Bad Request, because the situation fits the description perfectly:

> *The request could not be understood by the server due to malformed syntax. The client SHOULD NOT repeat the request without modifications.*

We have also added some data to the response to shed some light on the cause of the error.

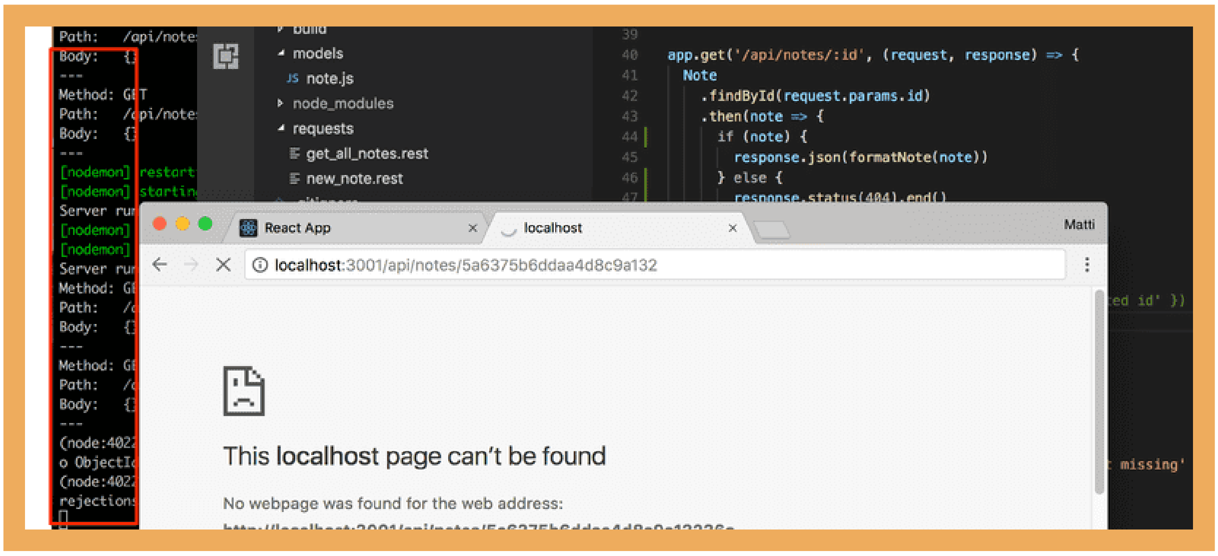
When dealing with Promises, it's almost always a good idea to add error and exception handling, because otherwise you will find yourself dealing with strange bugs.

It's never a bad idea to print the object that caused the exception to the console in the error handler:

```
.catch(error => {
  console.log(error)
  response.status(400).send({ error: 'malformatted id' })
})
```

The reason the error handler gets called might be something completely different than what you had anticipated. If you log the error to the console, you may save yourself from long and frustrating debugging sessions. Moreover, most modern services to where you deploy your application support some form of logging system that you can use to check these logs. As mentioned, Heroku is one.

Every time you're working on a project with a backend, *it is critical to keep an eye on the console output of the backend*. If you are working on a small screen, it is enough to just see a tiny slice of the output in the background. Any error messages will catch your attention even when the console is far back in the background:



```
Path:   /api/note
Body:   {}
---
Method: GET
Path:   /api/note
Body:   {}
---
[nodemon] restart
[nodemon] startin
Server rur
[nodemon]
[nodemon]
Server rur
Method: GE
Path:   /
Body:   {}
---
Method: GE
Path:   /
Body:   {}
---
(node:402
o ObjectI
(node:402
rejections
```

```
app.get('/api/notes/:id', (request, response) => {
  Note
    .findById(request.params.id)
    .then(note => {
      if (note) {
        response.json(formatNote(note))
      } else {
        response.status(404).end()
```

This localhost page can't be found

No webpage was found for the web address:

Moving error handling into middleware

We have written the code for the error handler among the rest of our code. This can be a reasonable solution at times, but there are cases where it is better to implement all error handling in a single place. This can be particularly useful if we later on want to report data related to errors to an external error tracking system like Sentry.

Let's change the handler for the */api/notes/:id* route, so that it passes the error forward with the `next` function. The next function is passed to the handler as the third parameter:

```
app.get('/api/notes/:id', (request, response, next) => {
  Note.findById(request.params.id)
    .then(note => {
      if (note) {
        response.json(note)
      } else {
        response.status(404).end()
      }
    })
    .catch(error => next(error))
})
```

The error that is passed forwards is given to the `next` function as a parameter. If `next` was called without a parameter, then the execution would simply move onto the next route or middleware. If the `next` function is called with a parameter, then the execution will continue to the *error handler middleware*.

Express error handlers are middleware that are defined with a function that accepts *four parameters*. Our error handler looks like this:

```
const errorHandler = (error, request, response, next) => {
  console.error(error.message)

  if (error.name === 'CastError') {
    return response.status(400).send({ error: 'malformatted id' })
  }

  next(error)
}

app.use(errorHandler)
```

The error handler checks if the error is a *CastError* exception, in which case we know that the error was caused by an invalid object id for Mongo. In this situation the error handler will send a response to the browser with the response object passed as a parameter. In all other error situations, the middleware passes the error forward to the default Express error handler.

The order of middleware loading

The execution order of middleware is the same as the order that they are loaded into express with the `app.use` function. For this reason it is important to be careful when defining middleware.

The correct order is the following:

```
app.use(express.static('build'))
app.use(express.json())
app.use(logger)

app.post('/api/notes', (request, response) => {
  const body = request.body
  // ...
})

const unknownEndpoint = (request, response) => {
  response.status(404).send({ error: 'unknown endpoint' })
}

// handler of requests with unknown endpoint
app.use(unknownEndpoint)

const errorHandler = (error, request, response, next) => {
  // ...
}

// handler of requests with result to errors
app.use(errorHandler)
```

The json-parser middleware should be among the very first middleware loaded into Express. If the order was the following:

```
app.use(logger) // request.body is empty!

app.post('/api/notes', (request, response) => {
  // request.body is empty!
  const body = request.body
  // ...
})

app.use(express.json())
```

Then the JSON data sent with the HTTP requests would not be available for the logger middleware or the POST route handler, since the `request.body` would be an empty object at that point.

It's also important that the middleware for handling unsupported routes is next to the last middleware that is loaded into Express, just before the error handler.

For example, the following loading order would cause an issue:

```
const unknownEndpoint = (request, response) => {
  response.status(404).send({ error: 'unknown endpoint' })
}

// handler of requests with unknown endpoint
app.use(unknownEndpoint)

app.get('/api/notes', (request, response) => {
  // ...
})
```

Now the handling of unknown endpoints is ordered *before the HTTP request handler*. Since the unknown endpoint handler responds to all requests with *404 unknown endpoint*, no routes or middleware will be called after the response has been sent by unknown endpoint middleware. The only exception to this is the error handler which needs to come at the very end, after the unknown endpoints handler.

Other operations

Let's add some missing functionality to our application, including deleting and updating an individual note.

The easiest way to delete a note from the database is with the findByIdAndRemove method:

```
app.delete('/api/notes/:id', (request, response, next) => {
  Note.findByIdAndRemove(request.params.id)
    .then(result => {
      response.status(204).end()
    })
    .catch(error => next(error))
})
```

In both of the "successful" cases of deleting a resource, the backend responds with the status code *204 no content*. The two different cases are deleting a note that exists, and deleting a note that does not exist in the database. The `result` callback parameter could be used for checking if a resource actually was deleted, and we could use that information for returning different status codes for the two cases if we deemed it necessary. Any exception that occurs is passed onto the error handler.

The toggling of the importance of a note can be easily accomplished with the findByIdAndUpdate method.

```
app.put('/api/notes/:id', (request, response, next) => {
  const body = request.body

  const note = {
    content: body.content,
    important: body.important,
  }

  Note.findByIdAndUpdate(request.params.id, note, { new: true })
    .then(updatedNote => {
      response.json(updatedNote)
    })
    .catch(error => next(error))
})
```

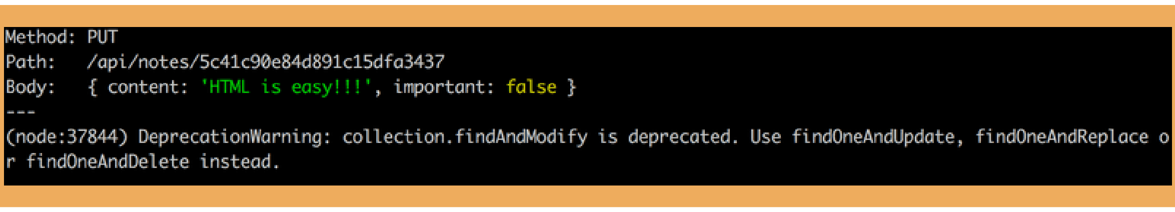
In the code above, we also allow the content of the note to be edited. However, we will not support changing the creation date for obvious reasons.

Notice that the `findByIdAndUpdate` method receives a regular JavaScript object as its parameter, and not a new note object created with the `Note` constructor function.

There is one important detail regarding the use of the `findByIdAndUpdate` method. By default, the `updatedNote` parameter of the event handler receives the original document without the modifications. We added the optional `{ new: true }` parameter, which will cause our event handler to be called with the new modified document instead of the original.

After testing the backend directly with Postman and the VS Code REST client, we can verify that it seems to work. The frontend also appears to work with the backend using the database.

When we toggle the importance of a note, we see the following worrisome error message in the console:



```
Method: PUT
Path:   /api/notes/5c41c90e84d891c15dfa3437
Body:   { content: 'HTML is easy!!!', important: false }
---
(node:37844) DeprecationWarning: collection.findAndModify is deprecated. Use findOneAndUpdate, findOneAndReplace o
r findOneAndDelete instead.
```

Googling the error message will lead to instructions for fixing the problem. Following the suggestion in the Mongoose documentation, we add the following line to the *note.js* file:

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

mongoose.set('useFindAndModify', false)

// ...

module.exports = mongoose.model('Note', noteSchema)
```

You can find the code for our current application in its entirety in the *part3-5* branch of this github repository.

Exercises 3.15.-3.18.

3.15: Phonebook database, step3

Change the backend so that deleting phonebook entries is reflected in the database.

Verify that the frontend still works after making the changes.

3.16: Phonebook database, step4

Move the error handling of the application to a new error handler middleware.

3.17*: Phonebook database, step5

If the user tries to create a new phonebook entry for a person whose name is already in the phonebook, the frontend will try to update the phone number of the existing entry by making an HTTP PUT request to the entry's unique URL.

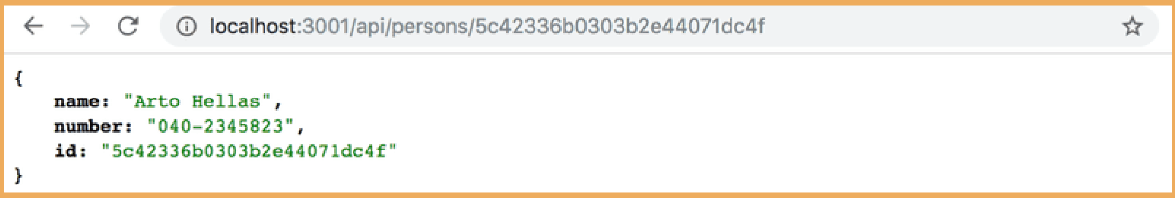
Modify the backend to support this request.

Verify that the frontend works after making your changes.

3.18*: Phonebook database step6

Also update the handling of the *api/persons/:id* and *info* routes to use the database, and verify that they work directly with the browser, Postman, or VS Code REST client.

Inspecting an individual phonebook entry from the browser should look like this:



```
{
   name: "Arto Hellas",
   number: "040-2345823",
   id: "5c42336b0303b2e44071dc4f"
}
```

Propose changes to material

< Part 3b
Previous part

Part 3d >
Next part

About course

Course contents

FAQ

Partners